REPORT DOCUMENTATION PAGE

Form Approve !
OPM No. 0704-0188

AD-A234 380

. REPORT DATE

3. REPORT TYPE AND DATES COVERED
Final: Dec 17, 1990 to Mar 1, 1991

4. TITLE AND SUBTITLE
Ada Compiler Validation Summary Report: DDC International A/S, DACS VAX/VMS to 68020 Bare Cross Compiler System, Version 4.6, microVAX 3100 (Host) to MOTOROLA MVME133 (Target), 901129S1.11051

5. FUNDING NUMBERS

6. AUTHOR(S)
National Institute of Standards and Technology
Gaithersburg, MD
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Institute of Standards and Technology
National Computer Systems Laboratory
Bldg. 255, Rm A266
Gaithersburg, MD 20899   USA

8. PERFORMING ORGANIZATION REPORT NUMBER
NIST90DDC500_2_1.11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, RM 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

DDC International A/S, DACS VAX/VMS to 68020 Bare Cross Compiler System, Version 4.6, Gaithersburg, MD, microVAX 3100 running VMS Version 5.3(Host) to MOTOROLA MVME133 board (Target), ACVC 1.11.

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	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

91 4   01   066

AVF Control Number: NIST90DDC500_2_1.11
DATE COMPLETED
    BEFORE ON-SITE: December 17, 1990
    AFTER   ON-SITE: November 30, 1990
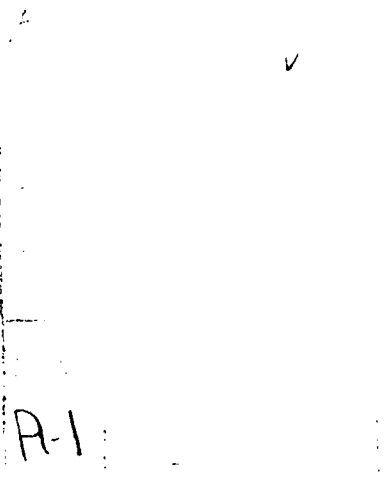   REVISIONS:

Ada   COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901129S1.11051
DDC International A/S
DACS VAX/VMS to 68020 Bare Cross Compiler System, Version 4.6
microVAX 3100 => MOTOROLA MVME133

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

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on November 29, 1990.

Compiler Name and Version: DACS VAX/VMS to 68020 Bare Cross Compiler System Version 4.6
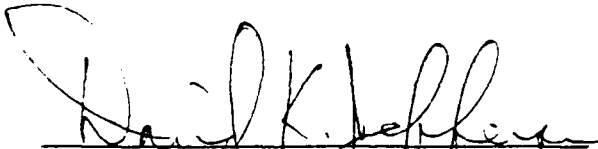
Host Computer System: microVAX 3100 running VMS Version 5.3
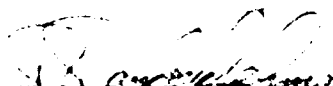
Target Computer System: MOTOROLA MVME133 board

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 901129S1.11051 is awarded to DDC International A/S. This certificate expires on March 01, 1993.

This report has been reviewed and is approved.

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

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

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

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

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

DECLARATION OF CONFORMANCE

Customer and Certificate Awardee: DDC International A/S

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

ACVC Version: 1.11

Ada Implementation:

   Compiler Name and Version: DACS. VAX/VMS to 68020 Bare Cross
                                Compiler System Version 4.6

   Host Computer System: microVAX 3100 running VMS Version
                                5.3

   Target Computer System: MOTOROLA MVME133 board

Declaration:

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

_____   _____
Customer Signature   Date
Company
Title

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
> Institute for Defense Analyses
> 1801 North Beauregard Street
> Alexandria VA 22311

1.2 REFERENCES

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

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

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

1.3 ACVC TEST CLASSES

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

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

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

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

withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

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

1.4   DEFINITION OF TERMS

Ada Compiler
The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.

Ada Compiler Validation Capability (ACVC)
The means for testing compliance of Ada implementations, Validation consisting of the test suite, the support programs, the ACVC Capability user's guide and the template for the validation summary (ACVC) report.

Ada Implementation
An Ada compiler with its host computer system and its target computer system.

Ada Validation Facility (AVF)
The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.

Ada Validation Organization (AVO)
The part of the certification body that provides technical guidance for operations of the Ada certification system.

Compliance of an Ada Implementation
The ability of the implementation to pass an ACVC version.

Computer System
A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity
Fulfillment by a product, process or service of all requirements specified.

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 81 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 90-10-12. ·

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026A
B83026B	C83041A	B85001L	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

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

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)

●

C45641L..Y (14 tests)   C46012L..Z (15 tests)

C24113I..K (3 TESTS) USE A LINE LENGTH IN THE INPUT FILE WHICH
EXCEEDS 126 CHARACTERS.

