

2

REPORT INFORMATION PAGE

AD-A234 350

Form Approved
OPM No. 0704-0188

Page 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data needed to complete the review estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE		3. REPORT TYPE AND DATES COVERED Final: Dec 12, 1990 to Mar 1, 1993	
4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: Digital Equipment Corporation, VAX Ada, Version 2.2, BAX 8800 (Host) to VAX MicroVAX II running VAXELN Version 4.1 (Target), 901109S1.11054				5. FUNDING NUMBERS	
6. AUTHOR(S) National Institute of Standards and Technology Gaithersburg, MD USA					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) National Institute of Standards and Technology National Computer Systems Laboratory Bldg. 255, Rm A266 Gaithersburg, MD 20899 USA				8. PERFORMING ORGANIZATION REPORT NUMBER NIST90DEC505_2_1.11	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Pentagon, RM 3E114 Washington, D.C. 20301-3081				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Digital Equipment Corporation, VAX Ada, Version 2.2, Gaithersburg, MD, VAX 8800 (Host) to VAX MicroVAX II running VAXELN Version 4.1 (Target), ACVC 1.11.					
14. SUBJECT TERMS Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.				15. NUMBER OF PAGES	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT		

AVF Control Number: NIST90DEC505_2_1.11
DATE COMPLETED
BEFORE ON-SITE: October 30, 1990
AFTER ON-SITE: November 9, 1990
REVISIONS: December 12, 1990

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901109S1.11054
Digital Equipment Corporation
VAX Ada, Version 2.2
VAX 8800 => VAX MicroVAX II running VAXELN Version 4.1

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

20

1
A-1

AVF Control Number: NIST90DEC505_2_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on November 09, 1990.

Compiler Name and Version: VAX Ada, Version 2.2

Host Computer System: VAX 8800

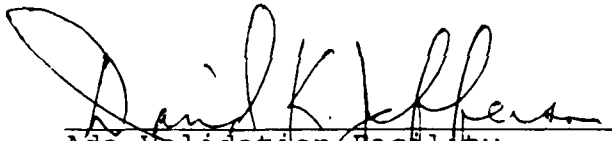
Target Computer System: VAX MicroVAX II running VAXELN
Version 4.1

Target Runtime System: VAXELN Ada Version 2.2

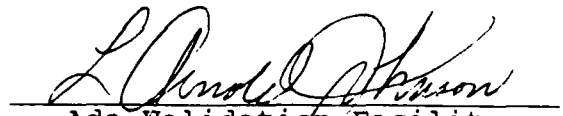
A more detailed description of this Ada implementation is found in section 3.1 of this report.

As a result of this validation effort, Validation Certificate 901109S1.11054 is awarded to Digital Equipment Corporation. This certificate expires on March 01, 1993.

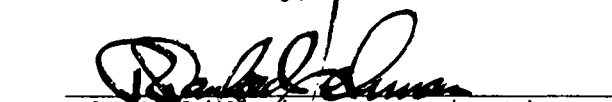
This report has been reviewed and is approved.



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



Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software
Validation Group
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899



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



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

DECLARATION OF CONFORMANCE

Customer: Digital Equipment Corporation

Certificate Awardee:

Ada Validation Facility: National Institute of Standards and
Technology
National Computer Systems Laboratory
(NCSL)
Software Validation Group
Building 225, Room A266
Gaithersburg, Maryland 20899

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: VAX Ada Version 2.2

Host Computer System: VAX 8800 running VMS Version 5.4

Target Computer System: MicroVAX II running VAXELN Version 4.1

Target Runtime System: VAXELN Ada Version 2.2

Declaration:

[I/we] the undersigned, declare that [I/we] have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

Charles F. Mitchell, Digital Equipment Corp. 18 Oct 1992
Customer Signature Date
Company

