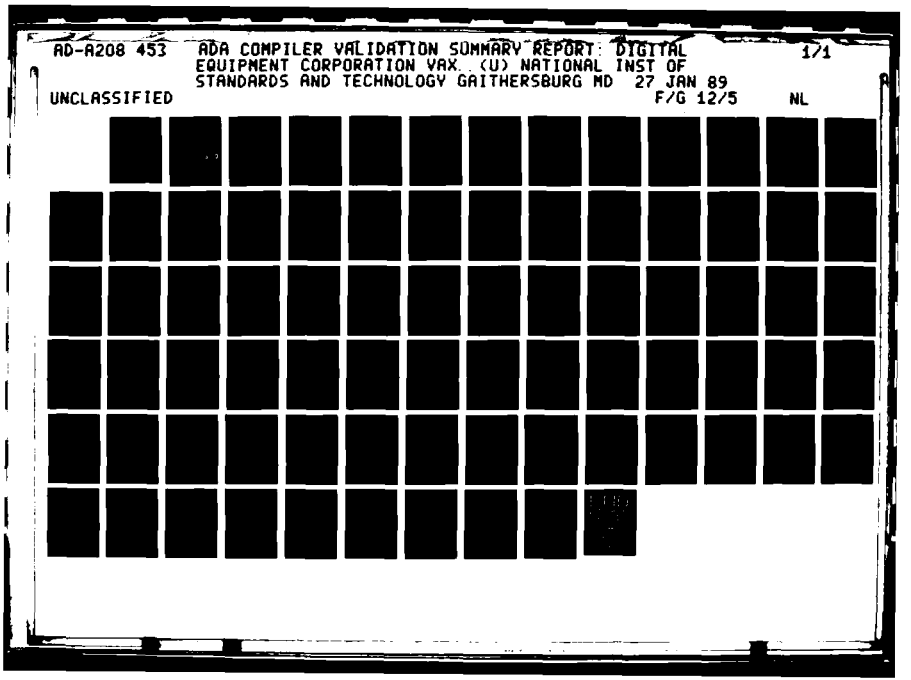
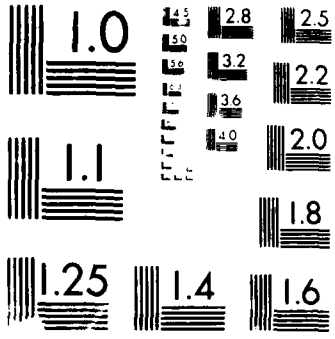

AD-A208 453
ADA COMPILER VALIDATION SUMMARY REPORT: DIGITAL EQUIPMENT CORPORATION VAX (U) NATIONAL INST OF STANDARDS AND TECHNOLOGY GAITHERSBURG MD 27 JAN 89
1/1
UNCLASSIFIED
F/G 12/5
NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

UNCLASSIFIED

Data Entered

FILE COPY

ION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

AD-A208 453

12. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

Ada Compiler Validation Summary Report: Digital Equipment Corporation, VAX Ada Version 2.0, VAX 8800 (host) to MicroVAX (target), 890127S1.10034

5. TYPE OF REPORT & PERIOD COVERED
27 Jan 1989 - 1 Dec 1990

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

National Institute of Standards and Technology
Gaithersburg, Maryland, USA

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS

National Institute of Standards and Technology
Gaithersburg, Maryland, 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*)

National Institute of Standards and Technology
Gaithersburg, Maryland, 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
MAY 30 1989
S 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*)

VAX Ada Version 2.0, Digital Equipment Corporation, National Institute of Standards and Technology, VAX 8800 VMS, Version 5.0 (host) to MicroVAX under VAXELN Toolkit, Version 3.2 in combination with VAXELN Ada, Version 2.0 (target), ACVC 1.10

89 5 30 003

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: NIST89DEC510_2_1.10

Ada Compiler Validation Summary Report:

Compiler Name: VAX Ada Version 2.0

Certificate Number: 890127S1.10034

Host: VAX 8800 VMS, Version 5.0

Target: MicroVAX under VAXELN Toolkit, Version 3.2 in combination with VAXELN Ada, Version 2.0

Testing Completed 01-27-89 Using ACVC 1.10

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Computer Systems Laboratory (NCSL)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

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

Ada Joint Program Office
Dr. John Solomond
Director
Washington D.C. 20301

AVF Control Number: NIST89DEC510_2_1.10
DRAFT PREPARED AFTER ON-SITE: 01-27-89
FINAL PREPARED: 04-27-89

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890127S1.10034
Digital Equipment Corporation
VAX Ada Version 2.0
VAX 8800 Host and MicroVAX Target

Completion of On Site Testing:
01-27-89

Prepared By:
Software Standards Validation Group
National Computer Systems
Laboratory
National Institute of Standards
and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

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

AVF Control Number: NIST89DEC510_2_1.10

Ada Compiler Validation Summary Report:

Compiler Name: VAX Ada Version 2.0

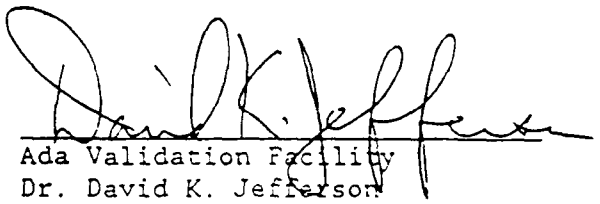
Certificate Number: 890127S1.10034

Host: VAX 8800 VMS, Version 5.0

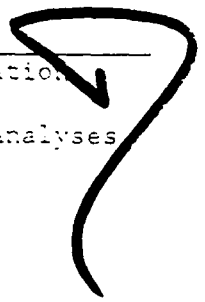
Target: MicroVAX under VAXELN Toolkit, Version 3.2 in combination with VAXELN Ada, Version 2.0

Testing Completed 01-27-89 Using ACVC 1.10

This report has been reviewed and is approved.



Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Computer Systems Laboratory (NCSL)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899



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

Ada Joint Program Office
Dr. John Solomond
Director
Washington D.C. 20301

TABLE OF CONTENTS

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

On-site testing was completed 01-27-89 at Nashua, New Hampshire.

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:

Software Standards Validation Group
National Computer Systems Laboratory
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

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	An Ada Commentary contains all information relevant to the Commentary 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 which executes the code generated by the compiler.

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 — An ACVC test found to be incorrect and not used to check test 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 errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run 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 B.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

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

Compiler: VAX Ada Version 2.0

ACVC Version: 1.10

Certificate Number: 890127S1.10034

Host Computer:

Machine: VAX 8800

Operating System: VMS, Version 5.0

Memory Size: [illegible] MBytes

Target Computer:

Machine: MicroVAX

Operating System: VAXELN Toolkit, Version 3.2 in combination with VAXELN Ada, Version 2.0

Memory Size: [illegible] MBytes

Communications Network: RC-25 REMOVABLE DISK

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 correctly processes 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 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined types SHORT_INTEGER, SHORT_SHORT_INTEGER, LONG_FLOAT, and LONG_LONG_FLOAT in the package STANDARD. (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) All 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) NUMERIC_ERROR 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) 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 (26 tests).)

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 (26 tests).)

(2) The method used for rounding to longest integer is round away from zero. (See tests C46012A..Z (26 tests).)

(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. (See test C36003A.)

(2) NUMERIC_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

(3) NUMERIC_ERROR is raised when an 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 NUMERIC_ERROR when the array type is declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array subtype is declared. (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 NUMERIC_ERROR when the array type is declared. (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, 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 supported for functions or procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

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.)

(3) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

(4) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

(5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

(6) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

(7) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

(8) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

(9) 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 and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) Mode IN_FILE is supported for the operation of CREATE for SEQUENTIAL_IO. (See test CE2102D.)

(4) Mode IN_FILE is supported for the operation of CREATE for DIRECT_IO. (See test CE2102I.)

(5) Mode IN_FILE is supported for the operation of CREATE for text files. (See test CE3102E.)

(6) DELETE operations are supported for SEQUENTIAL_IO. (See test CE2102X.)

(7) RESET operations of OUT_FILE to IN_FILE are supported but RESET operations of IN_FILE to OUT_FILE are not supported for SEQUENTIAL_IO. (See test CE2102G.)

(8) RESET and DELETE operations are supported for DIRECT_IO with the exceptions that RESET from IN_FILE to _OUT_FILE and RESET from IN_FILE to INOUT_FILE are not allowed. (See tests CE2102K and CE2102Y.)

(9) RESET and DELETE operations are supported for text files. (See tests CE2102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

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

(11) Temporary sequential files are not given names. (See test CE2108B.)

(12) Temporary direct files are not given names. (See test CE2108D.)

(13) Temporary text files are not given names. (See test [illegible].)

(14) More than one internal file can be associated with each external file for sequential files when reading only. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

(15) 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.)

(16) More than one internal file can be associated with each external file for text files when reading only. (See tests CE3111A..E (5 tests), 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 150 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. Modifications to the code, processing, or grading for 4 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	129	1132	2173	17	26	46	3523
Inapplicable	0	6	143	0	2	0	151
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	209	648	661	245	172	99	162	331	137	36	252	296	275	3523
Inapplicable	3	1	19	3	0	0	4	2	0	0	0	73	46	151
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:

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	CE2107I	CE3111C	CE3301A
CE3411B	E28005C	ED7004B	ED7005C	ED7005D	ED7006C
ED7006D					

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, 150 tests were inapplicable for the reasons indicated:

C24113W..Y (3 tests) have source lines that exceed the VAX Ada implementation limit of 255 characters.

