AD-A242 268
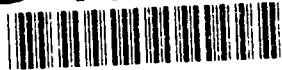
JMENTATION PAGE

Form Approved
OPM No. 0704-0188

d to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
its regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 17 May 1991 to

4. TITLE AND SUBTITLE

Ada Compiler Validation Summary Report: U.S. NAVY, AdaVAX, Version 5.0, (/NO OPTIMIZE) VAX 8350 (Host & Target), 910517S1.11163

5. FUNDING NUMBERS

6. AUTHOR(S)

National Institute of Standards and Technology
Gaithersburg, MD
USA

DTIC

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

NIST90USN510_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

No software available for distribution per Hatch, Michelle Kee, ADA 11/4/91 telecon mgn

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

U.S. NAVY, AdaVAX, Version 5.0, Gaithersburg, MD, (/NO OPTIMIZE) VAX 8350 (Host & Target), ACVC 1.11

91-15061

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	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

AVF Control Number: NIST90USN510_2_1.11
DATE COMPLETED
    BEFORE ON-SITE: 1991-04-05
    AFTER ON-SITE: 1991-05-17
    REVISIONS: 1991-07-24

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910517S1.11163
U.S. NAVY
AdaVAX, Version 5.0 (/NO_OPTIMIZE)
VAX 8350 => VAX 8350

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

A-1

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Customer: U.S. NAVY

Certificate Awardee: U.S. NAVY

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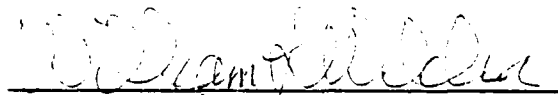   Gaithersburg, Maryland 20899

ACVC Version: 1.11

Ada Implementation:

   Compiler Name and Version: AdaVAX, Version 5.0 (/NO_OPTIMIZE)

   Host Computer System: VAX 8350, running VAX/VMS Version
                               5.3

   Target Computer System: VAX 8350, running VAX/VMS Version
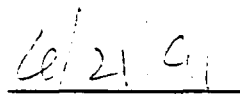   5.3

Declaration:

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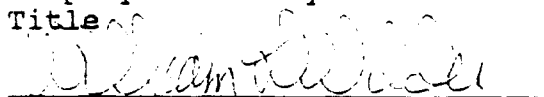8652-1987 in the implementation listed above.

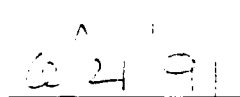_____   _____
Customer Signature   Date
Company U.S. Navy
Title

_____   _____
Certificate Awardee Signature   Date
Company U.S. Navy
Title

Certificate Information

The following Ada implementation was tested and determined to pass
ACVC 1.11.   Testing was completed on 1991-05-17.

   Compiler Name and Version: AdaVAX, Version 5.0 (/NO_OPTIMIZE)

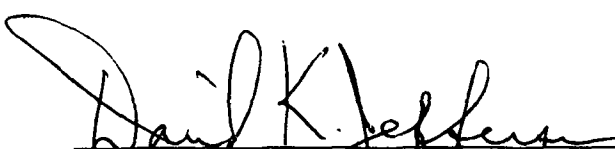   Host Computer System: VAX 8350, running VAX/VMS Version
                                    5.3

   Target Computer System: VAX 8350, running VAX/VMS Version
                                    5.3

A more detailed description of this Ada implementation is found in
section 3.1 of this report.
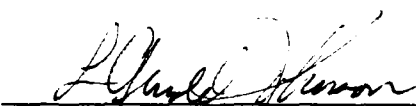
As a result of this validation effort, Validation Certificate
910517S1.11163 is awarded to U.S. NAVY. This certificate expires
on 01 March 1993.

This report has been reviewed and is approved.

_____
Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
 Engineering Division (ISED)

_____
Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
  Validation Group

Computer Systems Laboratory (CLS)
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

TABLE OF CONTENTS

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

1.2 REFERENCES

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

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint
Program Office, August 1990.

[UG89] <u>Ada Compiler Validation Capability User's Guide</u>, 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.

1-3

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 94 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 91-05-03.

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

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are
irrelevant for a given Ada implementation. 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.

The following 285 tests have floating-point type declarations
requiring more digits than SYSTEM.MAX_DIGITS:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)

```
C45241F..Y (20 tests)        C45321F..Y (20 tests)
C45421F..Y (20 tests)        C45521F..Z (21 tests)
C45524F..Z (21 tests)        C45621F..Z (21 tests)
C45641F..Y (20 tests)        C46012F..Z (21 tests)
```

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

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

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

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

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

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, there is no such type.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types; for this implementation, MACHINE_OVERFLOWS is TRUE.

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

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION; for this implementation, there are no such values.

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

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

AE2101C and EE2201D..E (2 tests) use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with

discriminants without defaults; these instantiations are rejected by this compiler.

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

The tests listed in the following table 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

CE2107B..D (3 tests), CE2110B, and CE2111D check operations on sequential files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107E and CE2107L check operations on direct and sequential files when files of both kinds are associated with the same external file; USE_ERROR is raised when this association is attempted.

CE2107G..H (2 tests), CE2110D, and CE2111H check operations on direct files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

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

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

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

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 41 tests.

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

B22003A	B22004A	B23004A	B24005A	B24005B	B28003A
B33201C	B33202C	B33203C	B33301B	B37106A	B37301I
B38003A	B38003B	B38009A	B38009B	B44001A	B44004A
B54A01L	B55A01A	B61005A	B85008G	B85008H	B95063A
B97103E	BB1006B	BC1102A	BC1109A	BC1109B	BC1109C
BC1109D	BC1201F	BC1201G	BC1201H	BC1201I	BC1201J
BC1201L	BC3013A	BE2210A	BE2413A		

"PRAGMA ELABORATE (REPORT)" has been added at appropriate points in order to solve the elaboration problems for:

C83030C   C86007A

C34005P and C34005S were graded passed by Test Modification as directed by the AVO. These tests contain expressions of the form "I - X'FIRST + Y'FIRST", where X and Y are of an array type with a lower bound of INTEGER'FIRST; this implementation recognizes that "X'FIRST + Y'FIRST" is a loop invariant and so evaluates this part of the expression separately, which raises NUMERIC_ERROR. These

2-4

tests were modified by inserting parens to force a different order of evaluation (i.p., to force the subtraction to be evaluated first) at lines 187 and 262/263, respectively; those modified lines are:

[C34005P, line 187]

```
IF NOT EQUAL (X (I), Y ((I - X'FIRST) + Y'FIRST)) THEN
```

[C34005S, lines 261..4 (only 262 & 263 were modified)]

```
IF NOT EQUAL (X (I, J),
             Y ((I - X'FIRST) + Y'FIRST,
             (J - X'FIRST(2)) +
                 Y'FIRST(2))) THEN
```

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:

Mr. Christopher T. Geyer
Fleet Combat Directions Systems Support Activity
Code 81, Room 301D
200 Catalina Blvd.
San Diego, California 92147
619-553-9447

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

NOT APPLICABLE FOR THIS IMPLEMENTATION

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   3695

    b) Total Number of Withdrawn Tests   94
    c) Processed Inapplicable Tests   381
    d) Non-Processed I/O Tests   0
    e) Non-Processed Floating-Point
            Precision Tests   0

f) Total Number of Inapplicable Tests   381   (c+d+e)
g) Total Number of Tests for ACVC 1.11   4170 (a+b+f)

When this implementation 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 381 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. In addition, the modified tests mentioned in section 2.3 were also processed.

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.

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

The tests were compiled, linked, and executed on the host/ target 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:

FOR /NO_OPTIMIZE the options were:

/SUMMARY /NO_TRACE_BACK /NO_OPTIMIZE /SOURCE
/OUT=<filename>

FOR /OPTIMIZE the options were:

/SUMMARY /NO_TRACE_BACK /OPTIMIZE   /SOURCE
/OUT=<filename>

The options invoked by default for validation testing during this test were:

FOR /NO_OPTIMIZE the options were:

```
/NO_MACHINE_CODE        /NO_ATTRIBUTE        /NO_CROSS_REFERENCE
/NO_DIAGNOSTICS         /NO_NOTES            /PRIVATE        /LIST
/CONTAINER_GENERATION   /CODE_ON_WARNING    /NO_MEASURE  /DEBUG
/CHECKS
```

FOR /OPTIMIZE the options were:

```
/NO_MACHINE_CODE        /NO_ATTRIBUTE        /NO_CROSS_REFERENCE
/NO_DIAGNOSTICS         /NO_NOTES            /PRIVATE        /LIST
/CONTAINER_GENERATION   /CODE_ON_WARNING    /NO_MEASURE  /DEBUG
/CHECKS
```

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 | 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 ·	120
$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	1073741823
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	ADAVAX
$DELTA_DOC	0.000_000_000_465_661_287_307_ 739_257_812_5
$ENTRY_ADDRESS	16#40#
$ENTRY_ADDRESS1	16#80#
$ENTRY_ADDRESS2	16#100#
$FIELD_LAST	32_767
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE_AVAILABLE
$FLOAT_NAME	NO_SUCH_TYPE_AVAILABLE
$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	1.0E303
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
$HIGH_PRIORITY	15

