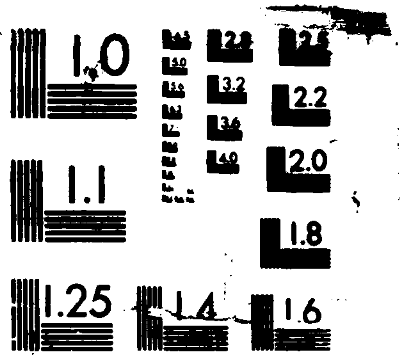
AD-A181 891 ADA (TRADENAME) COMPILER VALIDATION SUMMARY REPORT VAX ADA V13 VERSION 18 (U) FEDERAL SOFTWARE MANAGEMENT SUPPORT CENTER FALLS CHURCH VA 07 NOV 86 1/1

UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

2

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETEING FORM
1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Ada Compiler Validation Summary Report: VAX Ada V1.3, Version 1.8 of the Ada Compiler Validation Capability (ACVC)	5. TYPE OF REPORT & PERIOD COVERED 7 NOV 1986 to 7 NOV 1987
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* Federal Software Management Support Center	8. CONTRACT OR GRANT NUMBER*(s)*
9. PERFORMING ORGANIZATION AND ADDRESS Federal Software Management Support Center, 5203 Leesburg Pike, Suite 1100 Falls Church, VA 22041-3467	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 20301-3081ASD/SIOL	12. REPORT DATE 7 NOV 1986
	13. NUMBER OF PAGES 71
14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)* Federal Software Management Support Center	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

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)*

See Attached.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

EXECUTIVE SUMMARY

This Validation Summary Report summarizes the results and conclusions of validation testing performed on the VAX Ada V1.3 using Version 1.8 of the *Ada Compiler Validation Capability (ACVC).

The validation process includes submitting a suite of standardized tests (the ACVC) as inputs to an Ada compiler and evaluating the results. The purpose is to ensure conformance of the computer to ANSI/MIL-STD-1815A Ada by testing that it properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by ANSI/MIL-STD-1815A. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, or during execution.

On-site testing was performed 3 Nov 1986 through 7 Nov 1986 at Nashua, NH under the auspices of the Federal Software Management Support Center, according to Ada Validation Organization policies and procedures. The VAX Ada V1.3 is hosted on the VAX series operating under VAX/VMS V4.4 and the MicroVMS, V4.4.

The results of validation are summarized in the following table:

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	69	865	1329	17	13	46	2339
Failed	0	0	0	0	0	0	0
Inapplicable	0	2	39	0	0	0	41
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

FSV86VSRDEC540A

*Ada COMPILER VALIDATION SUMMARY REPORT: Digital Equipment Corp. VAX Ada V1.3

The host environment is the VAX series* of computers under VAX/VMS V4.4, and the MicroVAX II and VAXstation II under MicroVMS V4.4. The target environments are all hosts, and the MicroVAX II using the VAXELN Toolkit, V2.2 in combination with VAXELN Ada, V1.1.

Completion of On-Site Validation:
7 Nov 1986

Prepared By:
Federal Software Management Support Center
5203 Leesburg Pike
Suite 1100
Falls Church, Va 22041-3467

[illegible]	✓
[illegible]	☐
[illegible]	☐
[illegible]	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	25

QUALITY INSPECTED 4

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C.

*VAX series includes the VAX-11/730, VAX-11/750, Vax-11/780, VAX-11/782, VAX-11/785, VAX-11/8200, VAX-11/8300, VAX-11/8500, VAX-11/8600, VAX-11/8650, VAX-11/8700, VAX-11/8800

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

Ada Compiler Validation Summary Report:

Compiler Name: VAX Ada V1.3

Host Computer:	Target Computer:
VAX 8800 - - - - - - - -	VAX-11/750
	VAX-11/785
	VAX 8200
	VAX 8700
	VAX 8800
VAX-11/780 - - - - - - -	VAX-11/730
	VAX-11/780
	VAX-11/782
	VAX 8300
	VAX 8500
	VAX 8600
	VAX 8650
under	under
VAX/VMS	VAX/VMS
and	
VAX 8800	MicroVAX II
under	under
VAX/VMS	MicroVMS and VAXELN
and	
VAXstation II	VAXstation II under MicroVMS
under	
MicroVMS	VAX-11/780 under VAX/VMS
	MicroVAX II under VAXELN

Testing Completed on 7 Nov 1986 Using ACVC 1.8.

EXECUTIVE SUMMARY

This Validation Summary Report summarizes the results and conclusions of validation testing performed on the VAX Ada V1.3 using Version 1.8 of the *Ada Compiler Validation Capability (ACVC).