C35702A and B86001T (2 tests) are not applicable because this implementation supports no predefined type SHORT_FLOAT.

C45531M..P and C45532M..P (8 tests) are not applicable because this implementation does not support the particular fixed point base types required by these tests.

C45231C, C45304C, C45502C, C45503C, C45504C, C45504F, C45611C, C45613C, C45614C, C45631C, C45632C, B52004D, C55B07A, B55B09C, B86001W, and CD7101F (16 tests) are not applicable because this implementation does not support a predefined type LONG_INTEGER.

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

C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. This test recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

B91001H is not applicable because this implementation does not support address clauses for task entries (AI-325). Typically, the address of the code to be executed when an interrupt occurs is stored in an interrupt vector at some particular memory location. However, the VMS operating system uses asynchronous system trap (or AST) for a software interrupt. The ASTs receive dynamically the address of the code to be executed when an interrupt occurs as a parameter to a system service routine when a service is requested.

C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

CD1009C, CD2A41A, CD2A41B, CD2A41E, CD2A42A, CD2A42B, CD2A42C, CD2A42D, CD2A42E, CD2A42F, CD2A42G, CD2A42H, CD2A42I, and CD2A42J (14 tests) are not applicable because this implementation does not support shortened mantissa and/or exponent lengths for floating point types.

CD1009I, CD2A31C, and CD2A41C (3 tests) are not applicable because a representation clause specifying SMALL for a derived fixed point type is only allowed if the resulting model numbers are representable values of the parent type.

CD2A21C, CD2A21D, CD2A21E, CD2A21H, CD2A54C, CD2A54D, and CD2A54H (7 tests) are not applicable because these tests contain size representation clauses that are illegal for certain derived fixed-point subtypes due to the particular base type selected by the implementation.

CD2A61A, CD2A61B, CD2A61C, CD2A61D, CD2A61F, CD2A61H, CD2A61I, CD2A61J, CD2A61K, CD2A61L, CD2A62A, CD2A62B, CD2A62C, CD2A64A, CD2A64B, CD2A64C, CD2A64D, CD2A65A, CD2A65B, CD2A65C, and CD2A65D (21 tests) are not applicable because this implementation does not support packing by means of a length clause for an array type.

CD2A71A, CD2A71B, CD2A71C, CD2A71D, CD2A72A, CD2A72B, CD2A72C, CD2A72D, CD2A74A, CD2A74B, CD2A74C, CD2A74D, CD2A75A, CD2A75B, CD2A75C, and CD2A75D (16 tests) are not applicable because this implementation does not support packing by means of a length clause for a record type.

CD2A84B, CD2A84C, CD2A84D, CD2A84E, CD2A84F, CD2A84G, CD2A84H, CD2A84I, CD2A84K, and CD2A84L (10 tests) are not applicable because this implementation does not support biased pointer representations.

CD2B15B is not applicable because the LRM 13.7.2(12) states that T'STORAGE_SIZE for an access type or subtype T "yields the total number of storage units reserved for the collection associated with the base type of T." The meaning of "total number of storage units reserved" is open to interpretation and it is possible for an implementation to return one of two values: the number of bytes requested (and reserved) or the number of bytes actually allocated. This compiler implements the former.

CE2102D is not applicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

CE2102E is not applicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

CE2102F is not applicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

CE2102I is not applicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

CE2102J is not applicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

CE2102N is not applicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

CE2102O is not applicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

CE2102P is not applicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

CE2102Q is not applicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

CE2102R is not applicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

CE2102S is not applicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

CE2102T is not applicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

CE2102U is not applicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

CE2102V is not applicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

CE2102W is not applicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

CE2105A is inapplicable because CREATE with IN_FILE mode is not supported by this implementation for SEQUENTIAL_IO.

CE2105B is inapplicable because CREATE with IN_FILE mode is not supported by this implementation for DIRECT_IO.

CE2107B CE2107E, CE2107G, CE2107L, CE2110B, CE2110D, CE2111H, CE3111B, CE3111D, CE3111E, CE3114B, and CE3115A (12 tests) are not applicable because this implementation does not allow more than one association for OUT_FILE or INOUT_FILE in combination with mode IN_FILE or another mode OUT_FILE or INOUT_FILE (mixed readers and writers or multiple writers) unless a non-default FORM string is specified. The proper exception is raised when multiple access is attempted.

CE2107C, CE2107D, CE2107H, CE2108B, CE2108D, and CE3112B (6 tests) are not applicable because this implementation does not support temporary names for files and the NAME function raises USE_ERROR if called with such a file under the terms of AI-00046.

CE2111G and CE2111I are not applicable because this implementation does not allow the mode of the file to be changed from IN_FILE to INOUT_FILE or OUT_FILE.

CE2115B is not applicable because this implementation does not support the instantiation of DIRECT_IO with unconstrained record types when a maximum element size is specified in the FORM parameter of the CREATE procedure. This instantiation is rejected by this compiler.

CE3102F is not applicable because this implementation supports RESET for OUT_FILE to IN_FILE.

CE3102G is not applicable because this implementation supports deletion of an external file.

CE3102I is not applicable because this implementation supports CREATE with OUT_FILE mode.

CE3102J is not applicable because this implementation supports OPEN with IN_FILE mode.

CE3102K is not applicable because this implementation supports OPEN with OUT_FILE mode.

CE3109A is inapplicable because text file CREATE with IN_FILE mode is not supported by this implementation.

EE2401D and EE2401G are not applicable because this implementation does not support the instantiation of DIRECT_IO with unconstrained record types unless a maximum element size is specified in the FORM parameter of the CREATE procedure. This instantiation is rejected by this compiler.

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 was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 4 tests.

Tests C86001A and C86001B check that library units can be named STANDARD without affecting the predefined environment. There is no difficulty using these tests independently; however, if they are attempted in the same compilation library in the order named, then the second one fails to compile because the function body named STANDARD in C86001A1 substitutes an illegal redeclaration of the package specification named STANDARD left over from the compilation of C86001A.

CB1011D uses a procedure OVERFLOW_STACK that is coded so that it needs essentially no local storage. Thus, it allows the maximum number of frames to be allocated in a given amount of memory. The VAX/VMS operating environment limits the number of frames that an exception can propagate to 65,535 frames; beyond that limit the run-time structure of the program is assumed to be corrupted and the program is aborted abnormally. execution of the program using common user allocation quotas typically results in more than 65,535 frames and leads to abnormal termination when STORAGE_ERROR is then detected. This test was included in the normal batch command test sequence, but processing was modified so that the test was executed

in a more limited address space.

C34006D checks that a derived type inherits various properties from the parent; the 'SIZE attribute is used in the tests under assumptions that are not fully supported by the Ada standard, and are subject to ARG review. Thus, an implementation is ruled to have passed these tests if the result (REPORT.RESULT) is PASSED, or if the result is FAILED and the sole cause of failure is indicated by the particular output of REPORT.FAILED below:

C34006D: "INCORRECT OBJECT'SIZE"

This implementation reports the above message and only the above message.

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 VAX Ada Version 2.0 compiler 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 VAX Ada Version 2.0 compiler 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:	VAX 8800
Host operating system:	VMS, Version 5.0
Target computer:	MicroVAX
Target operating system:	VAXELN Toolkit, Version 3.2 in combination with VAXELN Ada, Version 2.0
Compiler:	VAX Ada Version 2.0
Linker:	ACS LINK
Runtime System:	VAXELN in combination with VAXELN Ada

The host and target computers were linked via RC-25 REMOVABLE DISK.

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

TEST INFORMATION

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled and linked on the VAX 8800, then all executable images were transferred to the VAX 8800 via RC-25 REMOVABLE DISK and run. Results were printed from the host computer.

The compiler was tested using command scripts provided by Digital Equipment Corporation and reviewed by the validation team. The compiler was tested using the following option settings. See Appendix E for a complete listing of the compiler options for this implementation.

```
/NOANALYSIS_DATA
/CHECK
/COPY_SOURCE
/NODEBUG
/NODIAGNOSTICS
/ERROR_LIMIT=1
/LIBRARY=ALA$LIB
/LIST
/NOMACHINE_CODE
/NOTE_SOURCE
/OPTIMIZE
/NOSHOW
/WARNINGS=default
```

Tests were compiled, linked, and executed using a single computer. 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 [illegible]

Testing [illegible] New Hampshire and was completed on [illegible]

APPENDIX A

DECLARATION OF CONFORMANCE

Digital Equipment Corporation has submitted the following Declaration of Conformance concerning the VAX Ada Version 2.0.

Digital Equipment Corporation is currently engaged in an Ada validation and has submitted the following declaration of conformance concerning VAX Ada.