A-2

```
$ILLEGAL_EXTERNAL_FILE_NAME1 BADCHAR^@.~!

$ I L L E G A L _ E X T E R N A L _ F I L E _ N A M E 2
MUCH_TOO_LONG_NAME_FOR_A_FILE_UNDER_VMS_SO_THE_SO_THERE

$INAPPROPRIATE_LINE_LENGTH    256

$INAPPROPRIATE_PAGE_LENGTH    -1

$INCLUDE_PRAGMA1              PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2              PRAGMA INCLUDE ("B28006F1.TST")

$INTEGER_FIRST               -32768

$INTEGER_LAST                32767

$INTEGER_LAST_PLUS_1         32768

$INTERFACE_LANGUAGE          ASMVAX_JSB

$LESS_THAN_DURATION          -75000.0

$LESS_THAN_DURATION_BASE_FIRST    -131073.0

$LINE_TERMINATOR             ' '

$LOW_PRIORITY                1

$MACHINE_CODE_STATEMENT      BYTE_OP_CODE'(OP=>NOP);

$MACHINE_CODE_TYPE           BYTE

$MANTISSA_DCC                31

$MAX_DIGITS                  9

$MAX_INT                     2147483647

$MAX_INT_PLUS_1              2147483648

$MIN_INT                     -2147483648

$NAME                        NO_SUCH_TYPE_AVAILABLE

$NAME_LIST                   ADAVAX, ADA_L, ADA_M

$NAME_SPECIFICATION1
     ALSN$TEST:[ALSN_TESTS.ACVC.TESTACVCVAX.RUNNING]X2120A.;1
```

```
$NAME_SPECIFICATION2
     ALSN$TEST:[ALSN_TESTS.ACVC.TESTACVCVAX.RUNNING]X2120B.;1

$NAME_SPECIFICATION3
     ALSN$TEST:[ALSN_TESTS.ACVC.TESTACVCVAAX.RUNNING]X3119A.;1

$NEG_BASED_INT                    16#FFFFFFFE#

$NEW_MEM_SIZE                     1073741823

$NEW_STOR_UNIT                    8

$NEW_SYS_NAME                     ADA_L

$PAGE_TERMINATOR                  ASCII.FF

$RECORD_DEFINITION                RECORD    LWORD_1:LONG_WORD;
                                  LWORD_2:LONG_WORD; END RECORD;

$RECORD_NAME                      QUADWORD

$TASK_SIZE                        1624

$TASK_STORAGE_SIZE                1024

$TICK                            0.01

$VARIABLE_ADDRESS                 16#0020#

$VARIABLE_ADDRESS1                16#0024#

$VARIABLE_ADDRESS2                16#0028#

$YOUR_PRAGMA                      TITLE ("THIS IS AN ALS/N ACVC
                                  TITLE")
```

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.

Section 9

Compiler Options

+---+
| Option   Function |
+---+
| |
| MEASURE   Generates code to monitor execution |
| frequency at the subprogram level for |
| the current unit.   Default: NO_MEASURE |
| |
| NO_CHECKS   NO_CHECKS suppresses all run-time |
| error checking.   CHECKS provides |
| run-time error checking. |
| Default: CHECKS |
| |
| NO_CODE_ON_WARNING |
| NO_CODE_ON_WARNING means no code is |
| generated when there is a diagnostic |
| of severity WARNING or higher. |
| CODE_ON_WARNING generates code |
| only if there are no diagnostics |
| of a severity higher than WARNING. |
| Default: CODE_ON_WARNING |
| |
| NO_CONTAINER_GENERATION |
| NO_CONTAINER_GENERATION means that no |
| container is produced even if there |
| are no diagnostics. |
| CONTAINER_GENERATION produces a |
| container if diagnostic serverity |
| permits. |
| Default: CONTAINER_GENERATION |
| |
+---+

Table 9-1a - Special Processing Options

Option	Function
NO_DEBUG	If NO_DEBUG is specified, only that information needed to link, export and execute the current unit is included in the compiler output.
	With the DEBUG option in effect, internal representations and additional symbolic information are stored in the container. Default: DEBUG
NO_TRACE_BACK	Disables the location of source exceptions that are not handled by built-in exception handlers. Default: TRACE_BACK
OPTIMIZE	Enables global optimizations in accordance with the optimization pragmas specified in the source program. If the pragma OPTIMIZE is not included, the optimizations emphasize TIME over SPACE. When NO_OPTIMIZE is in effect, no global optimizations are performed, regardless of the pragmas specified. Default: NO_OPTIMIZE

Table 9-1b - Special Processing Options (Continued)

Option	Function
ATTRIBUTE	Produces a Symbol Attribute Listing. (Produces an attribute cross-reference listing when both ATTRIBUTE and CROSS_REFERENCE are specified.) Default: NO_ATTRIBUTE
CROSS_REFERENCE	Produces a Cross-Reference Listing. (Produces an attribute cross-reference listing when both ATTRIBUTE and CROSS_REFERENCE are specified.) Default: NO_CROSS_REFERENCE
DIAGNOSTICS	Produces a Diagnostic Summary Listing. Default: NO_DIAGNOSTICS
MACHINE_CODE	Produces a machine code listing if code is generated. Code is generated when CONTAINER_GENERATION option is in effect and (1) there are no diagnostics of severity ERROR, SYSTEM or FATAL, and/or (2) NO_CODE_ON_WARNING option is in effect and there are no diagnostics of severity higher than NOTE. Default: NO_MACHINE_CODE
NOTES	Includes diagnostics of NOTE severity level in the Source Listing. Default: NO_NOTES
NO_PRIVATE	Excludes listing of Ada statements in private part if a Source Listing is produced. Default: PRIVATE
SOURCE	Produce listing of Ada source statements. Default: NO_SOURCE
SUMMARY	Produce a Summary Listing; always produced when there are errors in the compilation. Default: NO_SUMMARY

Table 9-2 - Listing Control Options

```
+-------------------------------------------------------------------+
|   Option            Function                                      |
+-------------------------------------------------------------------+
|                                                                   |
|   MSG               Sends error messages and the                 |
|                     Diagnostic Summary Listing to the            |
|                     file specified.  The default is to           |
|                     send error messages and the Diagnostic       |
|                     Summary Listing to Message Output            |
|                     (usually the terminal).                      |
|                                                                   |
|   OUT               Sends all selected listings to the           |
|                     single file specified.  The default          |
|                     is to send listings to Standard              |
|                     Output (usually the terminal).               |
|                                                                   |
+-------------------------------------------------------------------+
```

Table 9-3 - Control_Part (Redirection) Options

LINKER OPTIONS

The linker options c 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.

Section 11

Linker Options

Option	Function
DEBUG	Produces a linked_container to be debugged. Default: NO_DEBUG.
MEASURE	Produces a linked_container to be analyzed. Default: NO_MEASURE
NO_SEARCH	Limits the contents of the linked container to those units explicitly specified in the UNITLIST. Default: SEARCH.
PARTIAL	Produces an incomplete linked_container with unresolved references. Default: NO_PARTIAL.

Table 11-1 - LNKVAX Linker Special Processing Options

```
+----------------------------------------------------------------------+
| Option              Function                                         |
+----------------------------------------------------------------------+
|                                                                      |
|   no option         Linker Summary listing, always produced         |
|                     unless diagnostics prevent its generation.      |
|                                                                      |
|   ELAB_LIST         Generates an elaboration order listing.         |
|                     Default: NO_ELAB_LIST.                          |
|                                                                      |
|   SYMBOLS           Produces a Linker symbols listing.              |
|                     Default: NO_SYMBOLS.                            |
|                                                                      |
|   UNITS             Produces a Linker units listing.               |
|                     Default: NO_UNITS.                              |
|                                                                      |
+----------------------------------------------------------------------+
```

Table 11-2 - LNKVAX Linker Listing Options

```
+----------------------------------------------------------------------+
|  Option              Function                                        |
+----------------------------------------------------------------------+
|                                                                      |
|   MSG               Sends error messages to the file                |
|                     specified.  The default is to send              |
|                     error messages to Message Output                |
|                     (usually the terminal).                         |
|                                                                      |
|   OUT               Sends all selected listings to the              |
|                     single file specified.  The default             |
|                     is to send listings to Standard                 |
|                     Output (usually the terminal).                  |
|                                                                      |
+----------------------------------------------------------------------+
```

Table 11-3 - Control_Part (Redirection) Options

Section 12

Exporter Options

```
+--------------------------------------------------------+
| Option             Function                            |
+--------------------------------------------------------+
|                                                        |
|   ACCOUNTING       Causes the amount of CPU time and   |
|                    wall clock time used by the program |
|                    to be reported at program termination|
|                    to message output.                  |
|                    Default: NO_ACCOUNTING              |
|                                                        |
|   DEBUG            Produces a load module that can be   |
|                    debugged by the ALS/N Symbolic      |
|                    Debugger.   Default: NO_DEBUG       |
|                                                        |
|   DEBUG_SYMBOLS    Produces a file of external symbols  |
|                    suitable for input to the VAX/VMS    |
|                    Debugger.   Default: NO_DEBUG_SYMBOLS|
|                                                        |
|   MEASURE          Produces a load module that includes |
|                    the invocation of frequency and     |
|                    statistical analyzer.               |
|                    Default: NO_MEASURE                 |
|                                                        |
+--------------------------------------------------------+
```

Table 12-1 - Special Processing Options

```
+-----------------------------------------------------------------+
| Option              Function                                    |
+-----------------------------------------------------------------+
|                                                                 |
|  no option          Exporter Summary Listing is always         |
|                     produced unless diagnostics prevent        |
|                     its generation.                            |
|                                                                 |
|  MAP                Produces a program sections map            |
|                     listing that summarizes the               |
|                     executable image.  Default: NO_MAP        |
|                                                                 |
|  SYMBOLS            Produces a list of external symbol         |
|                     descriptor information for external        |
|                     definitions contained in the object       |
|                     module.  Default: NO_SYMBOLS              |
|                                                                 |
+-----------------------------------------------------------------+
```

Table 12-2 - Listing Options

```
+-----------------------------------------------------------------+
| Option              Function                                    |
+-----------------------------------------------------------------+
|                                                                 |
|  MSG                Sends error messages to the file          |
|                     specified.  The default is to send        |
|                     error messages to Message Output          |
|                     (usually the terminal).                   |
|                                                                 |
|  OUT                Sends all selected listings to the        |
|                     single file specified.  The default       |
|                     is to send listings to Standard           |
|                     Output (usually the terminal).            |
|                                                                 |
+-----------------------------------------------------------------+
```