The validation process includes submitting a suite of standardized tests (the ACVC) as inputs to an Ada compiler and evaluating the results. The purpose is to ensure conformance of the computer to ANSI/MIL-STD-1815A Ada by testing that it properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by ANSI/MIL-STD-1815A. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, or during execution.

On-site testing was performed 3 Nov 1986 through 7 Nov 1986 at Nashua, NH under the auspices of the Federal Software Management Support Center, according to Ada Validation Organization policies and procedures. The VAX Ada V1.3 is hosted on the VAX series operating under VAX/VMS V4.4 and the MicroVMS, V4.4.

The results of validation are summarized in the following table:

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	69	865	1329	17	13	46	2339
Failed	0	0	0	0	0	0	0
Inapplicable	0	2	39	0	0	0	41
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

There were 19 withdrawn tests in ACVC Version 1.8 at the time of this validation attempt. A list of these test appears in Appendix D.

Some tests demonstrate that some language features are or are not supported by an implementation. For this implementation, the test determined the following.

- SHORT_INTEGER is supported.
- LONG_INTEGER is not supported.
- SHORT_FLOAT is not supported.
- LONG_FLOAT is supported.
- The additional predefined types, LONG_LONG_FLOAT and SHORT_SHORT_INTEGER are supported.
- Representation specifications for noncontiguous enumeration representations are supported.
- The 'SIZE clause is supported.
- The 'STORAGE_SIZE clause is supported.
- The 'SMALL clause is supported.
- Generic unit specifications and bodies can be compiled in separate compilations.
- Pragma INLINE is supported for procedures. Pragma INLINE is supported for functions.
- The package SYSTEM is used by package TEXT_IO.
- Mode IN_FILE is supported for sequential I/O.
- Mode OUT_FILE is supported for sequential I/O.
- Instantiation of the package SEQUENTIAL_IO with unconstrained array types is supported.
- Instantiation of the package SEQUENTIAL_IO with unconstrained record types with discriminants is supported.

- Dynamic creation and resetting of files is supported for sequential I/O.

- RESET and DELETE are supported for sequential and direct I/O.

  Modes IN_FILE, INOUT_FILE, and OUT_FILE are supported for direct I/O.

- Dynamic creation and resetting of files is supported for direct I/O.

- Instantiation of package DIRECT_IO with unconstrained array types and unconstrained types with discriminants is not supported.

- Dynamic creation and deletion of files are supported.

- More than one internal file can be associated with the same external file only for reading.

- An external file associated with more than one internal file can be reset.

- Illegal file names cannot exist.

ACVC Version 1.8 was taken on-site via magnetic tape to Nashua, NH. All tests, except the withdrawn tests and any executable tests that make use of a floating point precision greater than SYSTEM.MAX_DIGITS, were compiled on a VAX 8800 and a VAXstation II. Class A, C, D, and E tests were executed on a VAX-11/750, 785, 8200, 8700, 8800, MicroVAX II, and a VAXstation II.

On completion of testing, execution results for Class A, C, D, or E tests were examined. Compilation results for Class B were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link editing results of Class L tests were analyzed for correct detection or errors.

The Federal Software Management Support Center identified 2362 of the 2399 tests in Version 1.8 of the ACVC as potentially applicable to the validation of VAX Ada V1.3. Excluded were 18 tests with source lines that were too long; and the 19 withdrawn tests. After the 2362 tests were processed, 23 tests were determined to be inapplicable. The remaining 2339 tests were passed by the compiler.

The Federal Software Management Support Center concludes that these results demonstrate acceptable conformance to ANSI/MIL-STD-1815A.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report describes the extent to which a specific Ada compiler conforms to 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 (ANSI/MIL-STD-1815A). Any implementation-dependent features must conform to the requirements of the Ada Standard. The entire Ada Standard must be implemented, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to ANSI/MIL-STD-1815A, 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 limitations imposed on a compiler by the operating systems and by the hardware. All of the dependencies demonstrated during the process of testing this compiler are given in the report.

Validation Summary Reports are written according to a standardized format. The report for several different compilers may, therefore, be easily compared. The information in this report is derived from the test results produced during validation testing. Additional testing information is given in section 3.7 and states problems and details which are unique for a specific compiler. The format of a validation report limits variance between reports, enhances readability of the report, and minimizes the delay between the completion of validation testing and the publication of the report.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

The Validation Summary Report 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 unsupported language constructs required by the Ada Standard

- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted under the supervision of the Federal Software Management Support Center according to policies and procedures established by the Ada Validation Organization (AVO). Testing was conducted from 3 Nov 1986 through 7 Nov 1986 at Nashua, NH.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Validation organization 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 nonconformances to ANSI/MIL-STD-1815A 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
1211 S. Fern, C-107
Washington, DC 20301-3081

or from the Ada Validation Facility (AVF) listed below.