The following 21 tests check for the predefined type
LONG_INTEGER:

         C35404C   C45231C   C45304C   C45411C   C45412C
         C45502C   C45503C   C45504C   C45504F   C45611C
         C45612C   C45613C   C45614C   C45631C   C45632C
         B52004D   C55B07A   B55B09C   B86001W   C86006C
         CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a
predefined integer type with a name other than INTEGER,
LONG_INTEGER, or SHORT_INTEGER.

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

C35713D AND B86001Z CHECK FOR A PREDEFINED FLOATING-POINT TYPE
WITH A NAME OTHER THAN FLOAT, LONG_FLOAT, OR SHORT_FLOAT.

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

C45624A CHECKS THAT THE PROPER EXCEPTION IS RAISED IF
MACHINE_OVERFLOWS IS FALSE FOR FLOATING POINT TYPES WITH DIGITS
5. FOR THIS IMPLEMENTATION, MACHINE_OVERFLOWS IS TRUE.

C45624B CHECKS THAT THE PROPER EXCEPTION IS RAISED IF
MACHINE_OVERFLOWS IS FALSE FOR FLOATING POINT TYPES WITH DIGITS
6. FOR THIS IMPLEMENTATION, MACHINE_OVERFLOWS IS TRUE.

C4A013B CONTAINS THE EVALUATION OF AN EXPRESSION INVOLVING
'MACHINE_RADIX APPLIED TO THE MOST PRECISE FLOATING-POINT TYPE.
THIS EXPRESSION WOULD RAISE AN EXCEPTION. SINCE THE EXPRESSION
MUST BE STATIC, IT IS REJECTED AT COMPILE TIME.

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

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

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

CA2009C, CA2009F, BC3204C, AND BC3205D THESE TESTS INSTANTIATE GENERIC UNITS BEFORE THEIR BODIES ARE COMPILED. THIS IMPLEMENTATION CREATES A DEPENDENCE ON GENERIC UNIT AS ALLOWED BY AI-00408 & AI-00530 SUCH THAT A THE COMPILATION OF THE GENERIC UNIT BODIES MAKES THE INSTANTIATING UNITS OBSOLETE.

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

The following 265 tests check for sequential, text, and direct access files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	
CE2120A..B (2)	CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)
CE2203A	CE2204A..D (4)	CE2205A	CE2206A
CE2208B	CE2401A..C (3)	EE2401D	CE2401E..F (2)
EE2401G	CE2401H..L (5)	CE2403A	CE2404A..B (2)
CE2405B	CE2406A	CE2407A..B (2)	CE2408A..B (2)
CE2409A..B (2)	CE2410A..B (2)	CE2411A	CE3102A..C (3)
CE3102F..H (3)	CE3102J..K (2)	CE3103A	CE3104A..C (3)
CE3106A..B (2)	CE3107B	CE3108A..B (2)	CE3109A
CE3110A	CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)
CE3114A..B (2)	CE3115A	CE3116A	CE3119A
EE3203A	EE3204A	CE3207A	CE3208A
CE3301A	EE3301B	CE3302A	CE3304A
CE3305A	CE3401A	CE3402A	EE3402B
CE3402C..D (2)	CE3403A..C (3)	CE3403E..F (2)	CE3404B..D (3)
CE3405A	EE3405B	CE3405C..D (2)	CE3406A..D (4)
CE3407A..C (3)	CE3408A..C (3)	CE3409A	CE3409C..E (3)
EE3409F	CE3410A	CE3410C..E (3)	EE3410F
CE3411A	CE3411C	CE3412A	EE3412C
CE3413A..C (3)	CE3414A	CE3602A..D (4)	CE3603A
CE3604A..B (2)	CE3605A..E (5)	CE3606A..B (2)	
CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)	CE3706D
CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)	CE3806A..B (2)
CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)	CE3905A..C (3)
CE3905L	CE3906A..C (3)	CE3906E..F (2)	

CE2103A..B and CE3107A EXPECT THAT NAME_ERROR IS RAISED WHEN AN ATTEMPT IS MADE TO CREATE A FILE WITH AN ILLEGAL NAME; THIS IMPLEMENTATION DOES NOT SUPPORT THE CREATION OF EXTERNAL FILES AND SO RAISES USE_ERROR.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 67 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	B26001A	B26002A	B26005A	B28003A	B29001A	B33301B
B35101A	B37106A	B37301B	B37302A	B38003A	B38003B	B38009A
B38009B	B55A01A	B61001C	B61001F	B61001H	B61001I	B61001M
B61001R	B61001W	B67001H	B83A07A	B83A07B	B83A07C	B83E01C
B83E01D	B83E01E	B85001D	B85008D	B91001A	B91002A	B91002B
B91002C	B91002D	B91002E	B91002F	B91002G	B91002H	B91002I
B91002J	B91002K	B91002L	B95030A	B95061A	B95061F	B95061G
B95077A	B97103E	B97104G	BA1001A	BA1101B	BC1109A	BC1109C
BC1109D	BC1202A	BC1202F	BC1202G	BE2210A	BE2413A	

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

C83030C

The value used to specify the collection size has been increased from 256 to 324 take alignment into account for:

CD2A83A

CE2103A..B and CE3107A abort with an unhandled execption when USE_ERROR is raised on the attempt to create an external file (see 2.2). The AVO ruled that these tests are to be graded as inapplicable.