Certificate Awardee Signature
Company

Date

TABLE OF CONTENTS

CHAPTER 1	1-1
INTRODUCTION	1-1
1.1 USE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2 REFERENCES	1-1
1.3 ACVC TEST CLASSES	1-2
1.4 DEFINITION OF TERMS	1-3
CHAPTER 2	2-1
IMPLEMENTATION DEPENDENCIES	2-1
2.1 WITHDRAWN TESTS	2-1
2.2 INAPPLICABLE TESTS	2-1
2.3 TEST MODIFICATIONS	2-4
CHAPTER 3	3-1
PROCESSING INFORMATION	3-1
3.1 TESTING ENVIRONMENT	3-1
3.2 SUMMARY OF TEST RESULTS	3-1
3.3 TEST EXECUTION	3-2
APPENDIX A	A-1
MACRO PARAMETERS	A-1
APPENDIX B	B-1
COMPILATION SYSTEM OPTIONS	B-1
LINKER OPTIONS	B-4
APPENDIX C	C-1
APPENDIX F OF THE Ada STANDARD	C-1

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

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

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

1.2 REFERENCES

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

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint

Program Office, August 1990.

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

1.3 ACVC TEST CLASSES

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

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued. Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

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

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

modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

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

1.4 DEFINITION OF TERMS

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

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 81 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 90-10-12.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026A
B83026B	C83041A	B85001L	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE21171	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Issues and commonly referenced in the format AI-dddd. For this implementation, the following tests were inapplicable for the reasons indicated; references to Ada Issues are included as appropriate.

C24113W..Y (3 TESTS) USE A LINE LENGTH IN THE INPUT FILE WHICH EXCEEDS 255 CHARACTERS.

THE FOLLOWING 21 TESTS CHECK FOR THE PREDEFINED TYPE LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C

C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35702A, C35713B, C45423B, B86001T, AND C86006H CHECK FOR THE PREDEFINED TYPE SHORT_FLOAT.

C45531M, C45531N, C45531O, C45531P, C45532M, C45532N, C45532O, AND C45532P CHECK FIXED-POINT OPERATIONS FOR TYPES THAT REQUIRE A SYSTEM.MAX_MANTISSA OF 47 OR GREATER.

C45624A and C45624B check that the proper exception is raised if MACHINE_OVERFLOW is FALSE for floating-point types; for this implementation, MACHINE_OVERFLOW is TRUE.

C86001F RECOMPILES PACKAGE SYSTEM, MAKING PACKAGE TEXT_IO, AND HENCE PACKAGE REPORT, OBSOLETE. FOR THIS IMPLEMENTATION, THE PACKAGE TEXT_IO IS DEPENDENT UPON PACKAGE SYSTEM.

B86001Y CHECKS FOR A PREDEFINED FIXED-POINT TYPE OTHER THAN DURATION.

B91001H CHECKS FOR SUPPORT OF ADDRESS CLAUSES FOR TASK ENTRIES (SEE 2.3).

C96005B CHECKS FOR VALUES OF TYPE DURATION'BASE THAT ARE OUTSIDE THE RANGE OF DURATION. THERE ARE NO SUCH VALUES FOR THIS IMPLEMENTATION.

CD1009C USES A REPRESENTATION CLAUSE SPECIFYING A NON-DEFAULT SIZE FOR A FLOATING-POINT TYPE.

CD2A84A, CD2A84E, CD2A84I..J (2 TESTS), AND CD2A84O USE REPRESENTATION CLAUSES SPECIFYING NON-DEFAULT SIZES FOR ACCESS TYPES.

CD2B15B CHECKS THAT STORAGE_ERROR IS RAISED APPROPRIATELY WHEN A SPECIFIED COLLECTION SIZE IS TOO SMALL TO HOLD A VALUE OF THE DESIGNATED TYPE; THIS IMPLEMENTATION ALLOCATES MORE SPACE THAN WHAT IS SPECIFIED, AS ALLOWED BY AI-00558/04.