Questions regarding this report or the validation tests should be directed to:

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

or to:

Ada Validation Facility
Federal Software Management Support Center
5203 Leesburg Pike
Suite 1100
Falls Church, VA 22041-3467

1.3 RELATED DOCUMENTS

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, FEB 1983.
2. Ada Validation Organization: Policies and Procedures, MITRE Corporation, JUN 1982, PB 83-110601.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., DEC 1984.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. A set of programs that evaluates the conformance of a compiler to the Ada language specification, ANSI/MIL-STD-1815A.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	Ada Validation Facility. The Federal Software Management Support Center. In the context of this report, the AVF is responsible for conducting compiler validations according to established policies and procedures.
AVO	The Ada Validation Organization. In the content of this report, the AVO is responsible for setting policies and procedures for compiler validations.
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	A test for which the compiler generates a result that demonstrates nonconformance to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable	A test that uses features of the language that a test compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	A test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that evaluates the conformance of a compiler to a language specification. In the context of this report, the term is used to designate a single ACVC test. The text of a program may be the text of one or more compilations
Withdrawn test	A test which has been found to be inaccurate in checking conformance to the Ada language specification. A withdrawn test 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

Conformance to ANSI/MIL-STD-1815A is measured using the Ada Compiler Validation Capability (ACVC). The ACVC contains both legal and illegal Ada program 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. Legal programs are compiled, linked, and executed while illegal programs are only compiled. Special program units are used to report the results of the legal programs.

Class A tests check that legal Ada programs can be successfully compiled and executed. (However, no checks are performed during execution to see if the test objective has been met.) 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 message indicating that it has passed.

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 syntactical 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 that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NON-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 requirements placed on a compiler by the Ada Standard for some parameters (e.g., the number of identifiers permitted in a compilation, the number of units in a library, and the number of nested loops in a subprogram body), 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.

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 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.

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 results. It also provides a set of identity functions used to detect some compiler optimization strategies and force computations to be made by the target computer instead of by the compiler on the host computer. 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 these units is checked by a set of executable test. 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.

Some of the conventions followed in the ACVC 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. The values used for this validation are listed in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformance 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. 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 nonconformant tests are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

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

Compiler: VAX Ada V1.3

Test Suite: Ada Compiler Validation Capability, Version 1.8

Host Computer:

Machine(s):	VAX-11/780, VAX 8800 and VAXstation II
Operating Systems:	VAX/VMS V4.4 MicroVMS V4.4
Memory Size:	12, 32, and 8 MB

Target Computer:

Machine(s):	VAX-11/730, 750, 780, 782, VAX-11/785, 8200, 8300, 8500, VAX 8600, 8650, 8700, 8800, MicroVAX II, VAXstation II
Operating System	VAX/VMS V4.4 MicroVMS V4.4 VAXELN V2.2
Memory Size:	4 - 32MB

Communications Network:

2.2 CERTIFICATE INFORMATION

Base Configuration:

Compiler: VAX Ada V1.3

Test Suite: Ada Compiler Validation Capability, Version 1.8

Completion Date: 7 Nov 1986

Host Computer:

Machine(s):	VAX-11/730, 750, 780, 782, 785, 8200, 8300, 8500, 8600, 8650, 8700, and 8800
Operating System:	VAX/VMS, V4.4
Machine(s):	MicroVAX II, VAXstation II
Operating System:	MicroVMS, V4.4

Target Computer:

Machine(s):	VAX-11/730, 750, 780, 782, 785, 8200, 8300, 8500, 8600, 8650, 8700, 8800
Operating System:	VAX/VMS, V4.4
Machine(s):	MicroVAX II, VAXstation II
Operating System:	MicroVMS, V4.4
Machine(s):	MicroVAX II
Operating System:	VAXELN Toolkit, V2.2, in combination with VAXELN Ada, V1.1

2.3 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 implementation to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. This compiler is characterized by the following interpretations of the Ada Standard:

- Nongraphic characters.

  Nongraphic characters are defined in the ASCII character set but are not permitted in Ada programs, even within character strings. The compiler correctly recognizes these characters as illegal in Ada compilations. The characters are not printed in the output listing. (See test B26005A.)

- Capacities.

  The compiler correctly processes compilations containing loop statements nested to 65 levels, block statements nested to 65 levels, procedures nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H, D56001B, D64005E..G, D29002A)

CONFIGURATION INFORMATION

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX INT. This implementation does not reject such calculations and processes them correctly. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Universal real calculations.