2-4

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 with the following additional information:

The DACS VAX/VMS to 68020 Bare Cross Compiler System Version 4.6 was executed on the target MOTOROLA MVME 133 board with the following:

    The MC68020
    The MC68881
    One internal timer
    One serial port
    1MB RAM

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

                    Mr. Svend Bodilsen
                    DDC International A/S
                    Gl. Lundtoftevej 1B
                       DK-2800 Lyngby
                         DENMARK

                Telephone: + 45 42 87 11 44
                Telefax: + 45 42 87 22 17

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

                    Mr. Palle Andersson
                    DDC International A/S
                    Gl. Lundtoftevej 1B
                       DK-2800 LYNGBY
                         Denmark

                Telephone: + 45 42 87 11 44
                Telefax: + 45 42 87 22 17

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	3561	
b)	Total Number of Withdrawn Tests	81	
c)	Processed Inapplicable Tests	528	
d)	Non-Processed I/O Tests	0	
e)	Non-Processed Floating-Point Precision Tests	0	
f)	Total Number of Inapplicable Tests	528	(c+d+e)
g)	Total Number of Tests for ACVC 1.11	4170	(a+b+f)

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 528 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 and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by Digital Equipment Corporation ethernet server system via an RS232 connection to the target board. The results were captured on the host computer system via the same communications process.

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:

        /LIST   /LIBRARY

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

        /CHECK   /CONFIGURATION_FILE = <default file>
        /NOPROGRESS   /NOSAVE_SOURCE   /NOXREF

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 ontains 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 126 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	126
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

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

Macro Parameter	Macro Value
ACC_SIZE	: 32
ALIGNMENT	: 4
COUNT_LAST	: 2_147_483_647
DEFAULT_MEM_SIZE	: 2_097_152
DEFAULT_STOR_UNIT	: 8
DEFAULT_SYS_NAME	: DACS_68020
DELTA_DOC	: 2#1.0#E-31
ENTRY_ADDRESS	: 16#FF#
ENTRY_ADDRESS1	: 16#FE#
ENTRY_ADDRESS2	: 16#FD#
FIELD_LAST	: 35
FILE_TERMINATOR	: ASCII.EM
FIXED_NAME	: NO_SUCH_TYPE
FLOAT_NAME	: NO_SUCH_TYPE
FORM_STRING	: ""
FORM_STRING2	:
"CANNOT_RESTRICT_FILE_CAPACITY"	
GREATER_THAN_DURATION	: 100_000.0
GREATER_THAN_DURATION_BASE_LAST	: 200_000.0
GREATER_THAN_FLOAT_BASE_LAST	: 1.80141E+38
GREATER_THAN_FLOAT_SAFE_LARGE	: 1.0E308
GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	: 1.0E308
HIGH_PRIORITY	: 24
ILLEGAL_EXTERNAL_FILE_NAME1	: \NODIRECTORY\FILENAME
ILLEGAL_EXTERNAL_FILE_NAME2	:
THIS-FILE-NAME-IS-TOO-LONG-FOR-MY-SYSTEM	
INAPPROPRIATE_LINE_LENGTH	: -1
INAPPROPRIATE_PAGE_LENGTH	: -1
INCLUDE_PRAGMA1	:
PRAGMA INCLUDE ("A28006D1.TST")	
INCLUDE_PRAGMA2	:
PRAGMA INCLUDE ("B28006E1.TST")	
INTEGER_FIRST	: -2147483648
INTEGER_LAST	: 2147483647
INTEGER_LAST_PLUS_1	: 2147483648
INTERFACE_LANGUAGE	: AS
LESS_THAN_DURATION	: -75000.0
LESS_THAN_DURATION_BASE_FIRST	: -131073.0
LINE_TERMINATOR	: ASCII.CR
LOW_PRIORITY	: 1
MACHINE_CODE_STATEMENT	:
AA_INSTR'(AA_EXIT_SUBPRGRM,0,0,0,AA_INSTR_INTG'FIRST,0);	
MACHINE_CODE_TYPE	: AA_INSTR
MANTISSA_DOC	: 31

```
MAX_DIGITS                          : 15
MAX_INT                             : 2147483647
MAX_INT_PLUS_1                      : 2147483648
MIN_INT                             : -2147483648
NAME                                : NO_SUCH_TYPE_AVAILABLE
NAME_LIST                           : dacs_68020
NAME_SPECIFICATION1                 :
    DISK$AWC_2:[CROCKETTL.ACVC.DEVELOPMENT]X2120A.;1
NAME_SPECIFICATION2                 :
    DISK$AWC_2:[CROCKETTL.ACVC.DEVELOPMENT]X2120B.;1
NAME_SPECIFICATION3                 :
    DISK$AWC_2:[CROCKETTL.ACVC.DEVELOPMENT]X2120C.;1
NEG_BASED_INT                       : 16#F000000E#
NEW_MEM_SIZE                        : 2097152
NEW_STOR_UNIT                       : 8
NEW_SYS_NAME                        : OUR_ULTRIX_ADA
PAGE_TERMINATOR                     : ASCII.FF
RECORD_DEFINITION                   :
    RECORD INSTR_NO:INTEGER;ARG0:INTEGER;ARG1:INTEGER;
    ARG2:INTEGER;ARG3:INTEGER;ARG4:INTEGER;END RECORD;
RECORD_NAME                         : AA_INSTR
TASK_SIZE                           : 32
TASK_STORAGE_SIZE                   : 1024
TICK                                : 2#1.0#E-14
VARIABLE_ADDRESS                    : 16#ffff00#
VARIABLE_ADDRESS1                   : 16#ffff20#
VARIABLE_ADDRESS2                   : 16#ffff40#
YOUR_PRAGMA                         : NOFLOAT
```

