UNCLASSIFIED

DTIC FILE COPY

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A211 643

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Ada Compiler Validation Summary Report: Rational Rational Environment, Version D_12_0_0, R1000 Series 200 Model 10 (Host & Target), 890601W1.10084

5. TYPE OF REPORT & PERIOD COVERED
01 June 89 - 1 Dec 90

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Wright-Patterson AFB
Dayton, OH, USA

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS
Wright-Patterson AFB
Dayton, OH, USA

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office)
Wright-Patterson AFB
Dayton, OH, USA

15. SECURITY CLASS (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

DTIC ELECTE AUG 22 1989 S B D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Rational, Rational Environment, Version D_12_0_0, Wright-Patterson AFB, R1000 Series 200 Model 10 under Rational Environment, D_12_0_0 (Host & Target), ACVC 1.10

89 8 21 191

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AVF Control Number: AVF-VSR-288.0689
89-03-16-RAT

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890601W1.10084
Rational
Rational Environment, Version D_12_0_0
R1000 Series 200 Host Model 10 and R1000 Series Model 10 200 Target

Completion of On-Site Testing:
01 June 1989

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: Rational Environment, Version D_12_0_0

Certificate Number: 890601W1.10084

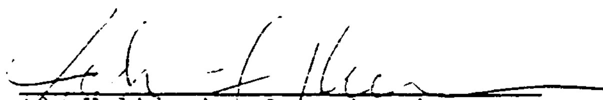
Host: R1000 Series 200 Model 10 under
Rational Environment, D_12_0_0

Target: R1000 Series 200 Model 10 under
Rational Environment, D_12_0_0

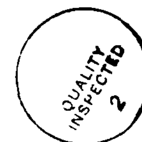
Testing Completed 01 June 1989 Using ACVC 1.10

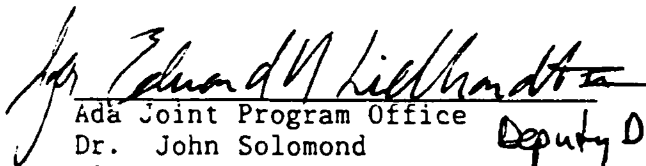
This report has been reviewed and is approved.

Ada Validation Facility
Steve P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503



Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311





Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

Deputy Director

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist Avail and/or Special
A-1

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT 1-2
1.2 USE OF THIS VALIDATION SUMMARY REPORT 1-2
1.3 REFERENCES. 1-3
1.4 DEFINITION OF TERMS 1-3
1.5 ACVC TEST CLASSES 1-4

CHAPTER 2 CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED. 2-1
2.2 IMPLEMENTATION CHARACTERISTICS. 2-2

CHAPTER 3 TEST INFORMATION

3.1 TEST RESULTS. 3-1
3.2 SUMMARY OF TEST RESULTS BY CLASS. 3-1
3.3 SUMMARY OF TEST RESULTS BY CHAPTER. 3-2
3.4 WITHDRAWN TESTS 3-2
3.5 INAPPLICABLE TESTS. 3-2
3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS. . 3-7
3.7 ADDITIONAL TESTING INFORMATION. 3-8
3.7.1 Prevalidation 3-8
3.7.2 Test Method 3-8
3.7.3 Test Site 3-9

APPENDIX A *DECLARATION OF CONFORMANCE*

APPENDIX B APPENDIX F OF THE Ada STANDARD

APPENDIX C TEST PARAMETERS

APPENDIX D WITHDRAWN TESTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it *must be understood that some* differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc., under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 01 June 89 at Santa Clara CA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO 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 compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

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

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC — The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.

Ada Commentary — An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Ada Standard — ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant — The agency requesting validation.

AVF — The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.

AVO — The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler — A processor for the Ada language. In the context of this report, a compiler is any language processor, including

cross-compilers, translators, and interpreters.

Failed test — An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host — The computer on which the compiler resides.

Inapplicable test — An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test — An ACVC test for which a compiler generates the expected result.

Target — The computer for which a compiler generates code.

Test — A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn test — An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada 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, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation of legal Ada programs with certain language constructs which cannot be verified at compile time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

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 every syntax or semantic error in the test is detected. A Class B test is passed if every

illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It 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 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. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate

tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Rational Environment, Version D_12_0_0

ACVC Version: 1.10

Certificate Number: 890601W1.10084

Host Computer:

Machine:	R1000 Series 200 Model 10
Operating System:	Rational Environment D_12_0_0
Memory Size:	32 MB

Target Computer:

Machine:	R1000 Series 200 Model 10
Operating System:	Rational Environment D_12_0_0
Memory Size:	32 MB

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler rejects tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 10 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined type LONG_INTEGER. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

(4) No exception is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) Sometimes NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is not gradual. (See tests C45524A..Z.)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round away from zero. (See tests C46012A..Z.)