Table 12-3 - Control_Part (Redirection) Options

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 -32_768 .. 32_767;
   type LONG_INTEGER is range -2_147_483_648 .. 2_147_483_647;
   type FLOAT is digits 6 range
       -(2#0.1111_1111_1111_1111_1111_1#E127) ..
       (2#0.1111_1111_1111_1111_1111_1#E127);
   type LONG_FLOAT is digits 9 range
       -(2#0.1111_1111_1111_1111_1111_1111_1111_111#E127) ..
       (2#0.1111_1111_1111_1111_1111_1111_1111_111#E127);
   type DURATION is delta 2.0 ** (-14) range
       -131_072.0 .. 131_072.0 - 2.0 ** (-14);

end STANDARD;
```

C-1

Appendix F

The Ada Language for the VAX Target

The source language accepted by the compiler is Ada, as
described in the Military Standard, Ada Programming Language,
ANSI/MIL-STD-1815A-1983, 17 February 1983 ("Ada Language
Reference Manual").

The Ada definition permits certain implementation
dependencies. Each Ada implementation is required to supply a
complete description of its dependencies, to be thought of as
Appendix F to the Ada Language Reference Manual. This section is
that description for the VAX/VMS target.

F.1 Options

There are several compiler options provided by all ALS/N
Compilers that directly affect the pragmas defined in the Ada
Language Reference Manual. These compiler options currently
include the CHECKS and OPTIMIZE options that affect the SUPPRESS
and OPTIMIZE pragmas, respectively. A complete list of ALS/N
Compiler options can be found in Section 9.

The CHECKS option enables all run-time error checking for the
source file being compiled, which can contain one or more
compilation units. This allows the SUPPRESS pragma to be used in
suppressing the run-time checks discussed in the Ada Language
Reference Manual, but note that the SUPPRESS pragmas must be
applied to each compilation unit. The NO_CHECKS option disables
all run-time error checking for all compilation units within the
source file and is equivalent to SUPPRESSing all run-time checks
within every compilation unit.

The OPTIMIZE option enables all compile-time optimizations
for the source file being compiled, which can contain one or more
compilation units. This allows the OPTIMIZE pragma to request
either TIME-oriented or SPACE-oriented optimizations be
performed, but note that the OPTIMIZE pragma must be applied to
each compilation unit. If the OPTIMIZE pragma is not present,
the ALS/N Compiler's Global Optimizer tends to optimize for TIME
over SPACE. The NO_OPTIMIZE option disables all compile-time
optimizations for all compilation units within the source file
regardless of whether or not the OPTIMIZE pragma is present.

F.2 Pragmas

    Both implementation-defined and Ada language-defined pragmas
are provided by all ALS/N Compilers. The syntax defined in the
Ada Language Reference Manual allows pragmas as the only element
in a compilation, before a compilation unit, at defined places
within a compilation unit, or following a compilation unit. The
ALS/N Compilers associates pragmas with compilation units as
follows:

a. If a pragma appears before any compilation unit in a
    compilation, it will affect all following compilation units,
    as specified below, and in the Ada Language Reference Manual.

b. If a pragma appears inside a compilation unit, it will be
    associated with that compilation unit, and in listings
    associated with that compilation unit as described in the Ada
    Language Reference Manual, or in this document.

c. If a pragma follows a compilation unit, it will be associated
    with the preceding compilation unit, and the effects of the
    pragma will be found in the container of that compilation
    unit, and in listings associated with that container.

    The pragmas MEMORY_SIZE, STORAGE_UNIT, and SYSTEM_NAME are
described in Section 13.7 of the Ada Language Reference Manual.
They may appear only at the start of the first compilation when
creating a new program library. In the ALS/N, however, since
program libraries are created by the Program Library Manager and
not by the compiler, the use of these pragmas is obviated. If
they appear anywhere, a diagnostic of severity level WARNING is
generated.

F.2.1 Language-defined Pragmas

    The following notes specify the language-required definitions
of the predefined pragmas. Unmentioned language-defined pragmas
are implemented as defined by the Ada Language Reference Manual.

pragma INLINE (subprogram_name);

> There are three instances in which the INLINE pragma is
> ignored. Each of these cases produces a warning
> message that states the INLINE did not occur.

> a. If a call to an INLINE subprogram is compiled
> before the actual body of the subprogram has been
> compiled, a routine call is made instead.

> b. If the compilation unit containing the INLINE
> subprogram depends on the compilation unit of its
> caller, a routine call is made instead.

> c. If an immediately recursive subprogram call is made
> within the body of the INLINE subprogram, the
> pragma INLINE is ignored entirely.

pragma INTERFACE (language_name, subprogram_name);

> Two language_names will be recognized and implemented:

> ASMVAX_JSB, and ASMVAX_CALLS.

> The language_name ASMVAX_JSB indicates that a
> subprogram written in the VAX/VMS assembler language
> will be called with a JSB instruction and the
> parameters passed in registers. The language_name
> ASMVAX_CALLS will provide an interface to a VAX
> assembler language subprogram via the CALLS
> instruction, with the parameters passed on the stack,
> with the same parameter passing conventions used for
> calling Ada subprograms.

> The user must ensure that an assembly-language body
> container for this specification exists in the program
> library before linking.

pragma OPTIMIZE (arg);

> This pragma is effective only when the "OPTIMIZE"
> option has been given to the compiler. The argument is
> either TIME or SPACE. If TIME is specified, the
> optimizer concentrates on optimizing code execution
> time. If SPACE is specified, the optimizer
> concentrates on optimizing code size.

pragma PRIORITY (arg)1;

> The PRIORITY argument is an integer static expression
> value of predefined integer subtype PRIORITY. The
> pragma has no effect in a location other than a task
> (type) specification or outermost declarative part of a
> subprogram. If the pragma appears in the declarative
> part of a subprogram, it has no effect unless that
> subprogram is designated as the "main" subprogram at
> link time.

pragma SUPPRESS (arg[,arg]);

> Pragmas to suppress OVERFLOW_CHECK will have no effect
> for operations of integer types.

> A SUPPRESS pragma will have effect only within the
> compilation unit in which it appears, except that a
> SUPPRESS of ELABORATION_CHECK applied at the
> declaration of a subprogram or task unit will apply to
> all calls or activations.

pragma MEMORY_SIZE;

> This pragma is ignored and a WARNING diagnostic is
> issued.

pragma STORAGE_SIZE;

> This pragma is ignored and a WARNING diagnostic is
> issued.

pragma SYSTEM_NAME;

> This pragma is ignored and a WARNING diagnostic is
> issued.

F.2.2 Implementation-defined Pragmas

The following is the only implementation-defined pragma:

pragma TITLE (arg);

> This is a listing control pragma. It takes a
> single argument of type string. The string
> specified will appear on the second line of
> each page of the source listing produced for
> the compilation unit within which it appears.
> The pragma should be the first lexical unit
> to appear within a compilation unit
> (excluding comments). If it is not, a
> warning message is issued.

F.2.3 Scope of Pragmas

The scope of pragmas is as described in the Ada Language
Reference Manual except as noted below:

MEMORY_SI7ⁿ   No scope, but a WARNING diagnostic is generated.

PAGE - .ac scope.

STORAGE_SIZE - No scope, but a WARNING diagnostic is generated.

SYSTEM_NAME - No scope, but a WARNING diagnostic is generated.

TITLE - The compilation unit within which the pragma occurs.

F.3   Attributes

There is one implementation-defined attribute in addition to
the predefined attributes found in Appendix A of the Ada Language
Reference Manual.

X'DISP

> A value of type UNIVERSAL_INTEGER that
> corresponds to the displacement that is used
> to address the first storage unit occupied
> by a data object X at a static offset within
> an implemented activation record.
>
> This attribute differs from the ADDRESS
> attribute in that ADDRESS supplies the
> absolute address while DISP supplies the
> displacement relative to some base value
> (such as a stack frame pointer). It is the
> user's responsibility to determine the base
> value relevant to the attribute.

The following notes augment the language-required definitions
of the predefined attributes found in Appendix A of the Ada
Language Reference Manual.

T'MACHINE_EMAX	is 127.
T'MACHINE_EMIN	is -127.
T'MACHINE_MANTISSA	if the size of the base type T is 32, MACHINE_MANTISSA is 24. if the size of the base type T is 64, MACHINE_MANTISSA is 56.
T'MACHINE_OVERFLOWS	is true.
T'MACHINE_RADIX	is 2.
T'MACHINE_ROUNDS	is false.

F.4   Predefined Language Environment

The predefined Ada language environment consists of the
packages STANDARD and SYSTEM described below.

F.4.1 Package STANDARD

The Package STANDARD contains the following definitions in addition to those specified in Appendix C of the Ada Language Reference Manual:

```
-- For this implementation, there is no corresponding body.

type BOOLEAN is (FALSE,TRUE); for BOOLEAN'SIZE use 1;

-- The universal type UNIVERSAL_INTEGER is predefined for Ada.

type INTEGER is range -32_768 ..  32_767;

type LONG_INTEGER is range -2_147_483_648 ..  2_147_483_647;

-- The universal type UNIVERSAL_REAL is predefined for Ada.

type FLOAT is digits 6 range
   - (2#0.1111_1111_1111_1111_1111_1#E127) ..
     (2#0.1111_1111_1111_1111_1111_1#E127);

type LONG_FLOAT is digits 9 range
   -(2#0.1111_1111_1111_1111_1111_1111_1111_111#E127) ..
     (2#0.1111_1111_1111_1111_1111_1111_1111_111#E127);

-- Predefined subtypes within the Ada Language:

subtype NATURAL  is INTEGER range 0 .. INTEGER'LAST; -- 32_767
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST; -- 32_767
subtype LONG_NATURAL  is LONG_INTEGER
                        range 0 .. LONG_INTEGER'LAST;
subtype LONG_POSITIVE is LONG_INTEGER
                        range 1 .. LONG_INTEGER'LAST;

-- Predefined STRING type within the Ada Language:

type STRING is array (POSITIVE range <>) of CHARACTER;
pragma PACK(STRING);

-- The type DURATION is predefined for use with Ada DELAY.