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.

QUALIFIER	DESCRIPTION
/[NO]CHECK	Suppresses run-time constraint checks.
/CONFIGURATION_FILE	Specifies the file used by the compiler.
/LIBRARY	Specifies program library used.
/[NO]LIST	Writes a source listing on the list file.
/OPTIMIZE	Specifies compiler optimization.
/[NO]PROGRESS	Displays compiler progress.
/[NO]SAVE_SOURCE	Inserts source text in program library.
/[NO]XREF	Creates a cross reference listing.
/UNIT	Assigns a specific unit number to the compilation (must be free and in a sublibrary).
/ADA_RTS	For maintenance purpose.
<source-file-spec>	The name of the source file to be compiled.

LINKER OPTIONS

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

QUALIFIER	DESCRIPTION
/LIBRARY	The library used in the link.
/[NO]WARNINGS	Print warnings.
/[NO]STATISTICS	Print statistics.
/[NO]VERIFY	Print information about the link.
/[NO]FLOAT	Control of co-processor use.
/[NO]VECTOR	Interrupt vector description.
/[NO]INTERRUPT_STACK	Interrupt stack description.
/[NO]MAIN_STACK	Main task stack description.
/[NO]TASKS	Number of TCB'S allocated.
/DEFAULTS	Default values for tasks.
/[NO]INTERRUPT_STACK	Interrupt vector description.
/MAIN_TASK	Main task defaults.
/[NO]HEAP	Control of memory management.
/[NO]EXCEPTIONS	Control of exception management.
/BASE	Base address of the link.
/RAM	Description of RAM memory.
/[NO]ROM	Description of ROM memory.
/[NO]ENTRY	Alternative program start label.
/[NO]INIT_FILE	Initialization file name.
/[NO]OPTION_FILE	Linker option file name.
/[NO]KEEP	Do not delete temporary files.
/BOOT	Generate a boot module.
/[NO]USR_LIBRARY	A user supplied object library.
/RTS_STACK_USE	Amount of memory used by RTS.
/[NO]ABSOLUTE	Name of absolute file.
/UCC_LIBRARY	UCC library name.
<unit-name>	The name of the main unit.

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 SHORT_INTEGER is range -32_768 .. 32_767;

type INTEGER is range  -2**31 .. 2**31-1;

type FLOAT is digits 6
     range -3.402823466385E+38 .. 3.402823466385E+38;

type LONG_FLOAT is digits 15
      range -1.7976931348623157E+308 .. 1.7976931348623157E+308;

type DURATION is delta 2#1.0#E-14 range -131_072.0 .. 131_071.0;