(2) The method used for rounding to longest integer is round away from zero. (See tests C46012A..Z.)

(3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR sometimes. (See test C36003A.)

(2) No exception is raised when 'LENGTH is applied to a null array type with INTEGER'LAST + 2 components. (See test C36202A.)

(3) NUMERIC_ERROR is raised when a null array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises STORAGE_ERROR when the array objects are declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

(7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(8) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, the test results indicate that all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised before all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

(1) The pragma INLINE is not supported for functions or procedures. (See tests LA3004A..B, EA3004C..D, and CA3004E..F.)

i. Generics

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output

(1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

(2) The package DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

(4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J, CE2102R, CE2102T, and CE2102V.)

(5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K.)

(6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

(7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

(8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G, CE3104C, CE3110A, and CE3114A.)

(9) Overwriting to a sequential file truncates to the last element written. (See test CE2208B.)

(10) Temporary sequential files are given names and not deleted when closed. (See test CE2108A.)

(11) Temporary direct files are given names and not deleted when closed. (See test CE2108C.)

(12) Temporary text files are given names and not deleted when closed. (See test CE3112A.)

(13) More than one internal file can be associated with each external file for sequential files when reading only. (See tests CE2107A..E, CE2102L, CE2110B, and CE2111D.)

(14) More than one internal file can be associated with each external file for direct files when reading only. (See tests CE2107F..H (3 tests), CE2110D, and CE2111H.)

(15) More than one internal file can be associated with each external file for text files when reading only. (See tests CE3111A..E, CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 43 tests had been withdrawn because of test errors. The AVF determined that 487 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 74 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	115	1130	1864	14	20	44	3187
Inapplicable	14	8	452	3	8	2	487
Withdrawn	1	2	34	0	6	0	43
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	197	561	549	243	171	99	157	332	131	36	252	180	279	3187
Inappl	15	88	131	5	1	0	9	1	6	0	0	189	42	487
Wdrn	1	1	0	0	0	0	0	1	0	0	1	35	4	43
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 43 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	BC3009B	CD2A62D	CD2A63A
CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B	CD2A66C
CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D	CD2A76A
CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G	CD2A84M
CD2A84N	CD2B15C	CD2D11B	CD5007B	CD50110	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	ED7004B	ED7005C
ED7005D	ED7006C	ED7006D	CE2107I	CE3111C	CE3301A
CE3411B					

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 487 tests were inapplicable for the reasons indicated:

a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y	C35705L..Y	C35706L..Y	C35707L..Y
C35708L..Y	C35802L..Z	C45241L..Y	C45321L..Y
C45421L..Y	C45521L..Z	C45524L..Z	C45621L..Z

C45641L..Y   C46012L..Z

b. E24201A, C45232A, and D4A004B are ruled not applicable by the AVO because static universal expressions with values that lie outside of the range SYSTEM.MIN_INT ... SYSTEM.MAX_INT are rejected.

c. The following 54 tests are not applicable because they include enumeration representation clauses in which the specified representation values are not contiguous. Under the terms of AI-00325, this implementation is not required to support such representation clauses:

C35502I	C35502J	C35502M	C35502N	C35507I
C35507J	C35507M	C35507N	C35508I	C35508J
C35508M	C55B16A	C35508N	A39005F	AD1009M
AD1009V	AD1009W	AD1C04D	CD1C03G	CD3021A
CD2A23A..E (5 tests)		CD2A24A..J (10 tests)		ED2A26A
AD3014C	AD3014F	AD3015C	AD3015F	AD3015H
AD3015K	CD3014A	CD3014B	CD3014D	CD3014E
CD3015A	CD3015B	CD3015D	CD3015E	CD3015G
CD3015I	CD3015J	CD3015L		

d. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

e. C35702B and B86001U are not applicable because this implementation supports no predefined type LONG_FLOAT.

f. A39005D, C87B62D, CD1009K, CD1009T, CD1009U, CD1C03E, CD1C04B, and CD1C06A are not applicable because this implementation does not support storage size representation clauses for task types.

g. The following 20 tests are not applicable because this implementation does not support small length clauses for fixed-point types:

A39005E	C87B62C	CD1009L	CD1C03F	ED2A56A
CD2A53A..E (5 tests)		CD2A54A..D (4 tests)		CD2D11A
CD2A54G..J (4 tests)		CD2D13A		

h. The following 16 tests are not applicable because this implementation does not support a predefined type SHORT_INTEGER:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	B86001V
CD7101E				

i. C4A013B is not applicable because the evaluation of an expression involving 'MACHINE_RADIX applied to the most precise floating-point type would raise an exception; since the expression must be static, it is rejected at compile time.

j. D56001B uses 65 levels of block nesting which exceeds the capacity of the compiler.

k. D64005G is not applicable because this implementation does not support 17 levels of static nesting.

l. B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

m. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

n. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

o. C87B62B is ruled not applicable by the AVO because this implementation has limited support of storage size representation clauses for access types.

p. C92005B is ruled not applicable by the AVO because the storage size attribute will not always yield a value in the range of STANDARD.INTEGER.

q. LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F are not applicable because this implementation does not support pragma INLINE.

r. CD1009C, CD2A41A..B (2 tests), CD2A41E, and CD2A42A..J (10 tests) are not applicable because this implementation does not support size clauses for floating point types.

s. The following 12 tests are not applicable because this implementation does not support record representation clauses:

CD1009N	CD1009X..Z (3 tests)	CD1C03H	CD1C04E
ED1D04A	CD4031A	CD4051A..C (3 tests)	CD7204C

t. CD1C04C is not applicable because this implementation does not support small representation clauses for fixed point types.

u. CD2A52A..D (4 tests) and CD2A52G..J (4 tests) are not applicable because the fixed-point size length clause is too small.

v. CD2A83A, CD2A84B, CD2A84E, CD2A84I, and CD2B11B are ruled not applicable by the AVO because the allocator raises a storage error because an insufficient amount of storage was specified. These tests assume that the value of STORAGE_UNIT is 8.

w. CD2C11A..E (5 tests) are not applicable because task storage size length clauses are not supported by this implementation.

x. CD4041A and CD4051D are not applicable because this implementation does not support record representation specifications.

y. The following 76 tests are not applicable because this implementation does not support address clauses:

CD5003B..I (8 tests)	CD5011A..I (9 tests)
CD5011K..N (4 tests)	CD5011Q..S (3 tests)
CD5012A..J (10 tests)	CD5012L..M (2 tests)
CD5013A..I (9 tests)	CD5013K..O (5 tests)
CD5013R..S (2 tests)	CD5014A..O (15 tests)
CD5014R..Z (9 tests)	

z. 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.

aa. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

ab. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

ac. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

ad. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

ae. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

af. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

ag. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

ah. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

ai. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

aj. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

ak. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

al. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

am. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

an. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

ao. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

ap. CE2107B..E (4 tests), CE2107L, and CE2110B are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

aq. CE2107G..H (2 tests), CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

ar. CE2111D is not applicable because this implementation does not support resetting of an external file from IN_FILE to OUT_FILE.

as. CE2201K is inapplicable because this implementation does not support I/O of access values.

at. CE3102E is inapplicable because this implementation supports CREATE with IN_FILE mode for text files.

au. CE3102F is inapplicable because this implementation supports RESET for text files.

av. CE3102G is inapplicable because this implementation supports deletion of an external file for text files.

aw. CE3102I is inapplicable because this implementation supports CREATE with OUT_FILE mode for text files.

ax. CE3102J is inapplicable because this implementation supports OPEN with IN_FILE mode for text files.

ay. CE3102K is inapplicable because this implementation supports OPEN with OUT_FILE mode for text files.

az. CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 76 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B22003B	B22004A	B22004B	B22004C	B23004A
B23004B	B24001A	B24001B	B24001C	B24005A	B24005B
B24007A	B24009A	B24204B	B24204C	B24204D	B25002B
B26001A	B26002A	B26005A	B28003A	B28003C	B29001A
B2A003B	B2A003C	B2A003D	B2A007A	B32103A	B33201B
B33202B	B33203B	B33301B	B35101A	B36002A	B36201A
B37205A	B37307B	B38003A	B38003B	B38009A	B38009B
B41201A	B41202A	B44001A	B44004B	B44004C	B45205A
B48002A	B48002D	B51001A	B51003A	B51003B	B53003A
B55A01A	B64001B	B64006A	B67001H	B74003A	B91001H
B95001C	B95003A	B95004A	B95079A	BB3005A	BC1303F
BC2001D	BC2001E	BC3003A	BC3003B	BC3005B	BC3013A
BD5008A					

C34006D required a modified evaluation because the test includes some comparisons that use the 'SIZE attribute under assumptions that are not fully supported by the Ada standard and are subject to ARG review. Thus, the AVO ruled that an implementation is considered to have passed this test if the only REPORT.FAILED output is "INCORRECT TYPE (or OBJECT) 'SIZE" (from lines 403 & 407, 462 & 466, and 219 & 228, respectively). This implementation produced "INCORRECT OBJECT'SIZE".