  When rounding to interger is used in a static universal real expression, the value appears to be rounded away from zero. (See test C4A014A.)

- Predefined types.

  This implementation supports the additional predefined types SHORT_INTEGER, LONG_FLOAT, and SHORT_SHORT_INTEGER in the package STANDARD. (See test B86001DT.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Array types.

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

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

  A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_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.)

In assigning one-dimensional array types, the entire expression appears to be evaluated before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the entire expression does not appear to be evaluated before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminate constraint. This implementation accepts such subtype indications during compilation. (See test E38104A.)

In assigning record types with discriminants, the entire expression appears to be evaluated before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index subtype. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

CONFIGURATION INFORMATION

- Functions.

  The declaration of a parameterless function with the same profile as an enumeration literal in the same immediate scope is rejected by the implementation. (See test E66001D.)

- Representation clauses.

  The Ada standard does not require an inplementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it. While the operation of representation clauses in not checked by Version 1.8 of the ACVC, they are used in testing other language features. Testing indicates that size specifications are supported, that specification of storage for a task activation is supported, and that specification of SMALL for a fixed point type is supported. Enumeration representation clauses including those that specify noncontiguous values appear to be supported. (See tests C55B16A, C87B62A, C87B62B, C87B62C, and BC1002A.)

- Generics.

  When given a separately compiled generic unit specification, some illegal instantiations, and a body, the compiler rejects the body because of the instantiations. (See tests BC3204C and BC3204D.)

- Pragmas.

  The pragma INLINE is supported for procedures. The pragma INLINE is supported for functions. (See tests CA3004E and CA3004F.)

- Input/output.

  The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants. The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests CE2201D, CE2201E, and CE2401D.)

  More than one internal file can be associated with each external file for sequential I/O for reading only. (See tests CE2107A..F.)

  More than one internal file can be associated with each external file for direct I/O for reading only. (See tests CE2107A..F.)

  An external file associated with more than one internal file can be deleted. (See test CE2110B.)

  More than one internal file can be associated with each external file for text I/O for reading only. (See tests CE3111A..E.)

  Dynamic creation and resetting of a sequential file is allowed. (See test CE2210A.)

  Temporary sequential files are given a name. Temporary direct files are given a name. Temporary files given names are not deleted when they are closed, but are not accessible after the completion of the main program. (See test CE2108A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

The Federal Software Management Support Center identified 2362 of the 2399 tests in Version 1.8 of the Ada Compiler Validation Capability as potentially applicable to the validation of VAX Ada V1.3. Excluded were 18 tests with source lines that were too long; and the 19 withdrawn tests. After they were processed 23 tests were determined to be inapplicable. The remaining 2339 tests were passed by the compiler.

The Federal Software Management Support Center concludes that the testing results demonstrate acceptable conformance to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	69	865	1329	17	13	46	2339
Failed	0	0	0	0	0	0	0
N/A	0	2	39	0	0	0	41
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT

	2	3	4	5	6	7	8	9	10	11	12	14	Total
Passed	98	322	420	244	161	97	138	261	130	32	218	218	2339
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
N/A	18	3	0	3	0	0	1	1	0	0	0	15	41
W/D	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

3.4 WITHDRAWN TESTS

The following tests have been withdrawn from the ACVC Version 1.8:

C32114A	B37401A	B49006A	C92005A
B33203C	C41404A	B4A010C	C940ACA
C34018A	B45116A	B74101B	CA3005A..D
C35904A	C48008A	C87B50A	BC3204C

See Appendix D for the rationale for withdrawing these tests.

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. For this validation attempt, 41 tests were inapplicable for the reasons indicated:

- C96005B - there are no out-of-range values for type DURATION

- CE2107B, CE2107C, CE2107D, CE2107E, CE2111D CE3111B, CE3111C, CE3111D, CE3111E, CE3114B CE2110B

  - with default open/create options (no FORM string), VAX Ada allows more than one internal file to be associated with the same external file for mode IN_FILE only (multiple readers) , but does not allow more than one association for OUT _FILE or INOUT_FILE in combination with mode IN _FILE or another mode OUT_FILE (mixed readers and writers or multiple writers).

- CE3115A - 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 either INOUT _FILE or OUT_FILE (an amplification of accessing privileges while the external file is being accessed). Thus CE3115A does not apply.