end STANDARD;
```

This appendix describes the implementation-dependent characteristics of DACS VAX/VMS to 68020 Bare Ada Cross Compiler System required in Appendix F of the Ada Reference Manual (ANSI/MIL-STD-1815A).

F.1 Implementation-Dependent Pragmas

This section describes all implementation defined pragmas.

F.1.1 PRAGMA INTERFACE_SPELLING

Format: **pragma** INTERFACE_SPELLING(<subprogram-name>, <string>)

Placement: The pragma may be placed as a declarative item.

Restrictions: Pragma **INTERFACE_SPELLING** must be applied to the subprogram denoted by <subprogram-name>. The <string> must be a string literal.

This pragma allows an Ada program to call routines with a name that is not a legal Ada name, the <string> provides the exact spelling of the name of the procedure.

F.1.2 PRAGMA INITIALIZE

Format: **pragma** INITIALIZE(<string_literal>)

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

When the pragma is applied the linker will, as part of the initialization code generate a call to the subprogram with the name <string_literal>. The call will be performed before the elaboration of the Ada program is initiated, with IPL on 7. If several pragmas INITIALIZE are applied to the same program the routines are called in the elaboration order, if several pragmas INITIALIZE are applied to one compilation unit the routines are called in the order of appearance. If several compilations units apply pragma INITIALIZE to the same routine the routine is called once only.

F.1.3   PRAGMA RUNDOWN

Format: **pragma RUNDOWN(<string_literal>)**

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

Similar to pragma initialize, but the subprogram is called after
the main program have terminated and in the reverse order as for
the pragma **INITIALIZE.**

F.1.4   PRAGMA TASKS

Format: **pragma TASKS;**

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

Marks the compilation unit with the task attribute. If the code
that is interfaced by a pragma **INTERFACE** uses any tasking
constructs, the compilation unit must be marked such that the
linker includes the tasking kernel in target programs that
reference the compilation unit.

F.1.5   PRAGMA FLOAT

Format: **pragma FLOAT;**

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

Marks the compilation unit with the float attribute. If the code
that is interfaced by a pragma **INTERFACE** uses any floating point
co-processor instructions, the compilation unit must be marked
such that the linker includes initialization of the floating
point co-processor in target programs that reference the
compilation unit.

F.1.6   PRAGMA INTERRUPTS

Format: **pragma INTERRUPTS;**

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

Marks the compilation unit with the interrupt attribute. If the code that is interfaced by a pragma INTERFACE uses any interrupts, the compilation unit must be marked such that the linker include the interrupt handling in target programs that reference the compilation unit.

F.1.7   PRAGMA STORAGE_MANAGER

Format: **pragma STORAGE MANAGER;**

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

Marks the compilation unit with the heap attribute. If the code that is interfaced by a pragma **INTERFACE** uses the storage manager, the compilation unit must be marked such that the linker include initialization of the storage manager in target programs that reference the compilation unit.

F.1.8   PRAGMA INTERRUPT_HANDLER

The pragma interrupt handler is defined with two formats.

F.1.8.1   PRAGMA INTERRUPT_HANDLER for Task Entries

Format: **pragma INTERRUPT_HANDLER;**

Placement: The   pragma   must   be   placed   as   the   first
                declarative item in the task specification that it
                applies to.

Restrictions: The   task for which the pragma INTERRUPT_HANDLER is
              applied must fulfill the following requirements:

- . address clause must be specified for all entries to the
  _ask.

- All entries of the task must be single entries with no
  parameters.

- The entries must not be called from any tasks.

- No other tasks may be specified in the body of the task.

- The body of the task must consist of a single sequence of
  accept statements for each of the defined interrupts, see
  below:

  ```
  task body fih is
     -- local simple data declaration, no tasks.
  begin
     accept handler1 do
        <statementlist>;
     end handler1;
     accept handler2 do
        <statementlist>;
     end handler2;
   end fih;
  ```

- No other tasking construct than unconditional entry calls may
  appear in the statement list for the select alternatives. The
  execution of a statement list must only execute one
  unconditional entry call.

- Any procedures called from the accept body must not use any
  tasking constructs at all.

- No heap storage must allocated.

- No exception must be proporgated out of the statement list of
  the accept alternatives.

If the restrictions described above are not fulfilled, the
program is erroneous and the result of the execution
unpredictable. The compiler cannot and is not checking all the
restrictions, but attempts to perform as many checks of the
requirements as possible.

The **PRAGMA_INTERRUPT** handler with no parameters allows the user
to implement immediate response to exceptions.

F.1.8.2   PRAGMA INTERRUPT_HANDLER for Procedures

Format: pragma INTERRUPT_HANDLER(procedure-name,integer-
                                          literal);

Placement: The pragma must be placed as a declarative item, in
                 the declarative part defining the specification of
                 the immediately after the procedure specification.

Restrictions: The procedure for which pragma INTERRUPT_HANDLER
                 applies must fulfill the following restrictions:

                 - The integer-literal must be in range 0..255, and
                   must not define an interrupt vector entry to
                   which the processor may generate a trap.

                 - The procedure must not be called anywhere in the
                   application.

                 - No tasks may be declared in the body of the
                   procedure.

                 - The only tasking construct that may be used from
                   the body of the procedure is unconditional entry
                   calls. Several unconditional entry calls may
                   appear in the body of the procedure but the
                   execution of the body must only lead to the
                   execution of one.

                 - Any subprograms called from the procedure must
                   not use any tasking constructs at all.

                 - The procedure must be parameterless.

                 - No heap storage must allocated from the
                   procedure.

                 - Exception must no be propagated out of the
                   procedure.

If the restrictions described above is not fulfilled the program
is erroneous and the result of the execution unpredictable. The
compiler cannot and is not checking all the restrictions, but
attempts to perform as many checks of the requirements as
possible.

The pragma INTERRUPT_HANDLER for procedures defines the named
subprogram to be an interrupt handler for the interrupt vector
entry defined by the integer-literal.

F.1.9   PRAGMA NOFLOAT

Format: pragma NOFLOAT(task-id)

Placement: The pragma must be placed as a declarative item, in
                 the declarative part defining the task type or
                 object denoted by the task-id.

Restrictions: The task(s) denoted by the task-id must not execute
              floating-point co-processor instructions.

This pragma informs the compiler and runtime system that the task
will not execute floating point co-processor instructions.
Consequently the context switch needs not save and restore the
state of the floating point co-processor yielding improved
performance.

F.2   Implementation-Dependent Attributes

No implementation-dependent attributes are defined.

F.3   Package SYSTEM

The spcification for package SYSTEM is as follows:

package SYSTEM is