type DURATION is delta 2.0 ** (-14)
           range -131_072.0 .. 131_072.0  - 2.0 ** (-14)

-- The predefined operators for the type DURATION are the same
-- as for any fixed point type within the Ada language.
```

F.4.2   Package SYSTEM

    Within the various implementations, no corresponding package
body is required for the package SYSTEM. The package SYSTEM is
as follows:

```
type ADDRESS is new LONG_INTEGER;
type NAME    is (AdaVAX, Ada_L, Ada_M);
SYSTEM_NAME  : constant NAME := AdaVAX;
STORAGE_UNIT : constant :=  8;
MEMORY_SIZE  : constant :=  2**30 - 1;

-- System-Dependent Named Numbers:

MIN_INT      : constant :=  -(2**31);
MAX_INT      : constant :=   (2**31)-1;
MAX_DIGITS   : constant :=  9;
MAX_MANTISSA : constant :=  31;
FINE_DELTA   : constant :=  2.0**(-31);
TICK         : constant :=  0.01;

-- Other System-Dependent Declarations

subtype PRIORITY is INTEGER range 1..15;
--
-- The following exceptions are provided as a "convention"
-- whereby the Ada program can be compiled with all implicit
-- checks suppressed (i.e., pragma SUPPRESS or equivalent),
-- explicit checks included as necessary, the appropriate
-- exception raised when required, and then the exception is
-- either handled or the Ada program terminates.
--
ACCESS_CHECK             : exception;
DISCRIMINANT_CHECK       : exception;
INDEX_CHECK              : exception;
LENGTH_CHECK             : exception;
RANGE_CHECK              : exception;
DIVISION_CHECK           : exception;
OVERFLOW_CHECK           : exception;
ELABORATION_CHECK        : exception;
STORAGE_CHECK            : exception;
--
-- The following exceptions provide for (1) Ada programs that
-- contain unresolved subprogram calls and (2) VAX/VMS
-- system-level errors.
--
UNRESOLVED_REFERENCE     : exception;
SYSTEM_ERROR             : exception;
```

F.5   Character Set

Ada compilations may be expressed using the following characters, in addition to the basic character set:

lower case letters:

a b c d e f g h i j k l m n o p q r s t u v w x y z

special characters:

! $ % ? @ [   ] ^ ` {) ~

The following transliterations are permitted (see Paragraph 2.10 of the Ada Language Reference Manual):

a.   Exclamation mark for vertical bar;

b.   Colon for sharp; and

c.   Percent for double_quote.

F.6   Declaration and Representation Restrictions

Declarations are described in Chapter 3 of the Ada Language Reference Manual. Representation specifications are described in Chapter 13 and discussed here.

In the following specifications, the capitalized word SIZE indicates the number of bits used to represent an object of the type under discussion. The upper case symbols D, L, and R correspond to those discussed in Section 3.5.9 of the Ada Language Refe~~nce Manual.

F.6.1   Integer Types

Integer types are specified with constraints of the form:

        RANGE L..R

where:
        R <= SYSTEM.MAX_INT & L >= SYSTEM.MIN_INT

For an integer type, length specifications of the form:

        FOR t'SIZE USE n;

may specify integer values n such that n is in 2..32,

        R <= 2**(n-1)-1 & L >= -2**(n-1);

or else such that

        R <= (2**N)-1 & L >= 0

and N is in 1..31.

For a stand-alone object of integer type, a default SIZE of 16 is used when:

        R <= 2**15-1 & L >= 2**15

Otherwise a SIZE of 32 is used.

For components of integer types within packed composite objects, the smaller of the default stand-alone SIZE or the SIZE from a length specification will be used.

F.6.2 Floating Types

Floating types are specified with constraints of the form:

DIGITS D

where D is an integer value in 1 through 9.

For floating point types, length specifications of the form:

FOR t'SIZE USE n;

are permitted only when the integer values N = 32 when D <= 6, or N = 64 when D <= 9.

When no length specification is provided, a size of 32 is used when D <= 6; 64 when D is 7 through 9.

F.6.3 Fixed Types

Fixed types are specified with constraints of the form:

delta D range L..R

where:

$$\frac{\max (abs(R), abs(L)) < 2^{**}31-1}{actual\_delta}$$

The actual delta defaults to the largest integral power of 2 less than or equal to the specified delta D. (This implies that fixed point values are stored right-aligned.)

For fixed point types, length specifications of the form:

for T'SIZE use N;

are permitted only when N in 1 .. 32, if:

R - actual_delta <= 2**(N-1)-1 * actual_delta

and

L + actual_delta >= -2**(n-1) * actual)delta

or

R - actual_delta <= 2**(N)-1 * actual_delta

and

L >= 0

For stand-alone objects of fixed point type, a default size of 32 is used. For components of fixed point types within packed composite objects, the size from the length specification will be used.

Specifications of the form:

for T'SMALL use X;

are permitted for any value of X, such that X <= D. X must be specified either as a base 2 value or as a base 10 value. Note that when X is specified as other than a power of 2, actual_delta will still be the largest integreal power of two less than X.

F.6.4   Enumeration Types

In the absence of a representation specification for an
enumeration type T, the internal representation of T'FIRST = 0.
The default SIZE for a stand-alone object of enumeration type T
will be the smallest of the values 8, 16, or 32, such that the
internal representation of T'FIRST and T'LAST both falls within
the range:

$$-2**(T'SIZE-1) \;\; .. \;\; 2**(T'SIZE-1)-1.$$

For enumeration types, length specification of the form:

for T'SIZE use N;

and/or enumeration representations of the form:

for T use <aggregate>;

are permitted for N in 2..32, provided that the internal
representations and the SIZE conform to the relationship
specified above.

Or else for N in 1..31, is supported for enumeration types
and provides an internal representation of:

$$T'FIRST>=0 \;\; .. \;\; T'LAST<=2**(T'SIZE)-1.$$

For components of enumeration types within packed composite
objects, the smaller of the default stand-alone SIZE, or the SIZE
from a length specification will be used.

Enumeration representation on types derived from the
predefined type BOOLEAN will not be accepted, but length
specifications will be accepted.

F.6.5 Access Types

For access type, T, length specifications of the form:

        for T'SIZE use N;

will not affect the run-time implementation of T, therefore N = 32 is the only value permitted for SIZE, which is the value returned by the attribute.

For collection size specifications of the form:

        for T'STORAGE_SIZE use N;

any value of N is permitted (and that value will be returned by the attribute call). The collection size specification will affect the implementation of T and its collection at run-time by limiting the number of objects for type T that can be allocated.

F.6.6 Arrays and Records

For arrays and records, length specifications of the form:

        for T'SIZE use N;

may cause arrays and records to be packed, if required, to accommodate the length specification. If the SIZE specified is not large enough to contain all possible values of the components, a diagnostic message of severity ERROR is issued.

The PACK pragma may be used to minimize wasted space, if any, between components of arrays and records. The pragma causes the type representation to be chosen such that storage space requirements are minimized at the possible expense of data access time and code space.

For records, a component clause of the form:

        at N [range i..j]

specifies the allocation of components in a record. Bits are numbered 0..7 from the right and bit 8 starts at the right of the next higher-number byte. Each location specification must allow at least X bits of range, where X is large enough to hold any value of the subtype of the component being allocated. Otherwise, a diagnostic message of severity ERROR is generated.

For records, an alignment clause of the form:

    at mod N

specify alignments of N bytes for 1 byte, 2 bytes (VAX "word"),
and 4 bytes (VAX "long_word").

If it is determinable at compilation time that the SIZE of a
record or array type or subtype maybe outside the range of
STANDARD.LONG_INTEGER, a diagnostic message of severity WARNING
is generated. Declaration of an object of such a type or subtype
would raise NUMERIC_ERROR when elaborated. Note that a
discriminant record or array may never raise the NUMERIC_ERROR
when elaborated based on the actual discriminant provided.

F.6.7 Other Length Specifications

Length Specifications are described in Section 13.2 of the
Ada Language Reference Manual.

A length specification for a task type T, of the form:

    for T'SIZE use N;

specifies the number of bits to be allocated for objects of the
task type T. For the VAX/VMS target, N must be defined:

$$N = 8 * (109 + 13 * number\_of\_entries)$$

Where number_of_entries is the number of entries declared in the
task type specification.

F.7 System Names

Refer to Section 13.7 of the Ada Language Reference Manual for a discussion of package SYSTEM.

The available system names are "AdaVAX", "Ada_L", and "Ada_M"; the system name is chosen based on the targets supported, but it can not be changed. In the case of VAX/VMS, the system name is "AdaVAX".

F.8 Address Clauses

Refer to Section 13.5 of the Ada Language Reference Manual for a discussion of Address Clauses. Address clauses for objects and code are allowed by the VAX/VMS target, but they have no effect beyond changing the value returned by the 'ADDRESS attribute call.

The Run-Time Support Library (RSL) for the VAX/VMS target does not handle hardware interrupts. All hardware interrupts are handled by the VAX/VMS operating system. However, the VAX_VMS target uses asynchronous system traps (ASTs) in a manner similar to interrupt entries.

F.9 Unchecked Conversions

Refer to Section 13.10.2 of the Ada Language Reference Manual for a description of UNCHECKED_CONVERSION.

A program is erroneous if it performs UNCHECKED_CONVERSION when the source and target have different sizes.

F.10 Restrictions on the Main (Sub)Program

Refer to Section 10.1 of the Ada Language Reference Manual for a discussion of the main (sub)program. The subprogram designated as the main (sub)program cannot have parameters. The designation as the main (sub)program of a subprogram whose specification contains a formal_part results in a diagnostic of severity ERROR at link time.

The main (sub)program can be a function, but the return value will not be available upon completion of the main (sub)program's execution. The main (sub)program may not be an imported subprogram.

F.11   Input/Output

Refer to Chapter 14 of the Ada Language Reference Manual for a description of Ada Input/Output (I/O).