- CE2102D, CE2102I, CE2111H - the creation of a file of mode IN_FILE is not allowed

- CE24113H..C24113Y - source lines exceed the limit of 120 characters

- B52004D, B55B09C, C34001E, C55B07A - LONG_INTEGER is not supported

- C34001F, C35702A - SHORT_FLOAT is not supported

- C86001F - TEXT_IO uses the predefined package SYSTEM, which is made obsolete by the user defined package SYSTEM

3.6 SPLIT TESTS

If one or more errors do not appear to have been detected in a Class B test because of compiler error recovery, then the test is split into a set of smaller tests that contain the undetected errors. There were no split tests required for this implementation.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, sets of test results for ACVC Version 1.8 produced by VAX Ada V1.3 were submitted to the Federal Software Management Support Center by the applicant for pre-validation review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests.

The specific configurations submitted for the pre-validation review were as follows:

Host		Target	
Processor	Op. Sys.	Processor	Op. Sys.
VAX-11/780	VAX/VMS	VAX-11/730	VAX/VMS
VAX-11/780	VAX/VMS	VAX-11/780	VAX/VMS
VAX-11/780	VAX/VMS	VAX-11/782	VAX/VMS
VAX-11/780	VAX/VMS	VAX 8300	VAX/VMS
VAX-11/780	VAX/VMS	VAX 8500	VAX/VMS
VAX-11/780	VAX/VMS	VAX 8600	VAX/VMS
VAX-11/780	VAX/VMS	VAX 8650	VAX/VMS
VAXstation II	MicroVMS	VAX-11/780	VAX/VMS
VAXstation II	MicroVMS	MicroVAX II	VAXELN

The VAX-11/782 results were compared against the VAX-11/730, 780, 8300, 8500, 8600 and the 8650 and found to be equivalent.

The results from the Vax-11/780 were compared against the MicroVAX II, 730, 782, 8300, 8500, 8600, 8650 and 780 and found to be equivalent.

The results produced by VAX Ada were the same for all tested members of the VAX family--for those using VMS, MicroVMS, or VAXELN.

3.7.2 Test Method

A test magnetic tape containing ACVC Version 1.8 was taken on-site by the validation team. This magnetic tape contained all tests applicable to this validation as well as all tests inapplicable to this validation except for any Class C tests that require floating-point precision exceeding the maximum value supported by the implementation. Tests that were withdrawn from ACVC Version 1.8 were not run. Tests that make use of values that are specific to an implementation were customized before being written to the magnetic tape.

The test tape was written in VAX BACKUP format and was loaded to disk using Digital Equipment Corp. standard utility routines.

Once all tests had been loaded to disk, processing was begun using command scripts provided by Digital Equipment Corp.

The validation was executed in batch control mode with the files organized by chapter and class to allow the tests to be run independently and in parallel.

A new compilation library was created and initialized with all units contained in the library given the logical name ADA$PREDEFINED. The startup control file established the newly created library as the current compilation library and then compiled REPORT and CHECK_FILE into that library.

The prevalidation results were verified on-site. The various tests results from the prevalidation execution were captured on disk and used to compare against the on-site results using "DIF", a difference utility.

The OPTIMIZE option was used to produce the compiled code.

The following configurations were tested on-site:

Host	Op. Sys.	Target	Op. Sys.
VAX 8800	VAX/VMS	VAX-11/750	VAX/VMS
		VAX-11/785	
		VAX 8200	
		VAX 8700	
		VAX 8800	
		MicroVAX II	MicroVMS
		MicroVAX II	VAXELN
VAXstation II	MicroVMS	VAXstation II	MicroVMS

3.7.3 Test Site

The validation team arrived at Nashua, NH on 3 Nov 1986 and departed after testing was completed on 7 Nov 1986.

APPENDIX A

COMPLIANCE STATEMENT

Digital Equipment Corporation has submitted the following compliance statement concerning VAX Ada and VAXELN Ada.

Compliance Statement

Base Configuration:

```
    Compiler: VAX Ada Version 1.3
    Test Suite: Ada Compiler Validation Capability, Version V1.8

    Host Computers:

        Machines:
            VAX-11/730, VAX-11/750, VAX-11/780, VAX-11/782,
            VAX-11/785, VAX 8200, VAX 8300, VAX 8500,
            VAX 8600, VAX 8650, VAX 8700, and VAX 8800.
        Operating System:
            VAX/VMS, Version 4.4

        Machines:
                        MicroVAX II,                and
            VAXstation II.
        Operating System:
            MicroVMS, Version 4.4

    Target Computers (same as host plus VAXELN):

        Machines:
            VAX-11/730, VAX-11/750, VAX-11/780, VAX-11/782,
            VAX-11/785, VAX 8200, VAX 8300, VAX 8500,
            VAX 8600, VAX 8650, VAX 8700, and VAX 8800.
        Operating System:
            VAX/VMS, Version 4.4

        Machines:
                        MicroVAX II,                and
            VAXstation II.
        Operating System:
            MicroVMS, Version 4.4

        Machines:
                        MicroVAX II
        Operating System:
            VAXELN Toolkit, Version 2.2, in combination with
            VAXELN Ada, Version 1.1.
```

COMPLIANCE STATEMENT

Digital Equipment Corporation has made no deliberate extensions to the Ada language standard.