C45651A was required to be modified to correct an IF statement that uses a problematic range (line 231). The upper bound of the range was changed from 960.0 to 1024.0.

CE2401B was graded with a modifed evaluation of the results. Because this implementation raises USE_ERROR on attempts to read or write access values, CE2401B reports "NOT_APPLICABLE". However, because the test uses I/O of two other types in checking its objective, and these checks were passed, the AVO ruled that CE2401B was passed.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the Rational Environment was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Rational Environment using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	R1000 Series 200 Model 10
Host operating system:	Rational Environment, D_12_0_0
Target computer:	R1000 Series 200 Model 10
Target operating system:	Rational Environment, D_12_0_0
Compiler:	Rational Environment, Version D_12_0_0

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded onto a VAX 11/750 and transferred to the host via FTP.

After the test files were loaded to disk, the full set of tests was compiled and linked on the R1000 Series 200, and all executable tests were run on the R1000 Series 200. Results were printed from the host computer.

The compiler was tested using command scripts provided by Rational and reviewed by the validation team. The compiler was tested using all default option settings except for the following:

OPTION	EFFECT
DIRECTORY.CREATE_SUBPROGRAM_SPECS:=FALSE	Does not automatically generate subprogram specs for library subprogram bodies.
R1000_CG.RETAIN_DELTA0_COMPATIBILITY:=FALSE	Does not generate code that is compatible with a previous version (DELTA0) of the compiler.

Tests were compiled, linked, and executed (as appropriate) using a single host computer and R1000 Series 200 computers. Test output, compilation 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.

3.7.3 Test Site

Testing was conducted at Santa Clara CA and was completed on 01 June 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Rational has submitted the following Declaration of Conformance concerning the Rational Environment.

DECLARATION OF CONFORMANCE

Compiler Implementor: Rational
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.10

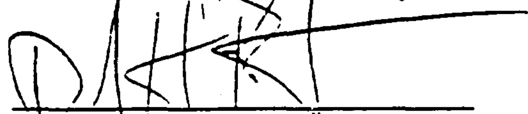
Base Configuration

Base Compiler Name: Rational Environment Version: D_12_0_0
Host Architecture: R1000 Series 200 Model 10
Operating System: Rational Environment Version D_12_0_0

Target Architecture: R1000 Series 200 Model 10
Operating System: Rational Environment Version D_12_0_0

Implementor's Declaration

I, the undersigned, representing Rational, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. I declare that Rational is the owner of record of the Ada language compilers listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compilers(s) listed in this declaration shall be made only in the owner's corporate name.

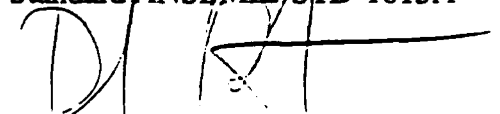


Date: 6/5/89

Rational
David H. Bernstein
Vice President, Product Development

Owners Declaration

I, the undersigned, representing Rational, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A



Date: 6/5/89

Rational
David H. Bernstein
Vice President, Product Development

APPENDIX B

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 the Rational Environment, Version D_12_0_0, as described in this Appendix, are provided by Rational. 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 -2147483647 .. 2147483647;
   type LONG_INTEGER is range -9223372036854775808 .. 9223372036854775807
   type FLOAT is digits 15
        range -1.79769313486231E+308 .. 1.79769313486231E+308;
   type DURATION is delta 3.0517578125E-5
        range -4.29496729600000E+09 .. 4.29496729599997E+09;

   ...

end STANDARD;
```

Appendix F for the R1000 Target

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

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

COMPILATION

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

Unit States

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

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

- *Archived*: The image of the unit cannot be edited. Units in this state also do not have the definition capability and structure-oriented highlighting that is available to units in the

source, installed, and coded states. Units can be put in the archived state to save space.

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

Treatment of Generics

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

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

Installation

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

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

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

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

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

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

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

Incremental Operations on Installed Units

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

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

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

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

Coding

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

Coding order differs in some respects from installation order. A library unit specification must be coded before its body can be coded. Package, generic package, and task subunits are coded before their parents are coded. Subprogram and generic subprogram subunits are coded after

their parents are coded. Library unit specifications must be coded before any clients can be coded. A main program body can be coded only after every specification and body in the closure of the main program has been coded. The system may optimize these strict ordering rules when it can make use of information from previous promotions.

Incremental Operations on Coded Units

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

- In a library unit specification, new declarations that are upwardly compatible (based on Ada semantics) can be inserted. Existing declarations with no dependents can be deleted, or they can be edited and reinserted. Because the elaboration code for the declarations in a specification is associated with the corresponding body, incremental insertions or deletions in a library unit specification result in the demotion of the corresponding body to the installed state.

- In a library unit specification, pragmas can be incrementally inserted, deleted, or edited only if all declarations to which the pragma refers are simultaneously inserted, deleted, or edited within the same insertion point.

- New context clauses that are upwardly compatible (based on Ada semantics) can be inserted only if the units named in the context clause are coded. Existing context clauses with no dependents can be deleted, or they can be edited and then reinserted. Incremental insertion or deletion of context clauses results in the demotion of any dependent main programs.

- Insertion, deletion, and editing of comments are allowed in all coded units.

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

THE PREDEFINED LANGUAGE ENVIRONMENT

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

Package Standard

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