The RSL I/O subsystem provides the following packages to the user: TEXT_IO, SEQUENTIAL_IO, DIRECT_IO, and LOW_LEVEL_IO.   These packages execute in the context of the an individual Ada task making the I/O request.   Consequently, all of the code that process an I/O request on behalf of the Ada task executes sequentially.   The package IO_EXCEPTIONS defines all of the exceptions needed by the packages TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO.   The specification of this package is given in Section 14.5 of the Ada LRM.   This package is visible to all of the constituent packages of the RSL I/O subsystem so that appropriate exception handlers can be inserted.

High-level I/O in AdaVAX is performed solely on external files.   No allowance is provided in the RSL I/O subsystem for memory resident files (i.e., files which do not reside on a peripheral device).   This is true even in the case of temporary files.   With the external files residing on peripheral devices, only the various VAX/VMS quotas restricts the number of files that may be open on an individual peripheral device.

Section 14.1 of the Ada LRM states that all I/O operations are expressed as operations on objects of some file type, rather than in terms of an external file.   File objects are implemented in AdaVAX as access objects that point to a data structure call the File Control Block (FCB).   This FCB is defined internally to each high-level I/O package; its purpose is to represent an external file.   The FCB contains all of the I/O-specific information about an external file that is needed by the high-level packages to accomplish the requested I/O operation.

F.11.1   Naming External Files

The naming conventions for external files in AdaVAX are of particular importance to the user.   An external file name for Ada I/O can be any valid path name (e.g., disk:[directories]filename.ext) in the VAX/VMS environment.

F.11.2 The FORM Specification for External Files

The FORM specification for external Files created by TEXT_IO include the default (i.e., the NULL string) and the two shorthand strings: "PASS_ALL" or "LOG_FILE". The only FORM specification for external files created by SEQUENTIAL_IO and DIRECT_IO is the default of the NULL string. Note that opening the external file after its creation still utilizes the file attributes assigned to the file when it was created and, therefore, the only legal FORM specification is the NULL string.

An allowable FORM string in TEXT_IO has syntax defined by the grammar is shown in Table F-1 below. The tokens of the grammar may be separated by any combination of blanks (' ') and horizontal tab (ASCII.HT) characters. The FORM parameter is not case sensitive, but repetition of a file_attribute_item is not allowed. The record format values valid with the file organization SEQUENTIAL are: STREAM, STREAM_CARRIAGE_RETURN, STREAM_LINE_FEED, and UNDEFINED. Note that the VARIABLE_FIXED_CONTROL record format is not valid with the INDEXED file organization.

In TEXT_IO, the following default FORM value is assumed when the FORM parameter is the NULL string:

        "RECORD_FORMAT := VARIABLE, " &
        "FILE_ORGANIZATION := SEQUENTIAL, " &
        "CARRIAGE_CONTROL := CARRIAGE_RETURN"

The "PASS_ALL" FORM parameter is equivalent to the string:

        "RECORD_FORMAT := VARIABLE, " &
        "FILE_ORGANIZATION := SEQUENTIAL, " &
        "CARRIAGE_CONTROL := NONE"

The "LOG_FILE" FORM parameter is equivalent to the string:

        "RECORD_FORMAT := VARIABLE_FIXED_CONTROL, " &
        "FILE_ORGANIZATION := SEQUENTIAL, " &
        "CARRIAGE_CONTROL := PRINT"

Left Hand Side	Right Hand Side
form_string	:== "" | shorthand_string | file_attribute_list
shorthand_string	:== PASS_ALL | LOG_FILE
file_attribute_list	:== file_attibute_item (,file_attribute_item)
file_attribute_item	:== record_format_string | file_organization_string | carriage_control_string
record_format_string	:== RECORD_FORMAT := record_format
record_format	:== VARIABLE | FIXED | STREAM | VARIABLE_FIXED_CONTROL | STREAM_CARRIAGE_CONTROL | STREAM_LINE_FEED | UNDEFINED
file_organization_string	:== FILE_ORGANIZATION := file_organization
file_organization	:== SEQUENTIAL | RELATIVE |INDEXED
carriage_control_string	:== CARRIAGE_CONTROL := carriage_control
carriage_control	:== FORTRAN | CARRIAGE_RETURN | PRINT | NONE

Table F-1 - FORM String Grammar

F.11.2 The FORM Specification for External Files   F-19

F.11.3   External File Processing

Section 14 of the Ada LRM defines two kinds of access to external files: sequential access and direct access. A file object used for sequential access is call a sequential file, and one used for direct access is called a direct file. Three file modes are defined: IN_FILE, OUT_FILE, and INOUT_FILE. All three file modes are allowed for direct files, whereas only the modes IN_FILE and OUT_FILE are allowed for sequential files.

AdaVAX takes the view that files of mode IN_FILE already contain data, making them suitable for reading, while files of mode OUT_FILE are empty, making them suitable for writing. Files of mode INOUT_FILE may contain data or may be empty, making them suitable for reading or writing. An attempt to create a file of mode IN_FILE will raise the exception USE_ERROR since a newly created file is empty (i.e., not suitable for reading). Stated more simply, AdaVAX restricts the creation of files to those of mode OUT_FILE or INOUT_FILE.

Processing allowed on external files is determined by the access controls set by the owner of the file and by the physical characteristics of the underlying device. The following restrictions apply:

a.   A user may open a file as an IN_FILE only if that user has read access to the node. A user may open a file as an OUT_FILE only if that user has write access to the node. Finally, a user may open a file as an INOUT_FILE only if that user has read and write access to the node.

b.   The attempt to CREATE a file with the mode IN_FILE is not supported since there will be no data in the file to read.

c.   Multiple OPENs are allowed to read from a file, but all OPENs to write require exclusive access to the file. The exception USE_ERROR is raised if this restriction is violated.

d.   No positioning operations are allowed on files associated with a printer or hard-copy terminal. The exception USE_ERROR is raised if this restriction is violated.

F.11.4 Text Input/Output

The specification of TEXT_IO is given by Section 14.3.10 of
the Ada LRM. TEXT_IO is invoked by the Ada task to perform
sequential access I/O operations on text files (i.e., files whose
content is in a human-readable form). TEXT_IO is not a generic
package, and thus, its subprograms may be invoked directly from
the Ada task, using objects with base type or parent type in the
language-defined type CHARACTER (and or course STRING). TEXT_IO
also provides the generic packages INTEGER_IO, FLOAT_IO, FIXED_IO
and ENUMERATION_IO for the reading and writing of numeric values
and enumeration values. The generic packages within TEXT_IO
require an instantiation for a given element type before any of
their subprograms are invoked.

The implementation-defined type COUNT that appears in Section
14.3.10 of the Ada LRM is defined as follows:

type COUNT is range 0..LONG_INTEGER'LAST;

The implementation-defined subtype FIELD that appears in Section
14.3.10 of the Ada LRM is defined as follows:

subtype FIELD is INTEGER range 0..INTEGER'LAST;

At the beginning of program execution, the STANDARD_INPUT
file and the STANDARD_OUTPUT file are open and associated with
the ALS/N-supported standard input and output files. The
STANDARD_INPUT and STANDARD_OUTPUT file cannot be deleted,
attempts to do so raise the exception USE_ERROR. Additionally,
if a program terminates before an open file is closed (except for
STANDARD_INPUT and STANDARD_OUTPUT), then the last line the user
put to the file may be lost.

A program is erroneous if concurrently executing tasks
attempt to perform overlapping GET and/or PUT operations on the
same terminal. Because of the physical nature of DecWriters and
Video terminals, the semantics of text layout as specified in Ada
Language Reference Manual Section 14.3.2 (especially the concepts
of current column number and current line) cannot be guaranteed
when GET operations are interweaved with PUT operations.
Programs that rely on the semantics of text layout under those
circumstances are erroneous.

For TEXT_IO processing, the line length can be no longer than
the maximum VAX/VMS record length minus one (i.e., 255
characters). An attempt to write over the record length boundary
will result in writing a full record and starting a new record.
An attempt to set the line length through SET_LINE_LENGTH to a
length greater than 255 will result in USE_ERROR. An attempt to
read a file with a line length greater than 255 will also result
in a USE_ERROR.

F.11.5 Sequential Input/Output

The specification of SEQUENTIAL_IO is given in Section 14.2.3 of the Ada LRM. SEQUENTIAL_IO is invoked by the Ada task to perform I/O of the records of a file in an arbitrary order. The package SEQUENTIAL_IO requires a generic instantiation for a given element type before any of its subprograms may be invoked. Once the package SEQUENTIAL_IO is made visible, it will perform any service defined by the subprograms declared in its specification.

The following restrictions are imposed on the use of the package Sequential_IO:

a. A null file name parameter to the CREATE procedure (for opening a temporary file) is not appropriate, and raises the exception NAME_ERROR.

b. Writing a record on a file associated with a tape adds the record to the file such that the record just written becomes the last record of the file.

c. On a disk or tape, the DELETE procedure closes the file and sets its size to zero so that its data may no longer be accessed.

d. The subprogram END_OF_FILE always returns FALSE for a character-oriented device and RESET performs no action on a character-oriented device.

F.11.6 Direct Input/Output

The specification of DIRECT_IO is given in Section 14.2.5 of the Ada LRM. DIRECT_IO is invoked by the Ada task to perform I/O of the records of a file in an arbitrary order. The package DIRECT_IO requires a generic instantiation for a given element type before any of its subprograms may be invoked. Once the package DIRECT_IO is made visible, it will perform any service defined by the subprograms declared in its specification.

The implementation-defined type COUNT that appears in Section 14.2.5 of the Ada LRM is defined as follows:

```
type COUNT is range 0..LONG_INTEGER'LAST;
```

F.11.7   Low Level Input/Output

The package LOW_LEVEL_IO defines a standard interface to
allow an application to interact directly with a physical device.
LOW_LEVEL_IO provides a definition of data types for a physical
device and data to be operated on, along with the standard
procedures SEND_CONTROL and RECEIVE_CONTROL.   The procedure
SEND_CONTROL may be used to send control information to a
physical device.   RECEIVE_CONTROL may be used to monitor the
execution of an I/O operation by requesting information from a
physical device.

with SYSTEM;

package LOW_LEVEL_IO is