```
    type ADDRESS              is new  INTEGER;
    subtype PRIORITY          is INTEGER range 1 .. 24;
    type NAME                 is ( dacs_68020 );
    SYSTEM_NAME:              constant NAME   := dacs_68020;
    STORAGE_UNIT:             constant        := 8;
    MEMORY_SIZE:              constant        := 2048 * 1024;
    MIN_INT:                  constant        := -2_147_483_648;
    MAX_INT:                  constant        := 2_147_483_647;
    MAX_DIGITS:               constant        := 15;
    MAX_MANTISSA:             constant        := 31;

    FINE_DELTA:               constant        := 2#1.0#E-31;
    TICK:                     constant        := 2#1.0#E-14;

    type interface_language is (AS);

end SYSTEM;
```

F.4   Representation Clauses

The DACS VAX/VMS to 68020 Cross Compiler fully supports the 'SIZE representation for derived types. The representation clauses that are accepted for non-derived types are described in the following subsections.

F.4.1   Length Clause

Some remarks on implementation dependent behavior of length clauses are necessary:

- When using the **SIZE** attribute for discrete types, the maximum value that can be specified is 32 bits.

- **SIZE** is only obeyed for discrete types when the type is a part of a composite object, e.g. arrays or records.

- Using the **STORAGE_SIZE** attribute for a collection will set an upper limit on the total size of objects allocated in this collection. If further allocation is attempted, the exception **STORAGE_ERROR** is raised.

- When **STORAGE_SIZE** is specified in a length clause for a task, the process stack area will be of the specified size.

F.4.2   Enumeration Representation Clauses

Enumeration representation clauses may specify representations in the range of INTEGER'FIRST + 1..INTEGER'LAST - 1.

F.4.3   Record Representation Clauses

When representation clauses are applied to records the following restrictions are imposed:

- if the component is a record or an unpacked array, it must start at a storage unit boundary (8 bits)

- a record occupies an integral number of storage units (words) (even though a record may have fields that only define an odd number of bytes)

- a record may take up a maximum of 2 Gigabits

- a component must be specified with its proper size (in bits), regardless of whether the component is an array or not.

- if a non-array component has a size which equals or exceeds one storage unit 32-bits the component must start on a storage unit boundary.

- the elements in an array component should always be wholly contained in 32-bits.

If the record type contains components which are not covered by a component clause, they are allocated consecutively after the component with the value. Allocation of a record component without a component clause is always aligned on a storage unit boundary. Holes created because of component clauses are not otherwise utilized by the compiler.

Pragma **PACK** on a record type will attempt to pack the components not already covered by a representation clause (perhaps none). This packing will begin with the small scalar components and larger components will follow in the order specified in the record. The packing begins at the first storage unit after the components with representation clauses.

F.4.3.1 Alignment Clauses

Alignment clauses for records are implemented with the following characteristics:

- If the declaration of the record type is done at the outermost level in a library package, any alignment is accepted.

- If the record declaration is done at a given static level (higher than the outermost library level, i.e., the permanent area), only word alignments are accepted.

- Any record object declared at the outermost level in a library package will be aligned according to the alignment clause specified for the type. Record objects declared elsewhere can only be aligned on a word boundary. If the record type is associated with a different alignment, an error message will be issued.

- If a record type with an associated alignment clause is used in a composite type, the alignment is required to be one word; an error message is issued if this is not the case.

F.5 Implementation-Dependent Names for Implementation — Dependent Components

None defined by the compiler.

F.6 Address Clauses

This section describes the implementation of address clauses and what types of entities may have their address specified by the user.

F.6.1 Objects

Address clauses are supported for scalar and composite objects whose size can be determined at compile time, if the specified address is a literal.

F.6.2 Task Entries

Address clauses are supported for task entries. The following restrictions applies:

-   The affected entries must be defined in a task object only, not a task type.

-   The entries must be single and parameterless.

-   The address specified must not denote an interrupt index which the processor may trap.

-   If the interrupt entry executes floating point co-processor instructions the state of the co-processor must be saved prior to execution of any floating point instructions, and restore before the return.
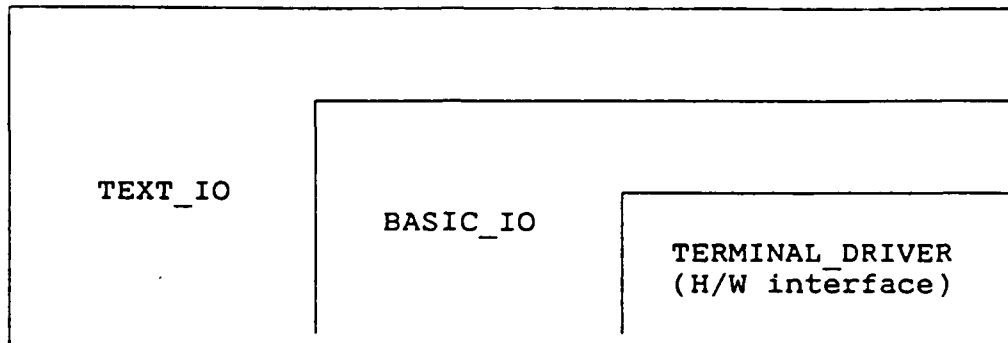
The address specified in the address clause denotes the interrupt vector index.

F.7 Input/Output Packages

In many embedded systems, there is no need for a traditional I/O system, but in order to support testing and validation, DDC-I has developed a small terminal oriented I/O system. This I/O system consists essentially of **TEXT_IO** adapted with respect to handling only a terminal and not file I/O (file I/O will cause a **USE_ERROR**

to be raised) and a low level package called TERMINAL_DRIVER. A BASIC_IO package has been provided for convenience purposes, forming an interface between TEXT_IO and TERMINAL_DRIVER as illustrated in the following figure.