BD8001A, BD8003A, BD8004A..B (2 TESTS), AND AD8011A USE MACHINE CODE INSERTIONS.

THE TESTS LISTED IN THE FOLLOWING TABLE ARE NOT APPLICABLE BECAUSE THE GIVEN FILE OPERATIONS ARE SUPPORTED FOR THE GIVEN COMBINATION OF MODE AND FILE ACCESS METHOD.

TEST	FILE OPERATION	MODE	FILE ACCESS METHOD
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO

CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102F	RESET	ANY_MODE	TEXT_IO
CE3102G	DELETE	-----	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

THE TESTS LISTED IN THE FOLLOWING TABLE ARE NOT APPLICABLE BECAUSE THE GIVEN FILE OPERATIONS ARE NOT SUPPORTED FOR THE GIVEN COMBINATION OF MODE AND FILE ACCESS METHOD.

TEST	FILE OPERATION	MODE	FILE ACCESS METHOD
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

CE2203A CHECKS THAT WRITE RAISES USE_ERROR IF THE CAPACITY OF THE EXTERNAL FILE IS EXCEEDED FOR SEQUENTIAL_IO. THIS IMPLEMENTATION DOES NOT RESTRICT FILE CAPACITY.

CE2403A CHECKS THAT WRITE RAISES USE_ERROR IF THE CAPACITY OF THE EXTERNAL FILE IS EXCEEDED FOR DIRECT_IO. THIS IMPLEMENTATION DOES NOT RESTRICT FILE CAPACITY.

CE2107B & E (2 TESTS), CE2110B, AND CE2111D ATTEMPT TO ASSOCIATE MULTIPLE INTERNAL SEQUENTIAL FILES WITH THE SAME EXTERNAL FILE WHEN ONE OR MORE FILES IS OPEN FOR WRITING. THE PROPER EXCEPTION IS RAISED WHEN THIS ASSOCIATION IS ATTEMPTED.

CE2107F & G (2 TESTS), CE2110D, AND CE2111H ATTEMPT TO ASSOCIATE MULTIPLE INTERNAL DIRECT FILES WITH THE SAME EXTERNAL FILE WHEN ONE OR MORE FILES IS OPEN FOR WRITING. THE PROPER EXCEPTION IS RAISED WHEN THIS ASSOCIATION IS ATTEMPTED.

CE3111B, CE3111D, CE3114B, AND CE3115A ATTEMPT TO ASSOCIATE MULTIPLE INTERNAL TEXT FILES WITH THE SAME EXTERNAL FILE WHEN ONE OR MORE FILES IS OPEN FOR WRITING. THE PROPER EXCEPTION IS RAISED WHEN THIS ASSOCIATION IS ATTEMPTED.

CE2107C..D (2 TESTS), CE2107H, CE2107L, AND CE3111E APPLY FUNCTION NAME TO TEMPORARY SEQUENTIAL, DIRECT, AND TEXT FILES IN AN ATTEMPT TO ASSOCIATE MULTIPLE INTERNAL FILES WITH THE SAME EXTERNAL FILE; USE_ERROR IS RAISED BECAUSE TEMPORARY FILES HAVE NO NAME.

CE2108B & D (2 TESTS) AND CE3112B USE THE NAMES OF TEMPORARY SEQUENTIAL, DIRECT, AND TEXT FILES THAT WERE CREATED IN OTHER TESTS IN ORDER TO CHECK THAT THE TEMPORARY FILES ARE NOT ACCESSIBLE AFTER THE COMPLETION OF THOSE TESTS; FOR THIS IMPLEMENTATION, TEMPORARY FILES HAVE NO NAME.

CE2111C TEST RESET THE MODE FROM IN_FILE TO INOUT_FILE OR OUT_FILE (AN AMPLIFICATION IN ACCESSING PRIVILEGES WHILE THE XTERNAL FILE IS BEING ACCESSED). THE PROPER EXCEPTION IS RAISED.