```
package Standard is

    type Boolean is (False, True);
    type Integer is range -2147483647 .. 2147483647;
    type Float is
       digits 15
          range -1.79769313486231E+308 .. 1.79769313486231E+308
    type Long_Integer is range -9223372036854775808 .. 9223372036854775807;
```

```
    subtype Natural is Integer range 0 .. 2147483647;
    subtype Positive is Integer range 1 .. 2147483647;
    type Duration is
       delta 3.0517578125E-5
          range -4.29496729600000E+09 .. 4.29496729599997E+09
    type String is array (Positive range <>) of Character;

    package Ascii is ...
    end Ascii;

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

    type Character is ...;
    for Character'Size use 8;

end Standard;
```

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

Package System

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

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

```
package System is

    type Name    is (R1000);

    System_Name  : constant Name := R1000;

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

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


    -- System-Dependent Named Numbers

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

    Max_Digits   : constant := 15;
```

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

    subtype Priority is Integer range 0 .. 5;


    type Byte is new Natural range 0 .. 255;

    type Byte_String is array (Natural range <>) of Byte;
    -- Basic units of transmission/reception to/from IO devices

    -- The following exceptions are raised by Unchecked_Conversion or
    -- Unchecked_Conversions

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

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

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

Unchecked_Deallocation Procedure

The Unchecked_Deallocation procedure may be instantiated for any access type. Its purpose is to deallocate storage for the designated objects, so that the storage can be reclaimed and reused. For storage to be reclaimed, it must be the case that either:

- the user has included pragma Enable_Deallocation in the program, or
- the effective library switch R1000_Cg.Enable_Deallocation had the value True when the program was compiled.

Its formal parameter list is:

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

The Unchecked_Deallocation procedure assigns null to X and it may then reclaim storage for the object it designates.

Enabling deallocation for an access type causes allocated objects to consume more space; typically, this is 48 bits per object. Some additional space in the collection is used to maintain a free list. Thus, enabling deallocation also has the effect of decreasing the maximum number of

objects that can be allocated in a collection.

There are certain types (such as access types whose designated type is or contains a task) for which the R1000 architecture prevents reclamation of storage; reclaiming storage for such types could jeopardize system integrity. In these situations, valid calls to Unchecked_Deallocation will succeed and the argument will be set to null, but the storage will not be reclaimed; no exception is raised.

The rules for determining whether it is safe to reclaim storage are complex. The user can determine whether storage reclamation is possible via an instantiation of the generic function Allows_Deallocation.

For additional information, see the reference entries in the *Rational Environment Reference Manual*, PT, function Allows_Deallocation and procedure Unchecked_Deallocation.

Unchecked_Conversion Function

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

Its formal parameter list is:

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

The Source type is the type of the source object bit pattern that is to be converted to the Target type. When converting from array or discriminated record types, the architecture includes bounds information in the result.

Unchecked conversion is not permitted between arbitrary types. In particular, unchecked conversion is not permitted if:

- either type is or contains an access type
- either type is or contains a task type
- either type contains a discriminant-dependent array field

Instantiations of the Unchecked_Conversion function that are not permitted are not rejected at compile time. If the conversion is prohibited, then one of the exceptions System.Capability_Error or System.Type_Error is raised at run time, at the point of the call to Unchecked_Conversion.

A faster, package version of the Unchecked_Conversion function is the package Unchecked_Conversions. For additional information and examples, see the reference entries in the *Rational Environment Reference Manual*, PT, for the Unchecked_Conversion function and package Unchecked_Conversions.

Package Machine_Code

Package Machine_Code is not supported.

ATTRIBUTES

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

- 'Address: This attribute is supported; however, the value returned is meaningless.
- 'Storage_Size: 'Storage_Size is meaningful only when applied to access types or access subtypes, in which case it returns the number of storage units reserved for the collection associated with the base type for the access type or subtype. The value returned by 'Storage_Size is meaningless for task types or task objects.

PRAGMAS

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

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

changed.

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

  The pragma optionally takes one argument, denoting the target. If the argument does not appear, the pragma applies for all targets. For the R1000 target, the pragma is