```
    type IO_BUFFER_ADDRESS is new SYSTEM.ADDRESS;
    type IO_BUFFER_COUNT is new INTEGER;
    type IO_TIME_OUT is new INTEGER;

    type IO_FUNCTION is (
        read_data,      -- read data
        write_data,     -- write data
        initialize,     -- initialize the device and
                        -- return the device_code
        cancel,         -- cancel IO request
        control);       -- return control information

    type DEVICE_TYPE is new LONG_INTEGER;
        DEVICE_NAME_LENGTH: constant INTEGER := 32;

    type IO_REQUEST_BLOCK is record
        REQUESTED_FUNCTION: IO_FUNCTION;
        DEVICE_NAME:        STRING(1..DEVICE_NAME_LENGTH);
        DEVICE:             DEVICE_TYPE;
        BUFFER_ADDRESS:     IO_BUFFER_ADDRESS;
        BUFFER_COUNT:       IO_BUFFER_COUNT;
        TIME_OUT:           IO_TIME_OUT;
    end record;
```

```
      type IO_RETURN_STATUS is (
            ss_normal,            -- normal completion
            ss_abort,             -- all "failure" status codes
            ss_accvio,
            ss_devoffline,
            ss_exquota,
            ss_illefc,
            ss_insfmem,
            ss_ivchan,
            ss_nopriv,
            ss_unasefc,
            ss_linkabort,
            ss_linkdiscon,
            ss_protocol,
            ss_connecfail,
            ss_filalracc,
            ss_invlogin,
            ss_indevnam,
            ss_linkexit,
            ss_nolinks,
            ss_nosuchnode,
            ss_reject,
            ss_remrsrc,
            ss_shut,
            ss_toomuchdata,
            ss_unreachable);


      type IO_STATUS_BLOCK is record
            BYTE_COUNT:           IO_BUFFER_COUNT;
            RETURNED_STATUS:      IO_RETURN_STATUS;
      end record;

      procedure SEND_CONTROL (DEVICE: in   DEVICE_TYPE;
                              DATA: in out IO_REQUEST_BLOCK);

      procedure RECEIVE_CONTROL (DEVICE: in   DEVICE_TYPE;
                                 DATA: in out IO_STATUS_BLOCK);

end LOW_LEVEL_IO;
```

F.12 System Defined Exceptions

In addition to the exceptions defined in the Ada Language
Reference Manual, this implementation pre-defines the exceptions
shown in Table F-2 below.

Name	Significance
ACCESS_CHECK	The ACCESS_CHECK exception has been raised explicitly within the program.
DISCRIMINANT_CHECK	DISCRIMINANT_CHECK exception has been raised explicitly within the program.
INDEX_CHECK	The INDEX_CHECK exception has been raised explicitly within the program.
LENGTH_CHECK	The LENGTH_CHECK exception has been raised explicitly within the program.
RANGE_CHECK	The RANGE_CHECK exception has been raised explicitly within the program.
DIVISION_CHECK	The DIVISION_CHECK exception has been raised explicitly within the program.
OVERFLOW_CHECK	The OVERFLOW_CHECK exception has been raised explicitly within the program.
ELABORATION_CHECK	ELABORATION_CHECK exception has been raised explicitly within the program.
STORAGE_CHECK	The STORAGE_CHECK exception has been raised explicitly within the program.
UNRESOLVED_REFERENCE	Attempted call to a routine not linked into the executable image.
SYSTEM_ERROR	Serious error detected in underlying VAX/VMS operating system.

Table F-2 - System Defined Exceptions

F.12 System Defined Exceptions

F.13 Machine Code Insertions

The Ada language definition permits machine code insertions
as described in Section 13.8 of the Ada Language Reference
Manual. This section describes the implementation specific
details for writing machine code insertions as provided by the
predefined library package MACHINE_CODE.

The user may, if desired, include MACRO instructions within
an Ada program. This is done by including a subprogram in the
program which contains only record aggregates defining machine
code instructions. The package MACHINE_CODE, included in the
system program library, contains type, record and constant
declarations which are used to form the instructions. Each field
of the aggregate contains a field of the resulting machine
instruction. These fields are specified in the order in which
they appear in the actural instruction. Records for one- and
two- byte instruction codes are available. Each instruction
record is discriminated using the instruction code. The record
components determined by the discriminant are the arguments of
the record. Arguments are represented using records whose
discriminants are called address modes. The discriminant
determines what additional information (if any) must be
associated with the argument. Separate records are available for
specifying data.

```
        WITH machine_code;
        USE machine_code;
        FUNCTION fixed_multiply
            (multiplier_1    : IN LONG_INTEGER;     -- in R0
             multiplier_2    : IN LONG_INTEGER;     -- in R1
             scaling_factor : IN LONG_INTEGER       -- in R2
            ) RETURN LONG_INTEGER IS                -- in R0
        BEGIN
            -- EMUL R0, R1, #0, R0
            -- named aggregate notation
            byte_op_code
                (op       => emul,
                 emul_1 => long_word_general_operand(op => R0),
                 emul_2 => long_word_general_operand(op => R1),
                 emul_3 => long_word_general_operand(op => L0),
                 emul_4 => quad_word_general_operand(op => R0));
            -- ASHQ R2, R0, R0
            -- positional notation
            byte_op_code
                (ashq,
                 byte_word_general_operand(op => R2),
                 quad_word_general_operand(op => R0),
                 quad_word_general_operand(op => R2));
        END fixed_multiply;
```

Note that either positional or named aggregates may be used.

ALS/N supports machine code insertions through calls to
procedures whose bodies are composed of sequences of assembly
language instructions. Each instruction in the sequence is
specified as an aggregate of either the record type BYTE_OP_CODE
or WORD_OP_CODE, both declared in the Runtime Support Library
package MACHINE_CODE. These types are variant records whose
discriminant is a symbolic VAX-11 instruction opcode. Components
of each discriminated record correspond to the instruction
operands appropriate to a given instruction opcode. Components
of BYTE_OP_CODE and WORD_OP_CODE are themselves variant records.
Their discriminated components are used to specify operand
addressing modes together with needed registers, displacements
and literal values. The type mark BYTE_OP_CODE is used for those
VAX-11 instructions whose opcodes can be represented in a single
byte (e.g., MOVL). WORD_OP_CODE is used for those VAX-11
instructions whose opcodes consume two bytes (e.g., CMPH).

These ideas are illustrated in Figure F-1 below. A more
detailed explanation of how machine code insertions are composed
for the VAX target is given in section 6.14. In this example the
procedure TIMES_TWO is used to double integer valued objects. It
effects a multiplication of its single argument using the
Arithmetic Shift Logical instruction, ASHL. The value to be
multiplied is passed by reference to the procedure TIMES_TWO and
can be found four bytes away from the address held in the
Argument Pointer, AP. Using byte displacement deferred
addressing mode (i.e., IB_AP) to access the procedure argument
allows the shift by one bit to occur "in place".

```
with MACHINE_CODE ; use MACHINE_CODE ;
  procedure TIMES_TWO(value : IN OUT integer) is
  begin

    BYTE_OP_CODE'(
              OP => ASHL,                            -- Instruction = ASHL
          ASHL_1 => ( OP => IMD, B_IMD => 1 ),       -- Operand 1 = "#1"
          ASHL_2 => ( OP => IB_AP, BYTE_DISP => 4),  -- Operand 2 = @4(AP)
          ASHL_3 => ( OP => IB_AP, BYTE_DISP => 4) ) ;-- Operand 3 = "@4(AP)

    end TIMES_TWO ;
```

Figure F-1 - Machine Code Insertion

F.13.1 Machine Features

     This paragraph describes specific machine language features
needed to write code statements. These machine features include
the DISP and ADDRESS attributes and the address mode specifiers.
The address mode specifiers make it possible to describe both the
address mode and register number of any operand as a single value
by mapping these values directly onto the first byte of each
operand. The following is an enumeration of all mode specifiers:

```
--
-- The first 64 are the short literal modes.
-- These mode specifiers signify (short literal mode, value)
-- combinations.  The values are in the range 0 to 63.
--
```

L0,	L1,	L2,	L3,
L4,	L5,	L6,	L7,
L8,	L9,	L10,	L11,
L12,	L13,	L14,	L15,
L16,	L17,	L18,	L19,
L20,	L21,	L22,	L23,
L24,	L25,	L26,	L27,
L28,	L29,	L30,	L31,
L32,	L33,	L34,	L35,
L36,	L37,	L38,	L39,
L40,	L41,	L42,	L43,
L44,	L45,	L46,	L47,
L48,	L49,	L50,	L51,
L52,	L53,	L54,	L55,
L56,	L57,	L58,	L59,
L60,	L61,	L62,	L63,

```
--
-- Next are the (index mode, register) combinations.
--

        X_R0,           X_R1,           X_R2,           X_R3,
        X_R4,           X_R5,           X_R6,           X_R7,
        X_R8,           X_R9,           X_R10,          X_R11,
        X_AP,           X_FP,           X_SP,           X_PC,
--
-- The following are the (register mode, register) combinations.
--

        R0,             R1,             R2,             R3,
        R4,             R5,             R6,             R7,
        R8,             R9,             R10,            R11,
        AP,             FP,             SP,             PC,
--
-- The following are the (indirect register mode, register)
-- combinations.
--

        IR0,            IR1,            IR2,            IR3,
        IR4,            IR5,            IR6,            IR7,
        IR8,            IR9,            IR10,           IR11,
        IAP,            IFP,            ISP,            IPC,
--
-- Next are the (autodecrement register mode, register)
-- combinations.
--