Digital Equipment Corporation agrees to public disclosure of this report.

Digital Equipment Corporation agrees to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office.

[illegible]

6 October 1986

Charles Z. Mitchell
VAX Ada Project Leader

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 MIL-STD-1815A, 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 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.5.

(6) Address Clauses

See Attachment 2, Section F.6.

(7) Unchecked Conversions

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

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

2. The actual subtype coresponding to the formal type TARGET must not be an unconstrained type with discriminants.

(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

Implementation-Dependent Pragmas

1 This attachment defines the pragmas LIST, PAGE, and OPTIMIZE, and summarizes the definitions given elsewhere of the remaining language-defined pragmas. VAX Ada implementation-dependent information (including the VAX Ada implementation-dependent pragmas) is marked with change bars.

The VAX Ada pragma TITLE is 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 VAX/VMS asynchronous system trap (AST) resulting from a

		VAX/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

	VAX/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. The 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 VAX/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 VAX/VMS Linker global symbol) and size designator (a VAX/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 VAX/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 VAX/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).
IMPORT_EXCEPTION	Takes an internal name denoting an exception, and optionally takes an external designator (the name of a VAX/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 VAX/VMS Linker global symbol), parameter types, result type, and mechanism arguments. 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 VAX/VMS Linker global symbol) and size (a VAX/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 VAX/VMS Linker global symbol) parameter types, and mechanism as arguments. 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 VAX/VMS Linker global symbol), parameter types, and mechanism as arguments. 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).
4	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).

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 pragma INTERFACE is required in combination with pragmas IMPORT_FUNCTION, IMPORT_PROCEDURE, and IMPORT_VALUED_PROCEDURE (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 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 and/or guard pages. 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 VAX/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).

VAX Ada does not support pragma SHARED (see VOLATILE).

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 eight (8).

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).

VAX Ada does not support pragma SUPPRESS (see SUPPRESS_ALL).

	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 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. Pragma TITLE has the 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 string(s) supersede(s) 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 VOLATILE pragma 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

VAX Ada Appendix F

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

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

F.2 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 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)
- 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)
- 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 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**(-30);
    TICK            : constant      := 10.0**(-2);

    subtype PRIORITY is INTEGER range 0 .. 15;

--  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 hve 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),

   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);

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

   type D_FLOAT is implementation_defined,
   type F_FLOAT is implementation_defined,
   type G_FLOAT is implementation_defined,
   type H_FLOAT is implementation_defined,

-- 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 (LEFT : BIT_ARRAY_8) return UNSIGNED_BYTE;
function TO_BIT_ARRAY_8 (LEFT : 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 (LEFT : BIT_ARRAY_16) return UNSIGNED_WORD;
function TO_BIT_ARRAY_16 (LEFT : 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;

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 (LEFT : BIT_ARRAY_32)
   return UNSIGNED_LONGWORD;
function TO_BIT_ARRAY_32 (LEFT : 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;

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 (LEFT : BIT_ARRAY_64)
   return UNSIGNED_QUADWORD;
function TO_BIT_ARRAY_64 (LEFT : 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_SHORT_INTEGER is
       record
          VALUE : SHORT_INTEGER := 0;
       end record;
    for  ALIGNED_SHORT_INTEGER use
       record
          at mod 2;
       end record;

    procedure ADD_INTERLOCKED (ADDEND  : in     SHORT_INTEGER;
                               AUGEND  : in out ALIGNED_SHORT_INTEGER;
                               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 Conventions for Implementation-Generated Names Denoting Implementation-Dependent Components in Record Representation Clauses

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

F.6 Interpretation of Expressions Appearing in Address Clauses

Expressions appearing in address clauses must be of the type ADDRESS defined in package SYSTEM (see 13.7a.1 and F.3). In VAX Ada, values of type SYSTEM.ADDRESS are interpreted as integers in the range 0..MAX_INT, and they refer to addresses in the user half of the VAX address space.

VAX Ada allows address clauses for variables (see 13.5.

VAX Ada does not support interrupts.

F.7 Restrictions on Unchecked Type Conversions

VAX Ada supports the generic function UNCHECKED_CONVERSION with the restrictions given in section 13.10.2.