  ```
  pragma Main (Target => R1000)
  ```

  If a pragma Main has another Target value, the pragma is ignored by the R1000 compiler. No warnings or other notifications are produced. (Other target values correspond to Rational cross-compilers.) Multiple pragma Mains, having different values for the Target parameter, may appear in a unit; this is for convenience, so that no source changes are required to recompile the unit for a different target.

  The pragma Main should be placed immediately after the end of the specification or body of a main subprogram. The subprogram must be a library unit subprogram that is without subprograms itself, and is not an instantiation. The parameters for subprograms containing this pragma must be of types defined in package Standard, package System, or another predefined package in the Environment directory structure provided by Rational.

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

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

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

- Memory_Size: This pragma has no effect.

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

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

  The pragma's syntax is:

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

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

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

- Open_Private_Part: This pragma must be placed before the first private type in a package specification. It is effective in spec views within subsystems; it indicates that a subsystem interface has an open private part. This permits inter-subsystem compilation dependencies based upon the contents of the private parts. The effect of the pragma is not inherited by nested packages: it must be included in each package that is to have an open private part. The pragma Open_Private_Part may also appear in load views within a subsystem or in library-level units within a world; it is ignored in these cases. For more information, see the "Key Concepts" section in the *Rational Environment Reference Manual*, PM.

- Optimize: This pragma currently has no effect.

- Pack: All records and arrays are stored packed in the minimum number of bits that they require. Thus, this pragma has no effect.

- Page: This pragma is used by the print spooler to cause a new page. The pragma will be the last line on the page. The next line will be printed on the next page.

- Page_Limit (X): This pragma is allowed after the end of a unit specification or body; it specifies that the page limit for the current job should be no less than X, where X is a number. The page limit is the number of virtual memory pages (containing 1024 bytes each) that can be created by the job that elaborates the unit in which the pragma appears. This pragma overrides the library switch Page_Limit, which overrides the session switch Default_Job_Page_Limit. For a more detailed description, see the reference entries in the *Rational Environment Reference Manual*, SMU, System_Utilities.Get_Page_Counts and System_Utilities.Set_Page_Limit procedures.

- Priority: Priorities can be specified only inside a task or a library main program. If multiple priorities are specified, only the first priority specified is used. The default priority is 1.

- Private_Eyes_Only: This pragma is used in conjunction with subsystems, to indicate that all items following the pragma in context clauses are required only in the closed private parts of the subsystem interface. The pragma has no effect in generic units, for which the private parts must be open. For more information, see the "Key Concepts" section in the *Rational Environment Reference Manual*, PM.

- Shared: This pragma currently has no effect.

- Storage_Unit: The only legal storage unit value for the Rational architecture is 1.

- Suppress: This pragma currently has no effect.
- System_Name: The only legal system name is R1000.

REPRESENTATION CLAUSES

The Rational Environment does not currently provide a complete implementation for representation specifications. In particular, the R1000 compiler does not support: enumeration or record representation specifications; address clauses; interrupts; 'Small length clauses for fixed point types; 'Storage_Size length clauses for tasks. The R1000 compiler does support:

- 'Size length clauses for all types
- 'Storage_Size length clauses for access types

with the characteristics and limitations discussed in following paragraphs.

To facilitate host/target development of target-dependent code containing representation clauses, the R1000 compiler accepts and ignores unsupported representation clauses if the Ignore_Unsupported_Rep_Specs library switch is set to true.

Representation of Objects

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

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

For example:

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

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

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

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

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

Size representation clauses are supported for all enumeration types. Thus, the above rules can be overridden. A specific example is the representation for the Character type in package

Standard, which takes 8 bits instead of 7 because of a size representation clause.

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

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

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

For records with discriminants, the layout can be quite complicated. The Rational architecture may introduce (unnamed) filler fields. The 'Bit_Offset attribute may be used to determine the layout of such records.

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

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

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

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

'Size Length Clauses

'Size clauses are supported for all static discrete types, and for other types as indicated in this section. The minimum acceptable specified value for the size is the default Environment storage size for the type, as described in the previous section. Thus 'Size length clauses can only increase the size of objects above the size the R1000 architecture would use in the absence of the 'Size length clause.

Effects and limitations of the 'Size length clause depend upon the type, as follows:

- Discrete Types: The 'Size limitations for type T follow the rules:

  If abs(T'First) > abs(T'Last), then 'Size >= Ceiling (Log_2(abs(T'First))) + 1

Else if T'First >= 0, then 'Size >= Ceiling (Log_2(abs(T'Last + 1)))

Else 'Size >= Ceiling (Log_2(abs(T'Last + 1))) + 1

Any value for 'Size greater than the minimum is accepted. If the length clause specifies a size less than or equal to 64 (or 63 if T'First is non-negative), then objects of the type will occupy exactly the specified number of bits. (Objects of a constrained subtype may occupy fewer bits.)

If the length clause specifies a value greater than 64, this value is regarded as an upper bound on the size of objects of type T and the length clause has no effect.

- Fixed Point Types: These objects are represented as integers scaled by 'Small. This means that the minimum 'Size for a fixed point type T is the same as that for the integer type