```
+---------------------------------------------------------------+
|                                                               |
|              +------------------------------------------------+
|              |                                                |
|              |                    +---------------------------+
|              |                    |                           |
|  TEXT_IO     |                    |                           |
|              |   BASIC_IO         |   TERMINAL_DRIVER         |
|              |                    |   (H/W interface)        |
|              |                    |                           |
|              |                    |                           |
+---------------------------------------------------------------+
```

The **TERMINAL_DRIVER** package is the only package that is target dependent, i.e., it is the only package that need be changed when changing communications controllers. The actual body of the **TERMINAL_DRIVER** is written in assembly language, but an Ada interface to this body is provided. A user can also call the terminal driver routines directly, i.e. from an assembly language routine. **TEXT_IO** and **BASIC_IO** are written completely in Ada and need not be changed.

BASIC_IO provides a mapping between **TEXT_IO** control characters and **ASCII** as follows:

TEXT_IO	ASCII Character
LINE_TERMINATOR	ASCII.CR
PAGE_TERMINATOR	ASCII.FF
FILE_TERMINATOR	ASCII.EM (ctrl Z)
NEW_LINE	ASCII.LF

The services provided by the terminal driver are:

1) Reading a character from the communications port.

2) Writing a character to the communications port.

F.7.1   Package TEXT_IO

The specification of package TEXT_IO:

```
pragma page;
with BASIC_IO;

with IO_EXCEPTIONS;
package TEXT_IO is

    type FILE_TYPE is limited private;

    type FILE_MODE is (IN_FILE, OUT_FILE);

    type    COUNT is range 0 .. INTEGER'LAST;
    subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
    UNBOUNDED: constant COUNT:= 0; -- line and page length

  -- max. size of an integer output field 2#....#
    subtype FIELD         is INTEGER range 0 .. 35;

    subtype NUMBER_BASE   is INTEGER range 2 .. 16;

    type TYPE_SET is (LOWER_CASE, UPPER_CASE);

pragma PAGE;
    -- File Management
    procedure CREATE (FILE  : in out FILE_TYPE;
                      MODE  : in     FILE_MODE :=OUT_FILE;
                      NAME  : in     STRING    :="";
                      FORM  : in     STRING    :=""
                     );

    procedure OPEN   (FILE  : in out FILE_TYPE;
                      MODE  : in     FILE_MODE;
                      NAME  : in     STRING;
                      FORM  : in     STRING    :=""
                     );

    procedure CLOSE (FILE  : in out FILE_TYPE);
    procedure DELETE (FILE : in out FILE_TYPE);
    procedure RESET (FILE  : in out FILE_TYPE;
                     MODE  : in FILE_MODE);
    procedure RESET (FILE  : in out FILE_TYPE);

    function MODE    (FILE  : in FILE_TYPE) return FILE_MODE;
    function NAME    (FILE  : in FILE_TYPE) return STRING;
    function FORM    (FILE  : in FILE_TYPE) return STRING;