Declaration of Conformance

Compiler Implementer:
Digital Equipment Corporation

Ada Validation Facility:
National Institute of Standards and Technology

Ada Compiler Validation Capability Version: 1.10

Base Configuration:

Compiler: VAX Ada Version 2.0

Host Configuration:

VAX 8800 (under VMS, Version 5.0)

Target Configuration:

VAX 8800 (under VMS, Version 5.0)
MicroVAX II (under VAXELN Toolkit, Version 3.2
in combination with VAXELN Ada, Version 2.0)

Derived Compiler Registration:

Compiler: VAX Ada Version 2.0

Host Configurations:

MicroVAX I
MicroVAX II
MicroVAX 2000
MicroVAX 3500
MicroVAX 3600
VAXstation II
VAXstation 2000
VAXstation 3200
VAXstation 3500
VAXstation 8000
VAXserver 3500
VAXserver 3600
VAXserver 3602
VAXserver 6210
VAXserver 6220
VAX-11/730
VAX-11/750
VAX-11/780
VAX-11/785
VAX 6210
VAX 6220
VAX 6230
VAX 6240
VAX 8200
VAX 8250
VAX 8300
VAX 8350
VAX 8500
VAX 8530
VAX 8550
VAX 8600
VAX 8650
VAX 8700
VAX 8800 (base configuration)
VAX 8810
VAX 8820
VAX 8830
VAX 8840
VAX 8842
VAX 8974
VAX 8978
Raytheon Military VAX Computer Model 860
(all under VMS, Version 5.0)

Target Configuration:

Same software and configurations as Host;

And the following VAXELN configurations

MicroVAX I
MicroVAX II
MicroVAX 2000
MicroVAX 3500
MicroVAX 3600
IVAX 620
IVAX 630
rtVAX 1000
rtVAX 3200
rtVAX 3500
rtVAX 3600
rtVAX 8550
rtVAX 8700
KA620
KA800
VAX-11/725
VAX-11/730
VAX-11/750
VAX 6210
VAX 6220
VAX 6230
VAX 6240
VAX 8500
VAX 8530
VAX 8550
VAX 8700
VAX 8800
VAX 8810
VAX 8820
(all under VAXELN Toolkit, Version 3.2 in combination with VAXELN Ada, Version 2.0)

All of the processors listed above, including MicroVAX, VAXstation, and VAXserver systems, are members of the VAX family. The VAX family includes multiple hardware/software implementations of the same instruction set architecture. All processors of the VAX family together with the VMS operating system provide an identical user mode instruction set execution environment and need not be distinguished for purposes of validation. Similarly, all VAX family processors supported as VAXELN Toolkit targets provide an identical user mode instruction set execution environment.

The Military VAX Computer Model 860 is an implementation of the VAX architecture that is manufactured by Raytheon Corporation. This implementation has been tested by Digital Equipment Corporation for conformance with the VAX Architecture Standard and provides a user mode instruction set environment that is identical to other members of the VAX family.

The identical VAX Ada compiler is used on all hosts, and the compiler has no knowledge of the particular VAX model on which it is being executed. Further, the compiler generates identical code for all targets. Thus, the code generated on any VAX host can be executed without modification on any of the VAX targets listed above.

All of the configurations listed under the derived compiler registration section above are equivalent to the base configuration. That is, all applicable ACVC Version 1.10 tests could be correctly compiled and executed on any of the configurations listed.

15 February 1989

Bill Keating
Senior Group Manager
Software Development Technologies

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 VAX Ada Version 2.0 compiler, as described in this Appendix, are provided by Digital Equipment Corporation. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

...

   type INTEGER is range -2147483648..2147483647;
   type SHORT_INTEGER is range -32768..32768;
   type SHORT_SHORT_INTEGER is range -128..127;

   type FLOAT is digits 6 range -1.70141E+38..1.70141E+38;
   type LONG_FLOAT is digits 15 range
                -8.988465674312E+307..8.988465674312E+307;
   type LONG_LONG_FLOAT is digits 33 range
                -5.94865747678615882542879663314OOE+4931..
                 5.94865747678615882542879663314OOE+4931;


   type DURATION is delta 1.0E-4 range -131072.0..131071.9999;

...

end STANDARD;
```

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 ANSI/MIL-STD-1815A-1983, and to certain allowed restrictions on representation classes. The implementation-dependent characteristics are described in the following sections which discuss topics one through eight as stated in Appendix F of the Ada Language Reference manual (ANSI/MIL-STD-1815A). Two other sections, package STANDARD and file naming conventions, are also included in this appendix.

Portions of this section refer to the following attachments:

1. Attachment 1 - Implementation-Dependent Pragmas
2. Attachment 2 - VAX Ada Appendix F

(1) Implementation-Dependent Pragmas

See Attachment 2, Section F.1 and Attachment 1.

(2) Implementation-Dependent Attributes

Name	Type
P'AST_ENTRY	The value of this attribute is of type SYSTEM.AST_HANDLER.
P'BIT	The value of this attribute is of type universal_integer.
P'MACHINE_SIZE	The value of this attribute is of type universal_integer.

P'NULL_PARAMETER	The value of this attribute is of type P.
P'TYPE_CLASS	The value of this attribute is of type SYSTEM.TYPE_CLASS.

(3) Package SYSTEM

See Attachment 2, Section F.3.

(4) Representation Clause Restrictions

See Attachment 2, Section F.4.

(5) Conventions

See Attachment 2, Section F.6.

(6) Address Clauses

See Attachment 2, Section F.7.

(7) Unchecked Conversions

See Attachment 2, Section F.5.

(8) Input-Output Packages

SEQUENTIAL_IO Package

SEQUENTIAL_IO can be instantiated with any file type, including an unconstrained array type or an unconstrained record type. However, input-output for access types is erroneous.

VAX Ada provides full support for SEQUENTIAL_IO, with the following restrictions and clarifications:

1. VAX Ada supports modes IN_FILE and OUT_FILE for sequential input-output. However, VAX Ada does not allow the creation of a file of mode IN_FILE.

2. More than one internal file can be associated with the same external file. However, with default FORM strings, this is only allowed when all internal files have mode IN_FILE (multiple readers). If one or more internal files have mode OUT_FILE (mixed readers and writers or multiple writers), then sharing can only be achieved using FORM strings.

3. VAX Ada supports deletion of an external file which is associated with more than one internal file. In this case, the external file becomes immediately unavailable for any new associations, but the current associations are not affected; the external file is actually deleted after the last association has been broken.

4. VAX Ada allows resetting of shared files, but an implementation restriction does not allow the mode of a file to be changed from IN_FILE to OUT_FILE (an amplification of accessing privileges while the external file is being accessed).

DIRECT_IO Package

```
type CNT is range 0 .. 2147483647;
```

TEXT_IO Package

```
type CNT is range 0 .. 2147483647;
subtype FIELD is INTEGER range 0 .. 2147483647;
```

LOW_LEVEL_IO

Low-level input-output is not provided.

(9) Package STANDARD

```
type INTEGER is range -2147483648 .. 2147483647;
type SHORT_INTEGER is range -32768 .. 32767;
type SHORT_SHORT_INTEGER is range -128 .. 127;
-- type LONG_INTEGER is not supported

type FLOAT is digits 6;
type LONG_FLOAT is digits 15;
type LONG_LONG_FLOAT is digits 33;
-- type SHORT_FLOAT is not supported

type DURATION is delta 1.0E-4
                 range -131072.0 .. 131071.9999;
```

(10) File Names

File names follow the conventions and restrictions of the target operating system.

ATTACHMENT 1

Predefined Language Pragmas

1 This annex defines the pragmas LIST, PAGE, and OPTIMIZE, and summarizes the definitions given elsewhere of the remaining language-defined pragmas.

The VAX Ada pragmas IDENT and TITLE are also defined in this annex.

	Pragma	Meaning
	AST_ENTRY	Takes the simple name of a single entry as the single argument; at most one AST_ENTRY pragma is allowed for any given entry. This pragma must be used in combination with the AST_ENTRY attribute, and is only allowed after the entry declaration and in the same task type specification or single task as the entry to which it applies. This pragma specifies that the given entry may be used to handle a VMS asynchronous system trap (AST) resulting from a VMS system service call. The pragma does not affect normal use of the entry (see 9.12a).
2	CONTROLLED	Takes the simple name of an access type as the single argument. This pragma is only allowed immediately within the declarative part or package specification that contains the declaration of the access type; the declaration must occur before the pragma. This pragma is

		not allowed for a derived type. This pragma specifies that automatic storage reclamation must not be performed for objects designated by values of the access type, except upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration, or after leaving the main program (see 4.8).
3	ELABORATE	Takes one or more simple names denoting library units as arguments. This pragma is only allowed immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument must be the simple name of a library unit mentioned by the context clause. This pragma specifies that the corresponding library unit body must be elaborated before the given compilation unit. If the given compilation unit is a subunit, the library unit body must be elaborated before the body of the ancestor library unit of the subunit (see 10.5).
	EXPORT_EXCEPTION	Takes an internal name denoting an exception, and optionally takes an external designator (the name of a VMS Linker global symbol), a form (ADA or VMS), and a code (a static integer expression that is interpreted as a VAX condition code) as arguments. A code value must be specified when the form is VMS (the default if the form is not specified). This pragma is only allowed at the place of a declarative item, and must apply to an exception declared by an earlier declarative item of the same declarative part or package specification; it is not allowed for an exception declared with a renaming declaration. This pragma permits an Ada exception to be handled by programs written in other VAX languages (see 13.9a.3.2).

EXPORT_FUNCTION	Takes an internal name denoting a function, and optionally takes an external designator (the name of a VMS Linker global symbol), parameter types, and result type as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a function declared by an earlier declarative item of the same declarative part or package specification. In the case of a function declared as a compilation unit, the pragma is only allowed after the function declaration and before any subsequent compilation unit. This pragma is not allowed for a function declared with a renaming declaration, and is not allowed for a generic function (it may be given for a generic instantiation). This pragma permits an Ada function to be called from a program written in another VAX language (see 13.9a.1.4).
EXPORT_OBJECT	Takes an internal name denoting an object, and optionally takes an external designator (the name of a VMS Linker global symbol) and size designator (a VMS Linker global symbol whose value is the size in bytes of the exported object) as arguments. This pragma is only allowed at the place of a declarative item at the outermost level of a library package specification or body, and must apply to a variable declared by an earlier declarative item of the same package specification or body; the variable must be of a type or subtype that has a constant size at compile time. This pragma is not allowed for objects declared with a renaming declaration, and is not allowed in a generic unit. This pragma permits an Ada object to be referred to by a routine written in another VAX language (see 13.9a.2.2).

EXPORT_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a VMS Linker global symbol) and parameter types as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. This pragma is not allowed for a procedure declared with a renaming declaration, and is not allowed for a generic procedure (it may be given for a generic instantiation). This pragma permits an Ada routine to be called from a program written in another VAX language (see 13.9a.1.4).
EXPORT_VALUED_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a VMS Linker global symbol) and parameter types as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. The first (or only) parameter of the procedure must be of mode out. This pragma is not allowed for a procedure declared with a renaming declaration and is not allowed for a generic procedure (it may be given for a generic instantiation). This pragma permits an Ada procedure to behave as

a function that both returns a value and causes side effects on its parameters when it is called from a routine written in another VAX language (see 13.9a.1.4).

IDENT

Takes a string literal of 31 or fewer characters as the single argument. The pragma IDENT has the following form:

```
pragma IDENT (string_literal);
```

This pragma is allowed only in the outermost declarative part or declarative items of a compilation unit. The given string is used to identify the object module associated with the compilation unit in which the pragma IDENT occurs.

IMPORT_EXCEPTION

Takes an internal name denoting an exception, and optionally takes an external designator (the name of a VMS Linker global symbol), a form (ADA or VMS), and a code (a static integer expression that is interpreted as a VAX condition code) as arguments. A code value is allowed only when the form is VMS (the default if the form is not specified). This pragma is only allowed at the place of a declarative item, and must apply to an exception declared by an earlier declarative item of the same declarative part or package specification; it is not allowed for an exception declared with a renaming declaration. This pragma permits a non-Ada exception (most notably, a VAX condition) to be handled by an Ada program (see 13.9a.3.1).

IMPORT_FUNCTION

Takes an internal name denoting a function, and optionally takes an external designator (the name of a VMS Linker global symbol), parameter types, result type, and mechanism as arguments. The pragma INTERFACE must be used with this pragma (see 13.9). This pragma is

	only allowed at the place of a declarative item, and must apply to a function declared by an earlier declarative item of the same declarative part or package specification. In the case of a function declared as a compilation unit, the pragma is only allowed after the function declaration and before any subsequent compilation unit. This pragma is allowed for a function declared with a renaming declaration; it is not allowed for a generic function or a generic function instantiation. This pragma permits a non-Ada routine to be used as an Ada function (see 13.9a.1.1).
IMPORT_OBJECT	Takes an internal name denoting an object, and optionally takes an external designator (the name of a VMS Linker global symbol) and size (a VMS Linker global symbol whose value is the size in bytes of the imported object) as arguments. This pragma is only allowed at the place of a declarative item at the outermost level of a library package specification or body, and must apply to a variable declared by an earlier declarative item of the same package specification or body; the variable must be of a type or subtype that has a constant size at compile time. This pragma is not allowed for objects declared with a renaming declaration, and is not allowed in a generic unit. This pragma permits storage declared in a non-Ada routine to be referred to by an Ada program (see 13.9a.2.1).
IMPORT_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a VMS Linker global symbol) parameter types, and mechanism as arguments. The pragma INTERFACE must be used with this pragma (see 13.9). This

	pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. This pragma is allowed for a procedure declared with a renaming declaration; it is not allowed for a generic procedure or a generic procedure instantiation. This pragma permits a non-Ada routine to be used as an Ada procedure (see 13.9a.1.1).
IMPORT_VALUED_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a VMS Linker global symbol), parameter types, and mechanism as arguments. The pragma INTERFACE must be used with this pragma (see 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. The first (or only) parameter of the procedure must be of mode out. This pragma is allowed for a procedure declared with a renaming declaration; it is not allowed for a generic procedure. This pragma permits a non-Ada routine that returns a value and causes side effects on its parameters to be used as an Ada procedure (see 13.9a.1.1).
INLINE	Takes one or more names as arguments; each name is either the name of a

subprogram or the name of a generic subprogram. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. This pragma specifies that the subprogram bodies should be expanded inline at each call whenever possible; in the case of a generic subprogram, the pragma applies to calls of its instantiations (see 6.3.2).

INLINE_GENERIC

Takes one or more names as arguments; each name is either the name of a generic declaration or the name of an instance of a generic declaration. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. Each argument must be the simple name of a generic subprogram or package, or a (nongeneric) subprogram or package, declared by an earlier declarative item of the same declarative part or package specification. This pragma specifies that inline expansion of the generic template is desired for each instantiation of the named generic declarations or of the particular named instances; the pragma does not apply to calls of instances of generic subprograms (see 12.1a).

5 INTERFACE

Takes a language name and a subprogram name as arguments. This pragma is allowed at the place of a declarative item, and must apply in this case to a subprogram declared by an earlier declarative item of the same declarative part or package specification. This pragma is also allowed for a library unit; in this case the pragma must appear after the subprogram declaration, and

before any subsequent compilation unit. This pragma specifies the other language (and thereby the calling conventions) and informs the compiler that an object module will be supplied for the corresponding subprogram (see 13.9).

In VAX Ada, the pragma INTERFACE is required in combination with the pragmas IMPORT_FUNCTION, IMPORT_PROCEDURE, and IMPORT_VALUED_PROCEDURE when any of those pragmas are used (see 13.9a.1).

6 LIST

Takes one of the identifiers ON or OFF as the single argument. This pragma is allowed anywhere a pragma is allowed. It specifies that listing of the compilation is to be continued or suspended until a LIST pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing.

LONG_FLOAT

Takes either D_FLOAT or G_FLOAT as the single argument. The default is G_FLOAT. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. It specifies the choice of representation to be used for the predefined type LONG_FLOAT in the package STANDARD, and for floating point type declarations with digits specified in the range 7..15 (see 3.5.7a).

MAIN_STORAGE

Takes one or two nonnegative static simple expressions of some integer type as arguments. This pragma is only allowed in the outermost declarative part of a library subprogram; at most one such pragma is allowed in a library subprogram. It has an effect only when the subprogram to which it applies is used as a main program. This pragma causes a fixed-size stack to be created

		for a main task (the task associated with a main program), and determines the number of storage units (bytes) to be allocated for the stack working storage area or guard pages or both. The value specified for either or both the working storage area and guard pages is rounded up to an integral number of pages. A value of zero for the working storage area results in the use of a default size; a value of zero for the guard pages results in no guard storage. A negative value for either working storage or guard pages causes the pragma to be ignored (see 13.2b).
7	MEMORY_SIZE	Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number MEMORY_SIZE (see 13.7).
8	OPTIMIZE	Takes one of the identifiers TIME or SPACE as the single argument. This pragma is only allowed within a declarative part and it applies to the block or body enclosing the declarative part. It specifies whether time or space is the primary optimization criterion. In VAX Ada, this pragma is only allowed immediately within a declarative part of a body declaration.
9	PACK	Takes the simple name of a record or array type as the single argument. The allowed positions for this pragma, and the restrictions on the named type, are governed by the same rules as for a representation clause. The pragma specifies that storage minimization should be the main criterion when

		selecting the representation of the given type (see 13.1).
10	PAGE	This pragma has no argument, and is allowed anywhere a pragma is allowed. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).
11	PRIORITY	Takes a static expression of the predefined integer subtype PRIORITY as the single argument. This pragma is only allowed within the specification of a task unit or immediately within the outermost declarative part of a main program. It specifies the priority of the task (or tasks of the task type) or the priority of the main program (see 9.8).
	PSECT_OBJECT	Takes an internal name denoting an object, and optionally takes an external designator (the name of a program section) and a size (a VMS Linker global symbol whose value is interpreted as the size in bytes of the exported/imported object) as arguments. This pragma is only allowed at the place of a declarative item at the outermost level of a library package specification or body, and must apply to a variable declared by an earlier declarative item of the same package specification or body; the variable must be of a type or subtype that has a constant size at compile time. This pragma is not allowed for an object declared with a renaming declaration, and is not allowed in a generic unit. This pragma enables the shared use of objects that are stored in overlaid program sections (see 13.9a.2.3).
12	SHARED	Takes the simple name of a variable as the single argument. This pragma is allowed only for a variable declared by an object declaration and whose type

		is a scalar or access type; the variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. This pragma specifies that every read or update of the variable is a synchronization point for that variable. An implementation must restrict the objects for which this pragma is allowed to objects for which each of direct reading and direct updating is implemented as an indivisible operation (see 9.11).
	SHARE_GENERIC	Takes one or more names as arguments; each name is either the name of a generic declaration or the name of an instance of a generic declaration. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. Each argument must be the simple name of a generic subprogram or package, or a (nongeneric) subprogram or package, declared by an earlier declarative item of the same declarative part or package specification. This pragma specifies that generic code sharing is desired for each instantiation of the named generic declarations or of the particular named instances (see 12.1b).
13	STORAGE_UNIT	Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number STORAGE_UNIT (see 13.7). In VAX Ada, the only argument allowed for this pragma is 8 (bits).

14 SUPPRESS

Takes as arguments the identifier of a check and optionally also the name of either an object, a type or subtype, a subprogram, a task unit, or a generic unit. This pragma is only allowed either immediately within a declarative part or immediately within a package specification. In the latter case, the only allowed form is with a name that denotes an entity (or several overloaded subprograms) declared immediately within the package specification. The permission to omit the given check extends from the place of the pragma to the end of the declarative region associated with the innermost enclosing block statement or program unit. For a pragma given in a package specification, the permission extends to the end of the scope of the named entity.

If the pragma includes a name, the permission to omit the given check is further restricted: it is given only for operations on the named object or on all objects of the base type of a named type or subtype; for calls of a named subprogram; for activations of tasks of the named task type; or for instantiations of the given generic unit (see 11.7).

SUPPRESS_ALL

This pragma has no argument and is only allowed following a compilation unit. This pragma specifies that all run-time checks in the unit are suppressed (see 11.7).

15 SYSTEM_NAME

Takes an enumeration literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the enumeration literal with the specified identifier for the definition of the constant SYSTEM_NAME. This

	pragma is only allowed if the specified identifier corresponds to one of the literals of the type NAME declared in the package SYSTEM (see 13.7).
TASK_STORAGE	Takes the simple name of a task type and a static expression of some integer type as arguments. This pragma is allowed anywhere that a task storage specification is allowed; that is, the declaration of the task type to which the pragma applies and the pragma must both occur (in this order) immediately within the same declarative part, package specification, or task specification. The effect of this pragma is to use the value of the expression as the number of storage units (bytes) to be allocated as guard storage. The value is rounded up to an integral number of pages: a value of zero results in no guard storage; a negative value causes the pragma to be ignored (see 13.2a).
TIME_SLICE	Takes a static expression of the predefined fixed point type DURATION (in package STANDARD) as the single argument. This pragma is only allowed in the outermost declarative part of a library subprogram, and at most one such pragma is allowed in a library subprogram. It has an effect only when the subprogram to which it applies is used as a main program. This pragma specifies the nominal amount of elapsed time permitted for the execution of a task when other tasks of the same priority are also eligible for execution. A positive, nonzero value of the static expression enables round-robin scheduling for all tasks in the subprogram; a negative or zero value disables it (see 9.8a).

TITLE

Takes a title or a subtitle string, or both, in either order, as arguments. The pragma TITLE has the following form:

```
pragma TITLE (titling-option
   [,titling-option]);

titling-option :=
      [TITLE =>] string_literal
   | [SUBTITLE =>] string_literal
```

This pragma is allowed anywhere a pragma is allowed; the given strings supersede the default title and/or subtitle portions of a compilation listing.

VOLATILE

Takes the simple name of a variable as the single argument. This pragma is only allowed for a variable declared by an object declaration. The variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. The pragma must appear before any occurrence of the name of the variable other than in an address clause or in one of the VAX Ada pragmas IMPORT_OBJECT, EXPORT_OBJECT, or PSECT_OBJECT. The variable cannot be declared by a renaming declaration. The pragma VOLATILE specifies that the variable may be modified asynchronously. This pragma instructs the compiler to obtain the value of a variable from memory each time it is used (see 9.11).

ATTACHMENT 2

Implementation-Dependent Characteristics

NOTE

This appendix is not part of the standard definition of the Ada programming language.

This appendix summarizes the implementation-dependent characteristics of VAX Ada by presenting the following:

- Lists of the VAX Ada pragmas and attributes.
- The specification of the package SYSTEM.
- The restrictions on representation clauses and unchecked type conversions.
- The conventions for names denoting implementation-dependent components in record representation clauses.
- The interpretation of expressions in address clauses.
- The implementation-dependent characteristics of the input-output packages.
- Other implementation-dependent characteristics.

F.1 Implementation-Dependent Pragmas

VAX Ada provides the following pragmas, which are defined elsewhere in the text. In addition, VAX Ada restricts the predefined language pragmas INLINE and INTERFACE, and provides alternatives to the pragmas SHARED and SUPPRESS (VOLATILE and SUPPRESS_ALL). See Annex B for a descriptive pragma summary.

- AST_ENTRY (see 9.12a).
- EXPORT_EXCEPTION (see 13.9a.3.2).

- EXPORT_FUNCTION (see 13.9a.1.4).
- EXPORT_OBJECT (see 13.9a.2.2).
- EXPORT_PROCEDURE (see 13.9a.1.4).
- EXPORT_VALUED_PROCEDURE (see 13.9a.1.4).
- IDENT (see Annex B).
- IMPORT_EXCEPTION (see 13.9a.3.1).
- IMPORT_FUNCTION (see 13.9a.1.1).
- IMPORT_OBJECT (see 13.9a.2.1).
- IMPORT_PROCEDURE (see 13.9a.1.1).
- IMPORT_VALUED_PROCEDURE (see 13.9a.1.1).
- INLINE_GENERIC (see 12.1a).
- LONG_FLOAT (see 3.5.7a).
- MAIN_STORAGE (see 13.2b).
- PSECT_OBJECT (see 13.9a.2.3).
- SUPPRESS_ALL (see 11.7).
- TASK_STORAGE (see 13.2a).
- TIME_SLICE (see 9.8a).
- TITLE (see Annex B).
- VOLATILE (see 9.11).

F.2 Implementation-Dependent Attributes

VAX Ada provides the following attributes, which are defined elsewhere in the text. See Annex A for a descriptive attribute summary.

- AST_ENTRY (see 9.12a).
- BIT (see 13.7.2).
- MACHINE_SIZE (see 13.7.2).
- NULL_PARAMETER (see 13.9a.1.3).
- TYPE_CLASS (see 13.7a.2).

F.3 Specification of the Package System

```
package SYSTEM is

    type NAME is (VAX_VMS, VAXELN);

    SYSTEM_NAME    : constant NAME := VAX_VMS;
    STORAGE_UNIT   : constant := 8;
    MEMORY_SIZE    : constant := 2**31-1;
    MAX_INT        : constant := 2**31-1;
    MIN_INT        : constant := -(2**31);
    MAX_DIGITS     : constant := 33;
    MAX_MANTISSA   : constant := 31;
    FINE_DELTA     : constant := 2.0**(-31);
    TICK           : constant := 10.0**(-2);

    subtype PRIORITY is INTEGER range 0 .. 16;

--  Address type
--
    type ADDRESS is private;

    ADDRESS_ZERO : constant ADDRESS;

    function "+" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;
    function "+" (LEFT : INTEGER; RIGHT : ADDRESS) return ADDRESS;
    function "-" (LEFT : ADDRESS; RIGHT : ADDRESS) return INTEGER;
    function "-" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;

--  function "="  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
--  function "/=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function "<"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function "<=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function ">"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function ">=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;

--  Note that because ADDRESS is a private type
--  the functions "=" and "/=" are already available and
--  do not have to be explicitly defined

    generic
        type TARGET is private;
    function FETCH_FROM_ADDRESS (A : ADDRESS) return TARGET;

    generic
        type TARGET is private;
    procedure ASSIGN_TO_ADDRESS (A : ADDRESS; T : TARGET);

--  VAX Ada floating point type declarations for the VAX
--  hardware floating point data types

    type F_FLOAT is implementation_defined;
    type D_FLOAT is implementation_defined;
    type G_FLOAT is implementation_defined;
    type H_FLOAT is implementation_defined;
```

```
    type TYPE_CLASS is (TYPE_CLASS_ENUMERATION,
                        TYPE_CLASS_INTEGER,
                        TYPE_CLASS_FIXED_POINT,
                        TYPE_CLASS_FLOATING_POINT,
                        TYPE_CLASS_ARRAY,
                        TYPE_CLASS_RECORD,
                        TYPE_CLASS_ACCESS,
                        TYPE_CLASS_TASK,
                        TYPE_CLASS_ADDRESS);

--  AST handler type

    type AST_HANDLER is limited private;

    NO_AST_HANDLER : constant AST_HANDLER;

--  Non-Ada exception

    NON_ADA_ERROR : exception;

--  VAX hardware-oriented types and functions

    type    BIT_ARRAY is array (INTEGER range <>) of BOOLEAN;
    pragma  PACK(BIT_ARRAY);

    subtype BIT_ARRAY_8  is BIT_ARRAY (0 .. 7);
    subtype BIT_ARRAY_16 is BIT_ARRAY (0 .. 15);
    subtype BIT_ARRAY_32 is BIT_ARRAY (0 .. 31);
    subtype BIT_ARRAY_64 is BIT_ARRAY (0 .. 63);

    type UNSIGNED_BYTE   is range 0 .. 255;
    for  UNSIGNED_BYTE'SIZE  use 8;

    function "not" (LEFT        : UNSIGNED_BYTE) return UNSIGNED_BYTE;
    function "and" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
    function "or"  (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
    function "xor" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;

    function TO_UNSIGNED_BYTE (X : BIT_ARRAY_8) return UNSIGNED_BYTE;
    function TO_BIT_ARRAY_8 (X : UNSIGNED_BYTE) return BIT_ARRAY_8;

    type UNSIGNED_BYTE_ARRAY is array (INTEGER range <>) of UNSIGNED_BYTE;

    type UNSIGNED_WORD   is range 0    65535;
    for  UNSIGNED_WORD'SIZE  use 16;

    function "not" (LEFT        : UNSIGNED_WORD) return UNSIGNED_WORD;
    function "and" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
    function "or"  (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
    function "xor" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;

    function TO_UNSIGNED_WORD (X   BIT_ARRAY_16) return UNSIGNED_WORD;
    function TO_BIT_ARRAY_16 (X : UNSIGNED_WORD) return BIT_ARRAY_16;

    type UNSIGNED_WORD_ARRAY is array (INTEGER range <>) of UNSIGNED_WORD;

    type UNSIGNED_LONGWORD is range MIN_INT    MAX_INT;
    for UNSIGNED_LONGWORD'SIZE use 32;
```