```
type I is range Integer (T'Small * T'First) .. Integer (T'Small * T'Last);
```

where the conversion rounds to the nearest integer. Any 'Size value greater than 64 (or 63 if T'First is non-negative) is accepted as an upper bound and has no effect. Correctly specified values less than or equal to 64 (or 63, as appropriate) will cause objects of the type to be created with exactly the specified number of bits. The 'Size length clause can be used only to increase the size of an object from the system default. Note that the additional bits *do not* increase precision or range.

The minimum value for 'Size of a fixed point type F is typically F'Mantissa (or F'Mantissa + 1 for signed fixed point types). but one additional bit may be required. In the example

```
type T is delta 1.0 range 0.0 .. 4.0;
```

T'Mantissa is 2; note that 4.0 is not a model number here. We do choose to represent the value 4.0; the equivalent integer type declaration is

```
type I is range 0 .. 4;
```

so the minimum value for T'Size is 3. There is an exception to this rule: if Integer (T'Small * T'Last) = 2 ** 63, then we choose *not* to represent T'Last; in this case, the maximum value for T'Size is T'Mantissa (i.e., 63) or T'Mantissa + 1 (i.e., 64) if T is signed.

- Access Types: To be effective, the 'Size length clause for access types should be in the range 8 .. 32. Larger values are accepted; such a value is regarded as an upper bound on the size of the objects. The default size is 24 bits. If the specified value for 'Size is 32, then the access type must be the only object declared in the package, or Storage_Error may be raised. One way to accomplish this is to use a package "skin" around the access type declaration as follows:

```
package P is
  ...
    package Skin is
        type A is access Boolean;
        for A'Size use 32;
    end Skin;
```