EE2401D CHECKS WHETHER READ, WRITE, SET_INDEX, INDEX, SIZE, AND END_OF_FILE ARE SUPPORTED FOR DIRECT FILES FOR AN UNCONSTRAINED ARRAY TYPE. USE_ERROR WAS RAISED FOR DIRECT CREATE. THIS IMPLEMENTATION DOES NOT ALLOW THE INSTANTIATION OF DIRECT_IO WITH UNCONSTRAINED ARRAY TYPES UNLESS A MAXIMUM ELEMENT SIZE IS SPECIFIED IN THE FORM PARAMETER OF THE CREATE PROCEDURE.

EE2401G AND CE2401H TEST FOR INSTANTIATION OF DIRECT_IO WITH UNCONSTRAINED RECORD TYPES. THIS IMPLEMENTATION NOT ALLOW INSTANTIATION OF DIRECT_IO WITH UNCONSTRAINED RECORD TYPES UNLESS A MAXIMUM ELEMENT SIZE IS SPECIFIED IN THE FORM PARAMETER OF THE CREATE PROCEDURE.

CE3304A CHECKS THAT USE_ERROR IS RAISED IF A CALL TO SET_LINE_LENGTH OR SET_PAGE_LENGTH SPECIFIES A VALUE THAT IS INAPPROPRIATE FOR THE EXTERNAL FILE. THIS IMPLEMENTATION DOES NOT HAVE INAPPROPRIATE VALUES FOR EITHER LINE LENGTH OR PAGE LENGTH.

CE3413B CHECKS THAT PAGE RAISES LAYOUT_ERROR WHEN THE VALUE OF THE PAGE NUMBER EXCEEDS COUNT'LAST. FOR THIS IMPLEMENTATION, THE VALUE OF COUNT'LAST IS GREATER THAN 150000 MAKING THE CHECKING OF THIS OBJECTIVE IMPRACTICAL.

2.3 TEST MODIFICATIONS

MODIFICATIONS (SEE SECTION 1.3) WERE REQUIRED FOR NO TESTS.

B91001H was graded inapplicable by Evaluation Modification as directed by the AVO. This implementation does not support address clauses for entries.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

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

For a point of contact for technical information about this Ada implementation system, see:

Attn: Pat Bernard
Ada Product Manager
Digital Equipment Corporation
110 Spit Brook Road (ZK02-1/M11)
Nashua, NH 03062
(603) 881-0247

For a point of contact for sales information about this Ada implementation system, see:

Attn: Pat Bernard
Ada Product Manager
Digital Equipment Corporation
110 Spit Brook Road (ZK02-1/M11)
Nashua, NH 03062
(603) 881-0247

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

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

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

a) Total Number of Applicable Tests	3986
-------------------------------------	------

b) Total Number of Withdrawn Tests	81	
c) Processed Inapplicable Tests	103	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	103	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

All I/O tests of the test suite were processed because this implementation supports a file system. All floating-point precision tests were processed because this implementation supports floating-point precision to the extent that was tested. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 103 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

The communication process between the VAX 8800 and the MicroVAX was as follows:

1. VMS is brought up on the MicroVAX II.
2. The executable files are copied from the VAX 8800 to the MicroVAX II using the network.
3. A VAXELN system is then built on the VAX 8800 and copied to a TK50 tape. This tape is then used to boot VAXELN on the MicroVAX II.
4. The tests are executed under VAXELN and the results captured on a disk attached to the MicroVAX II.
5. VMS is again brought up on the MicroVAX II so the result files can be copied back to the VAX 8800 using the network.

After the test files were loaded onto the host computer, the full

set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link described above, and run. The results were captured on the host computer system.