```
    function "not" (LEFT        : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
    function "and" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
    function "or"  (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
    function "xor" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;

    function TO_UNSIGNED_LONGWORD (X : BIT_ARRAY_32)
       return UNSIGNED_LONGWORD;
    function TO_BIT_ARRAY_32 (X : UNSIGNED_LONGWORD) return BIT_ARRAY_32;

    type UNSIGNED_LONGWORD_ARRAY is
       array (INTEGER range <>) of UNSIGNED_LONGWORD;

    type UNSIGNED_QUADWORD is record
        L0 : UNSIGNED_LONGWORD;
        L1 : UNSIGNED_LONGWORD;
        end record;
    for UNSIGNED_QUADWORD'SIZE use 64;

    function "not" (LEFT        : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
    function "and" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
    function "or"  (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
    function "xor" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;

    function TO_UNSIGNED_QUADWORD (X : BIT_ARRAY_64)
       return UNSIGNED_QUADWORD;
    function TO_BIT_ARRAY_64 (X : UNSIGNED_QUADWORD) return BIT_ARRAY_64;

    type UNSIGNED_QUADWORD_ARRAY is
       array (INTEGER range <>) of UNSIGNED_QUADWORD;

    function TO_ADDRESS  (X : INTEGER)             return ADDRESS;
    function TO_ADDRESS  (X : UNSIGNED_LONGWORD)   return ADDRESS;
    function TO_ADDRESS  (X : universal_integer)   return ADDRESS;

    function TO_INTEGER          (X : ADDRESS)     return INTEGER;
    function TO_UNSIGNED_LONGWORD (X : ADDRESS)    return UNSIGNED_LONGWORD;

    function TO_UNSIGNED_LONGWORD (X : AST_HANDLER) return UNSIGNED_LONGWORD;

--  Conventional names for static subtypes of type UNSIGNED_LONGWORD

    subtype UNSIGNED_1  is UNSIGNED_LONGWORD range 0    2** 1-1;
    subtype UNSIGNED_2  is UNSIGNED_LONGWORD range 0    2** 2-1;
    subtype UNSIGNED_3  is UNSIGNED_LONGWORD range 0    2** 3-1;
    subtype UNSIGNED_4  is UNSIGNED_LONGWORD range 0    2** 4-1;
    subtype UNSIGNED_5  is UNSIGNED_LONGWORD range 0    2** 5-1;
    subtype UNSIGNED_6  is UNSIGNED_LONGWORD range 0    2** 6-1;
    subtype UNSIGNED_7  is UNSIGNED_LONGWORD range 0    2** 7-1;
    subtype UNSIGNED_8  is UNSIGNED_LONGWORD range 0    2** 8-1;
    subtype UNSIGNED_9  is UNSIGNED_LONGWORD range 0    2** 9-1;
    subtype UNSIGNED_10 is UNSIGNED_LONGWORD range 0    2**10-1;
    subtype UNSIGNED_11 is UNSIGNED_LONGWORD range 0    2**11-1;
    subtype UNSIGNED_12 is UNSIGNED_LONGWORD range 0    2**12-1;
    subtype UNSIGNED_13 is UNSIGNED_LONGWORD range 0    2**13-1;
    subtype UNSIGNED_14 is UNSIGNED_LONGWORD range 0    2**14-1;
    subtype UNSIGNED_15 is UNSIGNED_LONGWORD range 0    2**15-1;
```

```
    subtype UNSIGNED_16 is UNSIGNED_LONGWORD range 0 .. 2**16-1;
    subtype UNSIGNED_17 is UNSIGNED_LONGWORD range 0 .. 2**17-1;
    subtype UNSIGNED_18 is UNSIGNED_LONGWORD range 0 .. 2**18-1;
    subtype UNSIGNED_19 is UNSIGNED_LONGWORD range 0 .. 2**19-1;
    subtype UNSIGNED_20 is UNSIGNED_LONGWORD range 0 .. 2**20-1;
    subtype UNSIGNED_21 is UNSIGNED_LONGWORD range 0 .. 2**21-1;
    subtype UNSIGNED_22 is UNSIGNED_LONGWORD range 0 .. 2**22-1;
    subtype UNSIGNED_23 is UNSIGNED_LONGWORD range 0 .. 2**23-1;
    subtype UNSIGNED_24 is UNSIGNED_LONGWORD range 0 .. 2**24-1;
    subtype UNSIGNED_25 is UNSIGNED_LONGWORD range 0 .. 2**25-1;
    subtype UNSIGNED_26 is UNSIGNED_LONGWORD range 0 .. 2**26-1;
    subtype UNSIGNED_27 is UNSIGNED_LONGWORD range 0 .. 2**27-1;
    subtype UNSIGNED_28 is UNSIGNED_LONGWORD range 0 .. 2**28-1;
    subtype UNSIGNED_29 is UNSIGNED_LONGWORD range 0 .. 2**29-1;
    subtype UNSIGNED_30 is UNSIGNED_LONGWORD range 0 .. 2**30-1;
    subtype UNSIGNED_31 is UNSIGNED_LONGWORD range 0 .. 2**31-1;

--  Function for obtaining global symbol values

    function IMPORT_VALUE (SYMBOL : STRING) return UNSIGNED_LONGWORD;

--  VAX device and process register operations

    function  READ_REGISTER (SOURCE : UNSIGNED_BYTE)     return UNSIGNED_BYTE;
    function  READ_REGISTER (SOURCE : UNSIGNED_WORD)     return UNSIGNED_WORD;
    function  READ_REGISTER (SOURCE : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;

    procedure WRITE_REGISTER(SOURCE : UNSIGNED_BYTE;
                             TARGET : out UNSIGNED_BYTE);
    procedure WRITE_REGISTER(SOURCE : UNSIGNED_WORD;
                             TARGET : out UNSIGNED_WORD);
    procedure WRITE_REGISTER(SOURCE : UNSIGNED_LONGWORD;
                             TARGET : out UNSIGNED_LONGWORD);

    function  MFPR (REG_NUMBER : INTEGER) return UNSIGNED_LONGWORD;
    procedure MTPR (REG_NUMBER : INTEGER;
                    SOURCE     : UNSIGNED_LONGWORD);

--  VAX interlocked-instruction procedures

    procedure CLEAR_INTERLOCKED (BIT       : in out BOOLEAN,
                                 OLD_VALUE : out BOOLEAN);
    procedure SET_INTERLOCKED   (BIT       : in out BOOLEAN,
                                 OLD_VALUE : out BOOLEAN);

    type ALIGNED_WORD is
       record
          VALUE : SHORT_INTEGER;
       end record;
    for ALIGNED_WORD use
       record
          at mod 2;
       end record;

    procedure ADD_INTERLOCKED (ADDEND : in     SHORT_INTEGER,
                               AUGEND : in out ALIGNED_WORD,
                               SIGN   : out    INTEGER);
```

```
    type INSQ_STATUS is (OK_NOT_FIRST, FAIL_NO_LOCK, OK_FIRST);
    type REMQ_STATUS is (OK_NOT_EMPTY, FAIL_NO_LOCK,
                         OK_EMPTY, FAIL_WAS_EMPTY);

    procedure INSQHI (ITEM   : in  ADDRESS;
                      HEADER : in  ADDRESS;
                      STATUS : out INSQ_STATUS);

    procedure REMQHI (HEADER : in  ADDRESS;
                      ITEM   : out ADDRESS;
                      STATUS : out REMQ_STATUS);

    procedure INSQTI (ITEM   : in  ADDRESS;
                      HEADER : in  ADDRESS;
                      STATUS : out INSQ_STATUS);

    procedure REMQTI (HEADER : in  ADDRESS;
                      ITEM   : out ADDRESS;
                      STATUS : out REMQ_STATUS);