```
...
end P;
```

For the derived access type D specified as

```
type D is new T;
```

the minimum acceptable value for D'Size is T'Size; a value for D'Size larger than T'Size is accepted as an upper bound for the size of D.

Specifying a value for 'Size that is less than 8 will cause Storage_Error to be raised because the collection header requires 128 bits.

If 'Storage_Size is not specified, then it is set to the value (2 ** 'Size). If length clauses for both 'Size and 'Storage_Size are specified, then it must be the case that

'Size >= Ceiling (Log_2('Storage_Size))

If a length clause specifies a dynamic value for 'Storage_Size, then a length clause for 'Size is not allowed. Similarly, if a length clause for 'Size is specified and accepted, then a dynamic 'Storage_Size length clause will be rejected.

- Floating Point Types: All floating point objects require 64 bits for storage. A 'Size length clause that specifies a value greater than or equal to 64 will be accepted as an upper bound on the size of the objects.
- Task Types: All task objects required 64 bits. A 'Size length clause specifying any value greater than or equal to 64 will be accepted.
- Array and Record Types: If all the constraints on all the subcomponents are static, then a 'Size length clause is allowed for an array or record type. Any value greater than or equal to the system default size (as described in the "Representation of Objects" section) will be accepted as an upper bound on the size of the objects. The length clause has no effect: all objects of the type will have the default size.

'Storage_Size Length Clauses

Storage sizes can be specified for collections. The default collection size is 2**24 bits. The storage size for a collection can range from 2**8 to 2**32 bits. The storage size for a collection determines the number of bits required to represent access types for the collection. For example, for collections of the default 2**24 bit size, the number of bits required to store objects of the access type that is associated with this collection is 24.

A 'Storage_Size length clause may be specified for an access type, subject to some limitations:

- If both are specified, the relationship between the specified 'Size and the 'Storage_Size must be as indicated in the previous section.
- If a dynamic (nonstatic) expression is used in the 'Storage_Size length clause, then a 'Size length clause for the type is not allowed. If a 'Size length clause is specified for the type, then

a dynamic 'Storage_Size length clause is not allowed.

- The dynamic 'Storage_Size length clause must immediately follow the access type declaration. That declaration must be a one-part type declaration: it cannot be the completion of a private or incomplete type.

The storage size cannot be specified for a task in the Rational Environment. Each task in the Rational architecture has its own virtual address space, so specifying the storage size for tasks is meaningless.

Enumeration Representation Clauses

No enumeration representation clauses are currently supported.

Record Representation Clauses

No record representation clauses are currently supported.

Address Clauses

Address clauses are not supported.

Interrupts

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

CHAPTER 14 I/O

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

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

- Filenames: Filenames must conform to the syntax of Ada identifiers. They can, however, be keywords of the Ada language.

- Form parameter: Depending on the external file being written to, this parameter affects the way terminals and Ada units are read. For example, it can specify whether to have the Page pragma read with the Page_Pragma_Mapping option.

- Instantiations of package Direct_Io and package Sequential_Io with access types: Such instantiations are allowed. If files are created or opened using such instantiations, the

Use_Error exception is raised.

- Count type: The Count type for package Text_Io and package Direct_Io is defined as:

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

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

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

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

- Standard_Input and Standard_Output files: When a job is run from a Command window, these files are the interactive input/output windows provided by the Rational Editor. When a job is run from package Program, options allow the user to specify what Standard_Input and Standard_Output will be.

- Internal and external files: More than one internal file can be associated with a single external file for input only. Only one internal file can be associated with a single external file for output or inout operations.

- Sequential_Io and Direct_Io packages: Package Sequential_Io can be instantiated for unconstrained array types or for types with discriminants without default discriminant values. Package Direct_Io cannot be instantiated for unconstrained array types or for types with discriminants without default discriminant values.

- Terminators: The line terminator is denoted by the character Ascii.Lf, the page terminator is denoted by the character Ascii.Ff, and the end-of-file terminator is implicit at the end of the file. A line terminator directly followed by a page terminator is compressed to the single character Ascii.Ff. The line and page terminators preceding the file terminator are implicit and do not appear as characters in the file. For the sake of portability, programs should not depend on this representation, although it can be necessary to use this representation when importing source from another environment or exporting source from the Rational Environment.

- Treatment of control characters: Control characters, other than the terminators described above, are passed directly to and from files to application programs.

- Concurrent properties: The Chapter 14 I/O packages assume that concurrent requests for I/O resources will be synchronized by the application program making the requests, except for package Text_Io, which will synchronize requests for output.

LIMITS

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

```
with system;
package Limits is

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

   -- Scanner
  Max_Line_Length                             : constant := 254;

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

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

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

 end Limits;
```

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	24
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	(1..253 => 'A', 254 => '1')
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	(1..253 => 'A', 254 => '2')
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	(1..126 => 'A', 127 => '3', 128..254 => 'A')

TEST PARAMETERS

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	(1..126 => 'A', 127 => '4', 128..254 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..251 => '0', 252..254 => "298")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..249 => '0', 250..254 => "690.0")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1 => '"', 2..128 => 'A', 129 => '"')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1 => '"', 2..127 => 'A', 128 => '1', 129 => '"')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..234 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	1000000000
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	268435456
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	1

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	R1000
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	1.0842021724855044340074528008 6994171142578125E-19
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	5.0E09
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	3.0E14
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	5
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BAD_CHARACTERS&<>=
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	CONTAINS_WILDCARDS@
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483647

TEST PARAMETERS

Name and Meaning	Value
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-5.0E09
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-3.0E14
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	63
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	254
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	9223372036854775807
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	9223372036854775808
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 => "2:", 3..251 => '0', 252..254 => "11:")

Name and Meaning	Value
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3, => "16:", 4..250 => '0', 251..254 => "F.E:")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '" ', 2..253 => 'A', 254 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-9223372036854775808
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	64
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_INTEGER_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	R1000
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFFFFFFFFFF#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	268435456

TEST PARAMETERS

Name and Meaning	Value
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	1
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	R1000
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	64
$TICK A real literal whose value is SYSTEM.TICK.	200.0E-9

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 43 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C: This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this text that must appear at the top of the page.

b. A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. BC3009B: This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

e. CD2A62D: This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

f. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

g. CD2A81G, CD2A83G, CD2A84M..N, and CD50110 (5 tests): These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86, 96, and 58, respectively).

h. CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

i. CD2D11B: This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

j. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

k. ED7004B, ED7005C..D, and ED7006C..D (5 tests): These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

l. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

m. CD7203B and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

n. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

o. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).

p. CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

q. CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118,

132, and 136).

r. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.