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

```
/NOANALYSIS_DATA  
/CHECK  
/COPY_SOURCE  
/NODEBUG  
/NODESIGN  
/NODIAGNOSTICS  
/ERROR_LIMIT=1000  
/LIBRARY=ADA$LIB  
/LIST  
/LOAD  
/NOMACHINE_CODE  
/NOTE_SOURCE  
/OPTIMIZE  
/NOSHOW  
/NOSYNTAX_ONLY  
/WARNINGS=default
```

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

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is 255 the value for \$MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$MAX_IN_LEN	255
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	''' & (1..V/2 => 'A') & '''
\$BIG_STRING2	''' & (1..V-1-V/2 => 'A') & '1' & '''
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	''' & (1..V-2 => 'A') & '''

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
ACC_SIZE	: 32
ALIGNMENT	: 4
COUNT_LAST	: 2_147_483_647
DEFAULT_MEM_SIZE	: 2**31-1
DEFAULT_STOR_UNIT	: 8
DEFAULT_SYS_NAME	: VAX_VMS
DELTA_DOC	: 2.0**(-31)
ENTRY_ADDRESS	: 16#40#
ENTRY_ADDRESS1	: 16#80#
ENTRY_ADDRESS2	: 16#100#
FIELD_LAST	: 2_147_483_647
FILE_TERMINATOR	: ' '
FIXED_NAME	: NO_SUCH_FIXED_TYPE
FLOAT_NAME	: LONG_LONG_FLOAT
FORM_STRING	: ""
FORM_STRING2	:
"CANNOT RESTRICT FILE CAPACITY"	
GREATER_THAN_DURATION	: 75_000.0
GREATER_THAN_DURATION_BASE_LAST	: 131_073.0
GREATER_THAN_FLOAT_BASE_LAST	: 1.80141E+38
GREATER_THAN_FLOAT_SAFE_LARGE	: 1.7014117E+38
GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	: 1.0E308
HIGH_PRIORITY	: 15
ILLEGAL_EXTERNAL_FILE_NAME1	: BADCHAR^@.~!
ILLEGAL_EXTERNAL_FILE_NAME2	:
THIS-FILE-NAME-WOULD-BE-PERFECTLY-LEGAL-IF-IT-WERE-NOT-SO-LONG.SO_THERE	
INAPPROPRIATE_LINE_LENGTH	: 65_536
INAPPROPRIATE_PAGE_LENGTH	: -1
INCLUDE_PRAGMA1	:
PRAGMA INCLUDE ("A28006D1.TST")	
INCLUDE_PRAGMA2	:
PRAGMA INCLUDE ("B28006E1.TST")	
INTEGER_FIRST	: -2147483648
INTEGER_LAST	: 2147483647
INTEGER_LAST_PLUS_1	: 2_147_483_648
INTERFACE_LANGUAGE	: FORTRAN
LESS_THAN_DURATION	: -75_000.0
LESS_THAN_DURATION_BASE_FIRST	: -131_073.0
LINE_TERMINATOR	: ASCII.LF
LOW_PRIORITY	: 0
MACHINE_CODE_STATEMENT	: NULL;
MACHINE_CODE_TYPE	: NO_SUCH_TYPE
MANTISSA_DOC	: 31
MAX_DIGITS	: 33

MAX_INT	: 2147483647
MAX_INT_PLUS_1	: 2_147_483_648
MIN_INT	: -2147483648
NAME	: SHORT_SHORT_INTEGER
NAME_LIST	: VAX_VMS, VAXELN
NAME_SPECIFICATION1	:
ACVC_LFN_DEVICE:[ACVC_LFN_AREA]X2120A.DAT;1	
NAME_SPECIFICATION2	:
ACVC_LFN_DEVICE:[ACVC_LFN_AREA]X2120B.DAT;1	
NAME_SPECIFICATION3	:
ACVC_LFN_DEVICE:[ACVC_LFN_AREA]X3119A.DAT;1	
NEG_BASED_INT	: 16#FFFFFFFE#
NEW_MEM_SIZE	: 1_048_576
NEW_STOR_UNIT	: 8
NEW_SYS_NAME	: VAXELN
PAGE_TERMINATOR	: ASCII.LF & ASCII.FF
RECORD_DEFINITION	: NULL; END RECORD;
RECORD_NAME	: NO_SUCH_MACHINE_CODE_TYPE
TASK_SIZE	: 32
TASK_STORAGE_SIZE	: 0
TICK	: 10.0**(-2)
VARIABLE_ADDRESS	: 16#1000#
VARIABLE_ADDRESS1	: 16#1004#
VARIABLE_ADDRESS2	: 16#1008#
YOUR_PRAGMA	: EXPORT_OBJECT