private

    -- Not shown

end SYSTEM;
```

F.4 Restrictions on Representation Clauses

The representation clauses allowed in VAX Ada are length, enumeration, record representation, and address clauses.

In VAX Ada, a representation clause for a generic formal type or a type that depends on a generic formal type is not allowed. In addition, a representation clause for a composite type that has a component or subcomponent of a generic formal type or a type derived from a generic formal type is not allowed.

F.5 Restrictions on Unchecked Type Conversions

VAX Ada supports the generic function UNCHECKED_CONVERSION with the following restrictions on the class of types involved:

- The actual subtype corresponding to the formal type TARGET must not be an unconstrained array type.
- The actual subtype corresponding to the formal type TARGET must not be an unconstrained type with discriminants.

Further, when the target type is a type with discriminants, the value resulting from a call of the conversion function resulting from an instantiation of UNCHECKED_CONVERSION is checked to ensure that the discriminants satisfy the constraints of the actual subtype.

If the size of the source value is greater than the size of the target subtype, then the high order bits of the value are ignored (truncated); if the size of the source value is less than the size of the target subtype, then the value is extended with zero bits to form the result value.

F.6 Conventions for Implementation-Generated Names Denoting Implementation-Dependent Components in Record Representation Clauses

VAX Ada does not allocate implementation-dependent components in records.

F.7 Interpretation of Expressions Appearing in Address Clauses

Expressions appearing in address clauses must be of the type ADDRESS defined in the package SYSTEM (see 13.7a.1 and F.3). In VAX Ada, values of type SYSTEM.ADDRESS are interpreted as virtual addresses in the VAX address space.

VAX Ada allows address clauses for variables (see 13.5).

VAX Ada does not support interrupts as defined in section 13.5.1. VAX Ada does provide the pragma AST_ENTRY and the AST_ENTRY attribute as alternative mechanisms for handling asynchronous interrupts from the VMS operating system (see 9.12a).

F.8 Implementation-Dependent Characteristics of Input-Output Packages

The VAX Ada predefined packages and their operations are implemented using VAX Record Management Services (RMS) file organizations and facilities. To give users the maximum benefit of the underlying VAX RMS input-output facilities, VAX Ada provides packages in addition to the packages SEQUENTIAL_IO, DIRECT_IO, TEXT_IO, and IO_EXCEPTIONS.

and VAX Ada accepts VAX RMS File Definition Language (FDL) statements in form strings. The following sections summarize the implementation-dependent characteristics of the VAX Ada input-output packages. The *VAX Ada Run-Time Reference Manual* discusses these characteristics in more detail.

F.8.1 Additional VAX Ada Input-Output Packages

In addition to the language-defined input-output packages (SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO), VAX Ada provides the following input-output packages:

- RELATIVE_IO (see 14.2a.3).
- INDEXED_IO (see 14.2a.5).
- SEQUENTIAL_MIXED_IO (see 14.2b.4).
- DIRECT_MIXED_IO (see 14.2b.6).
- RELATIVE_MIXED_IO (see 14.2b.8).
- INDEXED_MIXED_IO (see 14.2b.10).

VAX Ada does not provide the package LOW_LEVEL_IO.

F.8.2 Auxiliary Input-Output Exceptions

VAX Ada defines the exceptions needed by the packages RELATIVE_IO, INDEXED_IO, RELATIVE_MIXED_IO, and INDEXED_MIXED_IO in the package AUX_IO_EXCEPTIONS (see 14.5a).

F.8.3 Interpretation of the FORM Parameter

The value of the FORM parameter for the OPEN and CREATE procedures of each input-output package may be a string whose value is interpreted as a sequence of statements of the VAX Record Management Services (RMS) File Definition Language (FDL), or it may be a string whose value is interpreted as the name of an external file containing FDL statements.

The use of the FORM parameter is described for each input-output package in chapter 14. For information on the default FORM parameters for each VAX Ada input-output package and for information on using the FORM parameter to specify external file attributes, see the *VAX Ada Run-Time Reference Manual*. For information on FDL, see the *Guide to VMS File Applications* and the *VMS File Definition Language Facility Manual*.

F.8.4 Implementation-Dependent Input-Output Error Conditions

As specified in section 14.4, VAX Ada raises the following language-defined exceptions for error conditions that occur during input-output operations: STATUS_ERROR, MODE_ERROR, NAME_ERROR, USE_ERROR, END_ERROR, DATA_ERROR and LAYOUT_ERROR. In addition, VAX Ada raises the following exceptions for relative and indexed input-output operations: LOCK_ERROR, EXISTENCE_ERROR, and KEY_ERROR. VAX Ada does not raise the language-defined exception DEVICE_ERROR; device-related error conditions cause the exception USE_ERROR to be raised.

The exception USE_ERROR is raised under the following conditions:

- If the capacity of the external file has been exceeded.
- In all CREATE operations if the mode specified is IN_FILE.
- In all CREATE operations if the file attributes specified by the FORM parameter are not supported by the package.
- In all CREATE, OPEN, DELETE, and RESET operations if, for the specified mode, the environment does not support the operation for an external file.
- In all NAME operations if the file has no name.
- In the WRITE operations on relative or indexed files if the element in the position indicated has already been written.
- In the DELETE_ELEMENT operations on relative and indexed files if the current element is undefined at the start of the operation.
- In the UPDATE operations on indexed files if the current element is undefined or if the specified key violates the external file attributes.
- In the SET_LINE_LENGTH and SET_PAGE_LENGTH operations on text files if the lengths specified are inappropriate for the external file.
- In text files if an operation is attempted that is not possible for reasons that depend on characteristics of the external file.

The exception NAME_ERROR is raised as specified in section 14.4: by a call of a CREATE or OPEN procedure if the string given for the NAME parameter does not allow the identification of an external file. In VAX Ada, the value of a NAME parameter can be a string that denotes a VMS file specification or a VMS logical name (in either case, the string names an external file). For a CREATE procedure, the value of a NAME parameter can also be a null string, in which case it names a temporary external file that is deleted when the main program exits. The *VAX Ada Run-Time Reference Manual* explains the naming of external files in more detail.

F.9 Other Implementation Characteristics

Implementation characteristics relating to the definition of a main program, various numeric ranges, and implementation limits are summarized in the following sections.

F.9.1 Definition of a Main Program

A main program can be a library unit subprogram under the following conditions:

- If it is a procedure with no formal parameters. In this case, the status returned to the VMS environment upon normal completion of the procedure is the value 1.
- If it is a function with no formal parameters whose returned value is of a discrete type. In this case, the status returned to the VMS environment upon normal completion of the function is the function value.
- If it is a procedure declared with the pragma EXPORT_VALUED_PROCEDURE, and it has one formal out parameter that is of a discrete type. In this case, the status returned to the VMS environment upon normal completion of the procedure is the value of the first (and only) parameter.

Note that when a main function or a main procedure declared with the pragma EXPORT_VALUED_PROCEDURE returns a discrete value whose size is less than 32 bits, the value is zero or sign extended as appropriate.

F.9.2 Values of Integer Attributes

The ranges of values for integer types declared in the package STANDARD are as follows:

SHORT_SHORT_INTEGER	-128 .. 127
SHORT_INTEGER	-32768 .. 32767
INTEGER	-2147483648 .. 2147483647

For the packages DIRECT_IO, RELATIVE_IO, SEQUENTIAL_MIXED_IO, DIRECT_MIXED_IO, RELATIVE_MIXED_IO, INDEXED_MIXED_IO, and TEXT_IO, the ranges of values for the types COUNT and POSITIVE_COUNT are as follows:

COUNT	0 .. 2147483647
POSITIVE_COUNT	1 .. 2147483647

For the package TEXT_IO, the range of values for the type FIELD is as follows:

FIELD	0 .. 2147483647

F.9.3 Values of Floating Point Attributes

Attribute	F_floating value and approximate decimal equivalent (where applicable)
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON approximately	16#0.1000_000#e-4 9.53674E-07
SMALL approximately	16#0.8000_000#e-21 2.58494E-26
LARGE approximately	16#0.FFFF_F80#e+21 1.93428E+25
SAFE_EMAX	127
SAFE_SMALL approximately	16#0.1000_000#e-31 2.93874E-39
SAFE_LARGE approximately	16#0.7FFF_FC0#e+32 1.70141E+38
FIRST approximately	-16#0.7FFF_FF8#e+32 -1.70141E+38
LAST approximately	16#0.7FFF_FF8#e+32 1.70141E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	127
MACHINE_EMIN	-127
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

Attribute	D_floating value and approximate decimal equivalent (where applicable)
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON approximately	16#0.4000_0000_0000_000#e-7 9.3132257461548E-10
SMALL approximately	16#0.8000_0000_0000_000#e-31 2.3509887016446E-38
LARGE approximately	16#0.FFFF_FFFE_0000_000#e+31 2.1267647922655E+37
SAFE_EMAX	127
SAFE_SMALL approximately	16#0.1000_0000_0000_000#e-31 2.9387358770557E-39
SAFE_LARGE approximately	16#0.7FFF_FFFF_0000_000#e+32 1.7014118338124E+38
FIRST approximately	-16#0.7FFF_FFFF_FFFF_FF8#e+32 -1.7014118346047E+38
LAST approximately	16#0.7FFF_FFFF_FFFF_FF8#e+32 1.7014118346047E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	56
MACHINE_EMAX	127
MACHINE_EMIN	-127
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

Attribute	G_floating value and approximate decimal equivalent (where applicable)
DIGITS	15
MANTISSA	51
EMAX	204
EPSILON approximately	16#0.4000_0000_0000_00#e-12 8.8817841970001E-16

Attribute	G_floating value and approximate decimal equivalent (where applicable)
SMALL	16#0.8000_0000_0000_00#e-51
approximately	1.944692274332E-62
LARGE	16#0.FFFF_FFFF_FFFF_E0#e+51
approximately	2.571100870814E+61
SAFE_EMAX	1023
SAFE_SMALL	16#0.1000_0000_0000_00#e-255
approximately	5.562684646268E-309
SAFE_LARGE	16#0.7FFF_FFFF_FFFF_F0#e+256
approximately	8.988465674312E+307
FIRST	-16#0.7FFF_FFFF_FFFF_FC#e+256
approximately	-8.988465674312E+307
LAST	16#0.7FFF_FFFF_FFFF_FC#e+256
approximately	8.988465674312E+307
MACHINE_RADIX	2
MACHINE_MANTISSA	53
MACHINE_EMAX	1023
MACHINE_EMIN	-1023
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