        DEC_R0,         DEC_R1,         DEC_R2,         DEC_R3,
        DEC_R4,         DEC_R5,         DEC_R6,         DEC_R7,
        DEC_R8,         DEC_R9,         DEC_R10,        DEC_R11,
        DEC_AP,         DEC_FP,         DEC_SP,         DEC_PC,
--
-- Next are the (autoincrement register mode, register)
-- combinations. IMD (immediate mode) is autoincrement
-- mode using the PC.
--

        R0_INC,         R1_INC,         R2_INC,         R3_INC,
        R4_INC,         R5_INC,         R6_INC,         R7_INC,
        R8_INC,         R9_INC,         R10_INC,        R11_INC,
        AP_INC,         FP_INC,         SP_INC,         IMD,
--
-- The following are the (autoincrement deferred mode, register)
-- combinations.  A (absolute address mode) is autoincrement
-- deferred using the PC.
--

        IR0_INC,        IR1_INC,        IR2_INC,        IR3_INC,
        IR4_INC,        IR5_INC,        IR6_INC,        IR7_INC,
        IR8_INC,        IR9_INC,        IR10_INC,       IR11_INC,
        IAP_INC,        IFP_INC,        ISP_INC,        A,
```

```
--
-- The following are the (byte-displacement mode, register)
-- combinations. B_PC is byte-relative mode for the PC.
--
        B_R0,           B_R1,           B_R2,           B_R3,
        B_R4,           B_R5,           B_R6,           B_R7,
        B_R8,           B_R9,           B_R10,          B_R11,
        B_AP,           B_FP,           B_SP,           B_PC,
--
-- Next are the (byte-displacement deferred mode, register)
-- combinations.  IB_PC is byte-relative deferred mode for
-- the PC.
--
        IB_R0,          IB_R1,          IB_R2,          IB_R3,
        IB_R4,          IB_R5,          IB_R6,          IB_R7,
        IB_R8,          IB_R9,          IB_R10,         IB_R11,
        IB_AP,          IB_FP,          IB_SP,          IB_PC,
--
-- The following are the (word-displacement mode, register)
-- combinations.  W_PC is word relative mode for the PC.
--
        W_R0,           W_R1,           W_R2,           W_R3,
        W_R4,           W_R5,           W_R6,           W_R7,
        W_R8,           W_R9,           W_R10,          W_R11,
        W_AP,           W_FP,           W_SP,           W_PC,
--
-- The following are the (word-displacement deferred mode,
-- register) combinations.  IW_PC is word relative deferred
-- mode for the PC.
--
        IW_R0,          IW_R1,          IW_R2,          IW_R3,
        IW_R4,          IW_R5,          IW_R6,          IW_R7,
        IW_R8,          IW_R9,          IW_R10,         IW_R11,
        IW_AP,          IW_FP,          IW_SP,          IW_PC,
--
-- Next are the (longword-displacement mode, register)
-- combinations.  L_PC is longword-relative mode.
--
        L_R0,           L_R1,           L_R2,           L_R3,
        L_R4,           L_R5,           L_R6,           L_R7,
        L_R8,           L_R9,           L_R10,          L_R11,
        L_AP,           L_FP,           L_SP,           L_PC,
--
-- The following are the (longword-displacement deferred mode,
-- register) combinations.  IL_PC is longword-relative deferred
-- mode.
--
        IL_R0,          IL_R1,          IL_R2,          IL_R3,
        IL_R4,          IL_R5,          IL_R6,          IL_R7,
        IL_R8,          IL_R9,          IL_R10,         IL_R11,
        IL_AP,          IL_FP,          IL_SP,          IL_PC);
```

F.13.1 Machine Features

F.13.2 ADDRESS and DISP Attributes

The following restriction applies to the use of the ADDRESS
and DISP attributes:

a. All displacements and addresses (i.e., branch destinations,
   program counter addressing mode displacements, etc.) must be
   static expressions.

b. Since neither the ADDRESS nor the DISP attributes return
   static values, they can not be used in code statements within
   the Ada compilation unit.

F.13.3 Restrictions on Assembler Constructs

These unsupported Assembler constructs within the
MACHINE_CODE package are as follows:

a. The VAX/VMS assembler's capability to compute the length of
   immediate and literal data is not replicated in MACHINE_CODE.
   This means the user cannot supply a value without specifying
   the length of that value. This disallows the assembler
   operand general formats: D(R), G, G^G, #cons, #cons[Rx],
   D(R)[Rx], G[Rx], G^location[Rx], @D(R)[Rx], @G[Rx], @D(R), @G
   such that D and G are byte, word, or long_word values.
   Operands must contain address mode specifiers which
   explicitly define the length of any immediate or literal
   values of that operand.

b. The radix of the assembler notation is decimal. To express a
   hexadecimal literal, the notation 16#literal# should be used
   instead of ^X.

c. To construct an octaword, quadword, g_float or h_float
   number, it is important for the user to remember that the
   component fields of the records that make up the long numeric
   types are signed. This means that the user must take care to
   be assured that the values for these components, although
   signed, are interpreted correctly by the instruction set
   architecture.

d. Edit instruction streams must be constructed through the use
   of the VAX data statements described in Section 6.12.3.

e. Compatibility mode instruction streams must be constructed
    through the use of the VAX data statements described in
    Section 6.12.3, if still supported on the VAX computer being
    utilized as the target machine (i.e., VAX-11/780 and 785, but
    not the VAX-8600).

f. No error messages are generated if the PC is used as the
    register for operands taking a single register, if the SP or
    PC are used for operands taking two registers, or if the AP,
    FP, SP, or PC is used for operands taking four registers.

g. No error message is generated if the PC is used in register
    deferred or autodecrement mode.

h. If any register other than the PC is used as both the
    simple_operand and as the index_reg for an operand (see
    Section 6.14.1.2 for definitions of simple_operand and
    index_reg), no error message is generated. An example of
    this case is the VAX Assembler operand (7)[7].

i. Generic opcode selection is not supported. This means the
    opcode which reflects the specified number of operands must
    be used. For example, for 2 operand word addition, ADDW2
    must be used, not just ADDW.

j. The PC is not supplied as a default if no register is
    specified in an operand. The user must supply the mode
    specifier which is mapped onto the PC. Examples are IMD, A,
    B_PC, W_PC, etc.

F.14 Machine Instructions and Data

This section describes the syntactic details for writing code statements (machine code insertions) as provided for the VAX by the pre-defined package MACHINE_CODE. The format for writing code statements is detailed, as are descriptions of the values to be supplied in the code statements. Each value is described by the named association for that value and its defined in the order in which it must appear in positional notation. The programmer should refer to the VAX-11 Architecture Handbook along with this section to ensure that the machine instructions are correct from an architectural viewpoint.

To ensure a proper interface between Ada and machine code insertions, the user must be aware of the calling conventions used by the Ada compiler.

F.14.1 VAX Instructions

The general format for VAX code statements where the opcode is a one byte opcode is

```
BYTE_OP_CODE (OP => opcode (,"opcode"_1 => operand
                          (,"opcode"_2 => operand
                           (,"opcode"_3 => operand
                            (,"opcode"_4 => operand
                             (,"opcode"_5 => operand
                              (,"opcode"_6 => operand))))))));
```

The general format for VAX code statements where the opcode is a two byte opcode is

```
WORD_OP_CODE (OP => opcode2 (,"opcode2"_1 => operand
                           (,"opcode2"_2 => operand
                            (,"opcode2"_3 => operand
                             (,"opcode2"_4 => operand
                              (,"opcode2"_5 => operand
                               (,"opcode2"_6 => operand))))))));
```

where "opcode"_n and "opcode2"_n is the result of the concatenation of the VAX opcode, an underscore, and the position of the operand in the VAX instruction. The BYTE_OP_CODE and WORD_OP_CODE statements always require an opcode and may include from one to six operands. The opcode mnemonics are precisely the same as described in the previously referenced VAX-11 Architecture Handbook. The VAX address modes divide the operands into six general categories: Short Literal Operand, Indexed Operand, Register Operand, Byte-Displacement Operand, Word-Displacement Operand, and Long_Word-Displacement Operand.

F.14.1.1   Short Literal Operands

The VAX/VMS Assembler format for short literal operands is

S^#cons

where cons is an integer constant with a range from 0 to 63
(decimal).

The code statement format for short literal operands is

(OP => short_lit)

where short_lit is one of the enumerated values, range L0 to L63,
of the address mode specifiers in Section 6.11.1.

The following are examples of how some VAX Assembler short
literals would be expressed in code statements:

```
S^#7    becomes  (OP => L7)
S^#33   becomes  (OP => L33)
S^#60   becomes  (OP => L60)
```

(For explanations of named and unnamed component association, see
Section 4.3 of the Ada Language Reference Manual.)

F.14.1.2   Indexed Operands

The VAX/VMS Assembler format for the indexed operands is

simple_operand[Rx]

where a simple_operand is an operand of any address mode except
register, literal, or index.

The general code statement format for indexed operands is

(index_reg, simple_operand) or
    (OP => index_reg, OPND => simple_operand)

where index_reg is one of the enumerated address mode specifiers,
range X_R0 to X_SP, from Section 6.11.1. Simple_operand is an
operand of any address mode except register, literal, or index.

For example, the following are indexed assembler operands:

a. (R8)[R7] becomes (X_R7, (OP => IR8))

b. (R8)+[R7] becomes (X_R7, (OP => R8_INC))

c. I^#600[R4] becomes (X_R4, (IMD,600))

d. -(R4)[R3] becomes (X_R3, (OP => DEC_R4))

e. B^4(R9)[R3] becomes (X_R3, (B_R9,4))

f. W^800(R8)[R5] becomes (X_R5, (W_R8,800))

g. L^34000(R8)[R4] becomes (X_R4,(L_R8,34000))

h. B^10[R9] becomes (X_R9, (B_PC,10))

i. W^130[R2] becomes (X_R2, (W_PC,130))

j. L^35000[R6] becomes (X_R6, (L_PC,35000))

k. @(R3)+[R5] becomes (X_R5, (OP => IR3_INC))

l. @#1432[R5] becomes (X_R5, (A,1432))

m. @B^4(R9)[R3] becomes (X_R3, IB_R9,4))

n. @W^8(R8)[R5] becomes (X_R5, (IW_R8,8))

o. @L^2(R8)[R4] becomes (X_R4, (IL_R8,2))

p. @B^3[R1] becomes (X_R1, (IB_PC,3))

q. @W^150[R2] becomes (X_R2, (IW_PC,150))

r. @L^100000[R3] becomes (X_R3, (IL_PC,100000))

F.14.1.2 Indexed Operands

Then would be expressed in named notation as:

a. (OP => X_R7, OPND => (OP => IR7))

b. (OP => X_R7, OPND => (OP => R8_INC))

c. (OP => X_R4, OPND => (OP => IMD, W_IMD => 600))

d. (OP => X_R3, OPND => (OP => DEC_R4))

e. (OP => X_R3, OPND => (OP => B_R9, BYTE_DISP => 4))

f. (OP => X_R5, OPND => (OP => W_R8, WORD_DISP => 800))

g. (OP => X_R4, OPND => (OP => L_R8,
                        LONG_WORD_DISP => 34000))

h. (OP => X_R9, OPND => (OP => B_PC, BYTE_DISP => 10))

i. (OP => X_R2, OPND => (OP => W_PC, WORD_DISP => 130))

j. (OP => X_R6, OPND => (OP => L_PC,
                        LONG_WORD_DISP => 35000))

k. (OP => X_R5, OPND => (OP => IR3_INC))

l. (OP => X_R5, OPND => (OP => A, ADDR => 1432))

m. (OP => X_R3, OPND => (OP => IB_R9, BYTE_DISP => 4))

n. (OP => X_R5, OPND => (OP => IW_R8, WORD_DISP => 8))

o. (OP => X_R4, OPND => (OP => IL_R8,
                        LONG_WORD_DISP => 2))

p. (OP => X_R1, OPND => (OP => IB_PC, B_DISP => 3))

q. (OP => X_R2, OPND => (OP => IW_PC, WORD_DISP => 150))

r. (OP => X_R3, OPND => (OP => IL_PC,
                        LONG_WORD_DISP => 100000))

F.14.1.2   Indexed Operands

F.14.1.3 Register Operands

The VAX/VMS Assembler formats for register operands are

         Rn   -- Register mode
        (Rn)   -- Register deferred mode
       -(Rn)   -- Autodecrement mode
        (Rn)+   -- Autoincrement mode
       @(Rn)+   -- Autoincrement deferred mode

where Rn represents a register numbered from 0 to 15.

The general code statement format for register operands is

         (OP => regmode_value)

where regmode_value represents one of the enumerated address mode
specifier range R0 to PC, from Section 6.11.1.

The following are examples of how VAX/VMS Assembler register
mode operands would be written as code statements:

         R7   becomes   (OP => R7)
        (R8)   becomes   (OP => IR8)
       -(R9)   becomes   (OP => DEC_R9)
        (R1)+   becomes   (OP => R1_INC)
       @(R3)+   becomes   (OP => IR3_INC)

F.14.1.4 Byte-Displacement Operands

The VAX/VMS Assembler syntax for the byte-displacement
operands is

        B^d(Rn)   -- Byte-displacement mode
       @B^d(Rn)   -- Byte-displacement deferred mode

where d is the displacement added to the contents of register Rn.
If no register is specified, the program counter is assumed. The
code statement general format for the byte-displacement and
byte-displacement deferred modes is

         (byte_disp_spec, value)

                   or

         (OP => byte_disp_spec, BYTE_DISP => value)

where byte_disp_spec is one of the enumerated address mode
specifiers, range B_R0 to B_PC for byte-displacement or IB_R0 to
IB_PC for byte displacement deferred, from Section 6.11.1. Value
is in the range -128 to 127.

The following are examples of how VAX/VMS Assembler
byte-displacement operands would be written in code statements:

```
B^4(R5)      becomes  (B_R5, 4)      or
                      (OP => B_R5,  BYTE_DISP => 4)
B^200(R5)    becomes  (B_R5, 200)    or
                      (OP => B_R5,  BYTE_DISP => 200)
B^33         becomes  (B_PC, 33)     or
                      (OP => B_PC,  BYTE_DISP => 33)
@B^4(R5)     becomes  (IB_R5, 4)     or
                      (OP => IB_R5, BYTE_DISP => 4)
@B^200(R5)   becomes  (IB_R5, 200)   or
                      (OP => IB_R5, BYTE_DISP => 200)
@B^33        becomes  (IB_PC, 33)    or
                      (OP => IB_PC, BYTE_DISP => 33)
```

F.14.1.5 Word-Displacement Operands

The VAX/VMS Assembler syntax for the word-displacement
operands are

```
W^d(Rn)         -- Word-displacement
@W^d(Rn)        -- Word-displacement deferred
```

where d is the displacement to be added to the contents of
register Rn. If no register is specified, the program counter is
assumed. In code statements, word displacement operands are
represented in general as

$$(word\_disp\_spec, value)$$

or

$$(OP => word\_disp\_spec, WORD\_DISP => value)$$

where word_disp_spec is one of the enumerated address mode
specifiers, range W_R0 to W_PC for word-displacement mode or
IW_R0 or IW_PC for word-displacement deferred mode, from Section
6.11.1. Value is in the range -2**15 to 2**15 - 1.

The following are examples of how VAX/VMS Assembler
word-displacement operands would be written in code statements:

```
W^10(R5)    becomes (W_R5, 10)  or
                    (OP => W_R5, WORD_DISP => 10)
W^20        becomes (W_PC, 20)  or
                    (OP => W_PC  WORD_DISP => 20)
@W^128(R7)  becomes (W_R7, 128) or
                    (OP => IW_R7 WORD_DISP => 128)
@W^324      becomes (W_PC, 324) or
                    (OP => IW_PC WORD_DISP => 324)
```

F.14.1.6 Long_Word-Displacement Operands

The VAX/VMS Assembler general formats for the
long_word-displacement operands is

        L^d(Rn)   -- Long_word-displacement
        @L^d(Rn)   -- Long_word-displacement deferred

where d is the displacement to be added to the register
represented by Rn. Long_word-displacement operands are
represented in code statements by the general format

            (lword_disp_spec, value)

                    or

        (OP => lword_disp_spec, LONG_WORD_DISP => value)

where lword_disp_spec is one of the enumerated address mode
specifiers, range L_R0 to L_PC for long_word-displacement mode or
IL_R0 to IL_PC for long_word-displacement deferred mode, from
Section 6.11.1. Value is in the range -2**31 to 2**31 - 1.

The following are examples of how VAX/VMS Assembler
long_word-displacement operands would be written in code
statements:

    L^1000(R7)   becomes   (L_R7, 1000) or
                           (OP => L_R7, LONG_WORD_DISP => 1000)
    L^25000   becomes   (L_PC, 25000) or
                           (OP => L_PC, LONG_WORD_DISP => 25000)
    @L^1000(R9) becomes   (IL_R9, 1000) or
                           (OP => IL_R9, LONG_WORD_DISP => 1000)
    @L^3500   becomes   (IL_PC, 3500) or
                           (OP => IL_PC, LONG_WORD_DISP => 3500)

F.14.2 The CASE Statement

The VAX case statements (mnemonics CASEB, CASEW, and CASEL)
have the following general symbolic form

        opcode selector.rx, base.rx, limit.rx,
                            displ[0].bw, .. , displ[limit].bw

where x is dependent upon the opcode as to whether the operand is
of type BYTE, WORD, or LONG_WORD. Displ[0].bw, .. ,
displ[limit].bw is a list of displacements to which to branch.
Case statements would be written as code statements as:

```
        BYTE_OP_CODE(OP => case_opcode,"case_opcode"_1=> operand,
                     "case_opcode"_2 => operand,
                     "case_opcode"_3 => case_operand)
```

where case_opcode is one of CASEB, CASEW, or CASEL. The type of
operand and case operand are as indicated in the opcode (BYTE,
WORD, or LONG_WORD). A case_operand is a special case operand of
the form:

```
        case_operand => (case_limit_address_mode, (case_enum))
```

                              or

```
        case_operand => (LIMIT => case_limit_address_mode,
            (CASES=>case_enum))
```

if case_limit_address_mode is one of the short literal address
specifiers If case_limit_address_mode is the mode specifier
IMD, the case_operand takes the form:

```
        case_operand => (IMD, (case_limit, (case_enum)))
```

                              or

```
        case_operand => (LIMIT => IMD, CASE_LIST =>
            (LIMIT => case_limit, (CASES => case_enum)))
```

where case_operand is one of BYTE_CASE_OPERAND,
WORD_CASE_OPERAND, or LONG_WORD_CASE_OPERAND. The
case_limit_address_mode is one of the short literal mode
specifiers or the mode specifier IMD. Case_enum is a list of
branch addresses. The branch addresses must be of type WORD.
The case_limit is a value of the type indicated by the
case_opcode.

    Some examples of case statements written as code statements
are:

```
<<START>>    BYTE_OP_CODE(CASEB, (OP =>R3, (IMD, 5), (IMD
             (2,(15,30,45))))); -- Case statement using
                                 -- immediate mode.

S2           BYTE_OP_CODE(CASEW, (OP => (W_PC, 10)), (IMD, 100),
             (L2,(10,20,30)));   -- Case statement using
                                 -- short literal mode.
```

F.14.2 The CASE Statement

F.14.3   VAX Data

Constant values such as absolute addresses or displacements
may be entered into the code stream with any of these nine
statements:

```
BYTE_VALUE'(byte)
WORD_VALUE'(word)
LONG_WORD_VALUE'(long_word)
QUADWORD_VALUE'(quadword)
OCTAWORD_VALUE'(octaword)
FLOAT_VALUE'(float)
LONG_FLOAT_VALUE'(long_float)
G_FLOAT_VALUE'(g_float)
H_FLOAT_VALUE'(h_float)
```