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 RMS input-output facilities, VAX Ada provides packages in addition to 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 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 the FORM parameter to specify external file attributes, see the *VAX Ada Run-Time Reference Manual*. For information on FDL, see the *Guide to VAX/VMS File Applications* and the *VAX/VMS File Definition Language Facility Reference 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 occurring 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 USE_ERROR to be raised.

USE_ERROR is raised under the following conditions:

- 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 the WRITE operations on relative or indexed files if the element in the position indicated has already been written.
- In the UPDATE and DELETE_ELEMENT operations on relative or indexed files if the element to be updated or deleted is not locked.
- In the UPDATE operations on indexed files 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.
- If the capacity of the external file has been exceeded.

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 VAX/VMS file specification or a VAX/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 having to do with 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 library unit can be used as a main program provided it has no formal parameters and, in the case of a function, if its returned value is a discrete type. If the main program is a procedure, the status returned to the VAX/VMS environment upon normal completion of the procedure is the value one. If the main procedure is a function, the status returned is the function value. Note that when a main function 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 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 range of values for 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
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	16#0.1000_000#e-4
approximately	9.53674E-07
SMALL	16#0.8000_000#e-21
approximately	2.58494E-26
LARGE	16#0.FFFF_F80#e+21
approximately	1.93428E+25

Attribute	F_Floating Value and Approximate Decimal Equivalent
SAFE_EMAX	127
SAFE_SMALL	16#0.1000_000#e-31
approximately	2.93874E-39
SAFE_LARGE	16#0.7FFF_FC0#e+32
approximately	1.70141E+38
FIRST	-16#0.7FFF_FF8#e+32
approximately	-1.70141E+38
LAST	16#0.7FFF_FF8#e+32
approximately	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
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON	16#0.4000_0000_0000_000#e-7
approximately	9.3132257461548E-10
SMALL	16#0.8000_0000_0000_000#e-31
approximately	2.3509887016446E-38
LARGE	16#0.FFFF_FFFE_0000_000#e+31
approximately	2.1267647922655E+37
SAFE_EMAX	127
SAFE_SMALL	16#0.1000_0000_0000_000#e-31
approximately	2.9387358770557E-39

Attribute	D_Floating Value and Approximate Decimal Equivalent
SAFE_LARGE	16#0.7FFF_FFFF_0000_000#e+32
approximately	1.7014118338124E+38
FIRST	-16#0.7FFF_FFFF_FFFF_FF8#e+32
approximately	-1.7014118346047E+38
LAST	16#0.7FFF_FFFF_FFFF_FF8#e+32
approximately	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
DIGITS	15
MANTISSA	51
EMAX	204
EPSILON	16#0.4000_0000_0000_00#e-12
approximately	8.881784197001E-016
SMALL	16#0.8000_0000_0000_00#e-51
approximately	1.944692274332E-062
LARGE	16#0.FFFF_FFFF_FFFF_E0#e+51
approximately	2.571100870814E+061
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

Attribute	G_Floating Value and Approximate Decimal Equivalent
FIRST	-16#0.7FFF_FFFF_FFFF_FC#e+256
approximately	-8.9884656743312E+307
LAST	16#0.7FFF_FFFF_FFFF_FC#e+256
approximately	8.9884656743312E+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
DIGITS	33
MANTISSA	111
EMAX	444
EPSILON	16#0.4000_0000_0000_0000_0000_0000_0000_0#e-27
approximately	7.7037197775489434122239117703397E-0034
SMALL	16#0.8000_0000_0000_0000_0000_0000_0000_0#e-111
approximately	1.1006568214637918210934318020936E-0134
LARGE	16#0.FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFE_0#e+111
approximately	4.5427420268475430659332737993000E+0133
SAFE_EMAX	16383
SAFE_SMALL	16#0.1000_0000_0000_0000_0000_0000_0000_0#e-4095
approximately	8.4052578577802337656566945433044E-4933
SAFE_LARGE	16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_0#e+4096
approximately	5.9486574767861588254287966331400E+4931
FIRST	-16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#e+4096
approximately	-5.9486574767861588254287966331400E+4931

Attribute	H_Floating Value and Approximate Decimal Equivalent
LAST	16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#e+4096
approximately	5.9486574767861588254287966331400E+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 type DURATION are as follows:

DURATION'DELTA	1.00000E-04
DURATION'SMALL	2^{-14}
DURATION'FIRST	-131072.0000
DURATION'LAST	131071.9999
DURATION'LARGE	1.3107199993896484375E+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
120	Maximum identifier length (number of characters)
120	Maximum number of characters in a source line
245	Maximum number of discriminants for a record type