Attribute	H_floating value and approximate decimal equivalent (where applicable)
DIGITS	33
MANTISSA	111
EMAX	444
EPSILON	16#0.4000_0000_0000_0000_0000_0000_0000_0#e-27
approximately	7.703719777548943412223911770339 7E-34
SMALL	16#0.8000_0000_0000_0000_0000_0000_0000_0#e-111
approximately	1.100656821463791821093431802093 6E-134
LARGE	16#0.FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFE_0#e+111
approximately	4.542742026847543065933273799300 0E+133
SAFE_EMAX	16383

Attribute	H_floating value and approximate decimal equivalent (where applicable)
SAFE_SMALL	16#0.1000_0000_0000_0000_0000_0000_0000_0#e-4095
approximately	8.40525785778023376565669454330 44E-4933
SAFE_LARGE	16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_0#e+4096
approximately	5.94865747678615882542879663314 00E+4931
FIRST	-16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#e+4096
approximately	-5.94865747678615882542879663314 00E+4931
LAST	16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#e+4096
approximately	5.94865747678615882542879663314 00E+4931
MACHINE_RADIX	2
MACHINE_MANTISSA	113
MACHINE_EMAX	16383
MACHINE_EMIN	-16383
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.4 Attributes of Type DURATION

The values of the significant attributes of the type DURATION are as follows

DURATION'DELTA	1.00000E-04
DURATION'SMALL	2^{-14}
DURATION'FIRST	-131072.0000
DURATION'LAST	131071.9999
DURATION'LARGE	1.31071999938964843 75E+05

F.9.5 Implementation Limits

Limit	Description
32	Maximum number of formal parameters in a subprogram or entry declaration that are of an unconstrained record type
255	Maximum identifier length (number of characters)
255	Maximum number of characters in a source line

Limit	Description
245	Maximum number of discriminants for a record type
246	Maximum number of formal parameters in an entry or subprogram declaration
255	Maximum number of dimensions in an array type
1023	Maximum number of library units and subunits in a compilation closure[1]
4095	Maximum number of library units and subunits in an execution closure[2]
32757	Maximum number of objects declared with the pragma PSECT_OBJECT
65535	Maximum number of enumeration literals in an enumeration type definition
65535	Maximum number of characters in a value of the predefined type STRING
65535	Maximum number of frames that an exception can propagate
65535	Maximum number of lines in a source file
$2^{31} - 1$	Maximum number of bits in any object

[1]The compilation closure of a given unit is the total set of units that the given unit depends on, directly and indirectly.

[2]The execution closure of a given unit is the compilation closure plus all associated secondary units (library bodies and subunits).

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.

$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	1..254 => 'A', 255 => '1'
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	1..254 => 'A', 255 => '2'
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	1..127 => 'A', 128 => '3', 129..255 => 'A'
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	1..127 => 'A', 128 => '4', 129..255 => '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..252 => '0', 253..255 => '298'
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the	1..250 => '0', 251..255 => '690.0'

maximum line length.

$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	1..195 => "A"
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	196..254 => "127, 255 => "1"
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	1..235 => ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2**31-1
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	VAXELN
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2.0**(-31)
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2_147_483_647
$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	LONG_LONG_FLOAT

FLOAT, SHORT_FLOAT, or LONG_FLOAT.

$GREATER_THAN_DURATION 75_000.0
A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.

$GREATER_THAN_DURATION_BASE_LAST 131_073.0
A universal real literal that is greater than DURATION'BASE'LAST.

$HIGH_PRIORITY 15
An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.

$ILLEGAL_EXTERNAL_FILE_NAME1 BADCHAR^@.~!
An external file name which contains invalid characters.

$ILLEGAL_EXTERNAL_FILE_NAME2
THIS-FILE-NAME-WOULD-BE-PERFECTLY-LEGAL-IF-IT-WERE-NOT-SO-LONG.SO_THERE
An external file name which is too long.

$INTEGER_FIRST -2147483648
A universal integer literal whose value is INTEGER'FIRST.

$INTEGER_LAST 2147483647
A universal integer literal whose value is INTEGER'LAST.

$INTEGER_LAST_PLUS_1 2_147_483_648
A universal integer literal whose value is INTEGER'LAST + 1.

$LESS_THAN_DURATION -75_000.0
A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.

$LESS_THAN_DURATION_BASE_FIRST -131_073.0
A universal real literal that is less than DURATION'BASE'FIRST.

$LOW_PRIORITY 0
An integer literal whose value is the lower bound of the range

for the subtype SYSTEM.PRIORITY.

$MANTISSA_DOC
An integer literal whose value is SYSTEM.MAX_MANTISSA.

31

$MAX_DIGITS
Maximum digits supported for floating-point types.

33

$MAX_IN_LEN
Maximum input line length permitted by the implementation.

255

$MAX_INT
A universal integer literal whose value is SYSTEM.MAX_INT.

2147483647

$MAX_INT_PLUS_1
A universal integer literal whose value is SYSTEM.MAX_INT+1.

2_147_483_648

$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..250 => '0',
251..255 => '11:'

$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..2 => '16:', 3..248 => '0'
249..255 => '16:F.E'

$MAX_STRING_LITERAL
A string literal of size MAX_IN_LEN, including the quote characters.

1 => '"',2..254 => "A",
255 => '"'

$MIN_INT
A universal integer literal whose value is SYSTEM.MIN_INT.

-2147483648

$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.

32

$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	SHORT_SHORT_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	VAX_VMS
$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#FFFFFFFE#
$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.	1_048_576
$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.	8
$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.	VAXELN
$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 inout parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	10.0**(-2)

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-dddd is to an Ada Commentary.

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

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

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).

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).

CD2A63A..D, CD2A66A..D, CD2A73A..D, 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 3.4.1(4)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD2A81G, CD2A83G, CD2A84M & N, & CD2C11C
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, 86 & 96, and 66, resp.).

CD2B15C & 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.

CD2D11B
This test gives a SMALL representation clause for a derived fixed-point type (at line 34) 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.

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

ED7004B, ED7005C & D, 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.

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).

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

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.

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)

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

CE3301A
This test contains several calls to END_OF_LINE & 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, & 136).

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.

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 that must appear at the top of the page.

APPENDIX E

COMPILER OPTIONS AS SUPPLIED BY

Digital Equipment Corporation

Compiler: VAX Ada Version 2.0

ACVC Version: 1.10

. /ANALYSIS_DATA or /NOANALYSIS_DATA

Controls whether a data analysis file containing source code cross-referencing and static analysis information is created. The default is /NOANALYSIS_DATA.

. /CHECK or /NOCHECK

Controls whether run-time error checking is suppressed. (Use of /NOCHECK is equivalent to giving all possible suppress pragmas in the source program.) The default is /CHECK (error checking is not suppressed except by pragma).

. /COPY_SOURCE or /NOCOPY_SOURCE

Controls whether the source being compiled is copied into the compilation library for a successful compilation. The default is /COPY_SOURCE.

. /DEBUG or /NODEBUG or /DEBUG=option

where option is one of

ALL, SYMBOLS or NOSYMBOLS, TRACEBACK or NOTRACEBACK, or NONE

Controls the inclusion of debugging symbol table information in the compiled object module. The default is /DEBUG or, equivalently, /DEBUG=ALL.

- /DIAGNOSTICS, /DIAGNOSTICS=filename, or /NODIAGNOSTICS

  Controls whether a special diagnostics file is produced for use with the VAX Language-Sensitive Editor (a separate DIGITAL product). The default is /NODIAGNOSTICS.

- /ERROR_LIMIT=n

  Controls the number of error level diagnostics that are allowed within a single compilation unit before the compilation is aborted. The default is /ERROR_LIMIT=30.

- /LIBRARY=directory-name

  Specifies the name of the Ada compilation library to be used as the context for the compilation. The default is the library last established by the ACS SET LIBRARY command.

- /LIST, /LIST=filename, or /NOLIST

  Controls whether a listing file is produced. /LIST without a filename uses a default filename of the form sourcename.LIS, where sourcename is the name of the source file being compiled. The default is /NOLIST (for both interactive and batch mode).

- /MACHINE_CODE or /NOMACHINE_CODE

  Controls whether generated machine code (approximating assembler notation) is included in the listing file, if produced. The default is /NOMACHINE_CODE.

- /NOTE_SOURCE or /NONOTE_SOURCE

  Controls whether the file specification of the current source file is noted in the compilation library. (This copy is used for certain automated recompilation features.) The default is /NOTE_SOURCE.

- /OPTIMIZE or /NOOPTIMIZE

  Controls whether full or minimal optimization is applied in producing the compiled code. The default is /OPTIMIZE. (/NOOPTIMIZE is primarily of use in combination with /DEBUG.)

- /SHOW=PORTABILITY or /NOSHOW

  Controls whether a portability summary is included in the listing. The default is /SHOW=PORTABILITY.

/WARNINGS=(category:destination,...)

Specifies which categories of informational and warning level messages are displayed for which destinations. The categories can be WARNINGS, WEAK_WARNINGS, SUPPLEMENTAL, COMPILATION_NOTES AND STATUS. The destinations can be ALL, NONE or combinations of TERMINAL, LISTING or DIAGNOSTICS. The default is

/WARNINGS=(WARN:ALL,WEAK:ALL,SUPP:ALL,COMP:NONE,STAT:LIST)

END

FILMED

6-89

DTIC