APPENDIX B

COMPILATION SYSTEM OPTIONS

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

. /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.

. /DESIGN or /NODESIGN

Controls whether the input file is processed as a design or compiled as an Ada source. The default is /NODESIGN, in which case the file is compiled.

. /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).

./LOAD or /NOLOAD

Controls whether the current program library is updated with successfully processed units contained in the source file. The default is /LOAD.

. /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 (re)compilation 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.)

- . /SYNTAX or /NOSYNTAX_ONLY

Controls whether a syntax check only is performed. The default is /NOSYNTAX_ONLY, which indicates that full processing is done.

- . /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)

LINKER OPTIONS

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

ACS

LINK

Command_Qualifiers

/AFTER=time

Requests that the batch job be held until after a specific time, when the ACS LINK command is executed in batch mode (LINK/SUBMIT).

/BATCH_LOG=file-spec

Provides a file specification for the batch log file when the LINK command is executed in batch mode (LINK/SUBMIT).

/BRIEF

Directs the linker to produce a brief image map file.

/COMMAND[=file-spec]

If you specify the /COMMAND qualifier, the program library manager does not invoke the linker, and the generated command file is saved for you to invoke or submit as a batch job.

/CROSS_REFERENCE

/NOCROSS_REFERENCE (D)

Controls whether the image map file contains a symbol cross-reference.

/DEBUG[=file-spec]

/NODEBUG (D)

Controls whether a debugger symbol table is included in the executable image file.

/EXECUTABLE[=file-spec] (D)

/NOEXECUTABLE

Controls whether the linker creates an executable image file and

optionally provides a file specification for the file.

/FULL

Directs the linker to produce a full image map file, which is the most complete image map.

/KEEP (D)
/NOKEEP

This is a command qualifier.

Controls whether the batch log file generated is deleted after it is printed when the ACS LINK command is executed in batch mode (LINK/SUBMIT).

/LOG
/NOLOG (D)

Controls whether a list of all the units included in the executable image is displayed.

/MAIN (D)
/NOMAIN

Controls where the image transfer address is to be found.

/MAP[=file-spec]
/NOMAP (D)

Controls whether the linker creates an image map file and optionally provides a file specification for the file.

/NAME=job-name

Specifies a string to be used as the job name and as the file name for the batch log file when the ACS LINK command is executed in batch mode (LINK/SUBMIT).

/NOTIFY (D)
/NONOTIFY

Controls whether a message is broadcast when the ACS LINK command is executed in batch mode (LINK/SUBMIT). The message is broadcast to any terminal at which you are logged in, notifying you that your job has been completed or terminated.

/OBJECT=file-spec

Provides a file specification for the object file generated by the ACS LINK command.

`/OUTPUT=file-spec`

Requests that any ACS output generated before the linker is invoked be written to the file specified rather than to `SYSS$OUTPUT`.

`/PRINTER[=queue-name]`
`/NOPRINTER (D)`

Controls whether the log file is queued for printing when the `LINK` command is executed in batch mode (`LINK/SUBMIT`) and the batch job is completed.

`/QUEUE=queue-name`

Specifies the batch job queue in which the job is entered when the ACS `LINK` command is executed in batch mode (`LINK/SUBMIT`).

`/SUBMIT`

Directs the program library manager to submit the command file generated for the linker to a batch queue.