Limit	Description
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 PSECT_OBJECT pragmas
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 identified by names that begin with a dollar sign. A value is substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier of size MAX_IN_LEN with varying last character.	119 A's and a '1'
$BIG_ID2 Identifier of size MAX_IN_LEN with varying last character.	119 A's and a '2'
$BIG_ID3 Identifier of size MAX_IN_LEN with varying last character.	119 A's and a '3' in the middle
$BIG_ID4 Identifier of size MAX_IN_LEN with varying last character.	119 A's and a '4' in the middle
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is MAX_IN_LEN characters long.	116 0's and 0298

Name and Meaning	Value
$BIG_REAL_LIT A real literal that can be either of floating or fixed point type, has value 690.0, and has enough leading zeroes to be MAX_IN_LEN characters long.	114 0's and 69.0E1
$BLANKS Blanks of length MAX_IN_LEN - 20	BLANKS
[illegible]NT LAST Value of CNT'LAST in TEXT_IO package.	2147483647
$E[illegible]D_ASCII_CHARS A [illegible]ng literal containing all the [illegible]CII characters with [illegible]ble graphics that are not [illegible] basic 55 Ada character [illegible]	abcdefghijklmnopqrstuvwxyz!$%?@[\]^'{}~
$FIELD_LAST Value of Field'LAST in TEXT_IO package.	2147483647
$FILE_NAME_WITH_BAD_CHARS An illegal external file name that either contains invalid characters or is too long.	X}]!@#$^&~Y
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	XYZ*
$GREATER_THAN_DURATION A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST The universal real value that is greater than DURATION'BASE'LAST.	10_000_000.0
$ILLEGAL_EXTERNAL_FILE_NAME Illegal external file name.	BAD-CHARACTER*^

Name and Meaning	Value
$ILLEGAL_EXTERNAL_FILE_NAME2 MUCH-TOO-LONG-NAME-FOR-A-FILE-MUCH-TOO-LONG-NAME-FOR-A-FILE Illegal external file names.	
$INTEGER_FIRST The universal integer literal expression whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST The universal integer literal expression whose value is INTEGER'LAST.	2147483647
$LESS_THAN_DURATION A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST The universal real value that is less then DURATION'BASE'FIRST.	-10_000_000.0
$MAX_DIGITS floating-point types.	33
$MAX_IN_LEN Maximum input line length permitted by the implementation.	120
$NAME A name of predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER,	SHORT_SHORT_INTEGER
$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#
$NON_ASCII_CHAR_TYPE An enumerated type definition for a character type whose literals are the identifier NON_NULL and all non-ASCII characters with printable graphics.	(NON_NULL)

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. When testing was performed, the following 19 tests had been withdrawn at the time of validation testing for the reasons indicated:

- B4A010C: The object_declaration in line 18 follows a subprogram body of the same declarative part.
- BC3204C: The file BC3204C4 should contain the body for BC3204CO as indicated in line 25 of BC3204C3M.
- C35904A: The elaboration of subtype declarations SFX3 and SFX4 may raise NUMERIC_ERROR (instead of CONSTRAINT_ERROR).
- C41404A: The values of 'LAST and 'LENGTH are incorrect in IF statements from line 74 to the end of the test.
- C48008A: This test requires that the evaluation of default initial values not occur when an exception is raised by an allocator. However, the Language Maintenance Committee (LMC) has ruled that such a requirement is incorrect (AI-00397/01).
- C32114A: An unterminated string literal occurs at line 62.
- B33203C: The reserved word "IS" is misspelled at line 45.
- C34018A: The call of function G at line 114 is ambiguous in the presence of implicit conversions and inconsistente without.
- B37401A: The object declarations at lines 126-135 follow subprogram bodies declared in the same declarative part.
- B45116A: ARRPRIBL1 and ARRPRIBL2 are initialized with a value of the wrong type (PRIBOOL_TYPE instead of ARRPRIBOOL_TYPE) at line 41.
- B49006A: Object declaratives at lines 41 and 50 are terminated incorrectly with colons; "END CASE;" is missing from line 42.

- B74101B: The "BEGIN" at line 9 is mistaken; it causes the declarative part to be treated as a sequence of statements.

- C87B50A: The call of "/=" at line 31 requires a "USE" clause for package A.

- C92005A: At line 40, "/=" for type PACK.BIG_INT is not visible without a "USE" clause for package PACK.

- C940ACA: This test assumes that allocated task TT1 will run prior to the main program, and thus assign SPYNUMB the value checked for by the main program; however, such an execution order is not required by the Ada Standard, so the test is erroneous.

- CA3005A..D (4 tests): No valid elaboration order exists for these tests.

END OF LIST

END
8-87
DTIC