`/SYSLIB (D)`
`/NOSYSLIB`

Controls whether the linker automatically searches the default system library for unresolved references.

`/SYSSHR (D)`
`/NOSYSSHR`

Controls whether the linker automatically searches the default system shareable image library `SYSS$LIBRARY:IMAGELIB.OLB` for unresolved references.

`/SYSTEM_NAME=system`

Directs the program library manager to produce an image for execution on a particular operating system.

The possible system values are `VAX_VMS` and `VAXELN`.

`/TRACEBACK (D)`
`/NOTRACEBACK`

Controls whether the linker includes traceback information in the executable image file for run-time error reporting.

`/USERLIBRARY[=(table[,...])]`

/NOUSERLIBRARY

Controls whether the linker searches any user-defined default libraries after it has searched any specified user libraries.

/WAIT

Directs the program library manager to execute the command file generated for the linker in a subprocess.

Parameter_Qualifiers

/INCLUDE=(object-file,...)

Indicates that the associated input file is a VMS object module library or shareable image library with a default file type of .OLB, and that the named elements from that library should be linked with the main program named in the ACS LINK command.

/LIBRARY

Indicates that the associated input file is a VMS object module library or shareable image library to be searched for modules to resolve any undefined symbols in the input files.

/OPTIONS

Indicates that the associated input file is a VMS linker options file.

/SHAREABLE

Indicates that the associated input file is a VMS shareable image. The default file type is .EXE.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

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

package STANDARD is

```
type INTEGER is range -2147483648..2147483647;
type 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
    -5.9486574767861588254287966331400E+4931..
    5.9486574767861588254287966331400E+4931;
```

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

```
end STANDARD;
```

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 or for an exception declared

in a generic unit. 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 it 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, parameter mechanisms, result mechanism, and a first optional parameter 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, parameter mechanisms, and a first optional parameter 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, parameter mechanisms, and a first optional parameter 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).

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).

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 that is an instance of a generic 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

body 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 that is an instance of a generic 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 the 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, 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).

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`. 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).
- SHARE_GENERIC (see 12.1b).
- 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);
    for NAME use (1, 2);

    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 .. 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 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 objects (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 VMS Record Management Services (RMS) file organizations and facilities. To give users the maximum benefit of the underlying VMS 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 VMS 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:

<code>SHORT_SHORT_INTEGER</code>	-128 .. 127
<code>SHORT_INTEGER</code>	-32768 .. 32767
<code>INTEGER</code>	-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:

<code>COUNT</code>	0 .. 2147483647
<code>POSITIVE_COUNT</code>	1 .. 2147483647

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

<code>FIELD</code>	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	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_FF8#e+21
approximately	1.93428E+25
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_OVERFLOW	True

Attribute	D_floating value and approximate decimal equivalent (where applicable)
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

Attribute	D_floating value and approximate decimal equivalent (where applicable)
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
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 (where applicable)
DIGITS	15
MANTISSA	51
EMAX	204
EPSILON	16#0.4000_0000_0000_00#e-12
approximately	8.881784197001E-16
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

Attribute	G_floating value and approximate decimal equivalent (where applicable)
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_0#e-27
approximately	7.703719777548943412223911770339 7E-34
SMALL	16#0.8000_0000_0000_0000_0000_0000_0#e-111
approximately	1.1006568214637918210934318020936E-134
LARGE	16#0.FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFE_0#e+111
approximately	4.5427420268475430659332737993000E+133
SAFE_EMAX	16383
SAFE_SMALL	16#0.1000_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
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

Attribute	H_floating value and approximate decimal equivalent (where applicable)
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	1.3107199993896484375E+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
255	Maximum identifier length (number of characters)
255	Maximum number of characters in a source line
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
4095	Maximum number of library units and subunits in a compilation closure ¹
16383	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
65534	Maximum number of lines in a source file
2 ³¹ - 1	Maximum number of bits in any object

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

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