    function IS_OPEN(FILE  : in FILE_TYPE return BOOLEAN;
```

```
pragma PAGE;
     -- control of default input and output files

     procedure SET_INPUT   (FILE : in FILE_TYPE);
     procedure SET_OUTPUT  (FILE : in FILE_TYPE);

     function STANDARD_INPUT     return FILE_TYPE;
     function STANDARD_OUTPUT    return FILE_TYPE;

     function CURRENT_INPUT      return FILE_TYPE;
     function CURRENT_OUTPUT     return FILE_TYPE;
pragma PAGE;
     -- specification of line and page lengths

     procedure SET_LINE_LENGTH  (FILE : in FILE_TYPE;
                                  TO : in COUNT);
     procedure SET_LINE_LENGTH  (TO : in COUNT);

     procedure SET_PAGE_LENGTH  (FILE : in FILE_TYPE;
                                  TO : in COUNT);
     procedure SET_PAGE_LENGTH  (TO : in COUNT);

     function  LINE_LENGTH   (FILE : in FILE_TYPE)
                                  return COUNT;
     function  LINE_LENGTH   return COUNT;

     function  PAGE_LENGTH    (FILE : in FILE_TYPE)
                                  return COUNT;
     function  PAGE_LENGTH    return COUNT;


pragma PAGE;
     -- Column, Line, and Page Control

     procedure NEW_LINE (FILE : in FILE_TYPE;
                         SPACING : in POSITIVE_COUNT := 1);
     procedure NEW_LINE (SPACING : in POSITIVE_COUNT := 1);

     procedure SKIP_LINE (FILE : in FILE_TYPE;
                         SPACING : in POSITIVE_COUNT := 1);
     procedure SKIP_LINE (SPACING : in POSITIVE_COUNT := 1);

     function  END_OF_LINE (FILE : in FILE_TYPE) return BOOLEAN;
     function  END_OF_LINE                       return BOOLEAN;

     procedure NEW_PAGE    (FILE : in FILE_TYPE);
     procedure NEW_PAGE;

     procedure SKIP_PAGE   (FILE : in FILE_TYPE);
     procedure SKIP_PAGE;

     function  END_OF_PAGE (FILE : in FILE_TYPE) return BOOLEAN;
     function  END_OF_PAGE                       return BOOLEAN;
```

```
        function  END_OF_FILE  (FILE : in FILE_TYPE) return BOOLEAN;
        function  END_OF_FILE                         return BOOLEAN;

        procedure SET_COL      (FILE : in FILE_TYPE;
                                TO   : in POSITIVE_COUNT);
        procedure SET_COL      (TO : in POSITIVE_COUNT);

        procedure SET_LINE     (FILE : in FILE_TYPE;
                                TO : in POSITIVE_COUNT);
        procedure SET_LINE     (TO : in POSITIVE_COUNT);

        function  COL          (FILE : in FILE_TYPE)
                                return POSITIVE_COUNT;
        function  COL          return POSITIVE_COUNT;

        function  LINE         (FILE : in FILE_TYPE)
                                return POSITIVE_COUNT;
        function  LINE         return POSITIVE_COUNT;

        function  PAGE         (FILE : in FILE_TYPE)
                                return POSITIVE_COUNT;
        function  PAGE         return POSITIVE_COUNT;

pragma PAGE;
    -- Character Input-Output

        procedure GET  (FILE : in FILE_TYPE; ITEM : out CHARACTER);
        procedure GET  (                     ITEM : out CHARACTER);
        procedure PUT  (FILE : in FILE_TYPE; ITEM :  in CHARACTER);
        procedure PUT  (                     ITEM :  in CHARACTER);

        -- String Input-Output

        procedure GET  (FILE : in FILE_TYPE; ITEM : out CHARACTER);
        procedure GET  (                     ITEM : out CHARACTER);
        procedure PUT  (FILE : in FILE_TYPE; ITEM :  in CHARACTER);
        procedure PUT  (                     ITEM :  in CHARACTER);

        procedure GET_LINE    (FILE : in FILE_TYPE;
                               ITEM : out STRING;
                               LAST : out NATURAL);

        procedure GET_LINE    (ITEM : out STRING;
                               LAST : out NATURAL);

        procedure PUT_LINE    (FILE : in FILE_TYPE;
                               ITEM :  in STRING);
        procedure PUT_LINE    (ITEM :  in STRING);

pragma PAGE;
    -- Generic Package for Input-Output of Integer Types

    generic
```

```
      type NUM is range <>;
   package INTEGER_IO is

      DEFAULT_WIDTH : FIELD            := NUM'WIDTH;
      DEFAULT_BASE  : NUMBER_BASE   :=          10;

      procedure GET   (FILE  : in FILE_TYPE;
                       ITEM  : out NUM;
                       WIDTH : in FIELD := 0);
      procedure GET   (ITEM  : out NUM;
                       WIDTH : in FIELD := 0);

      procedure PUT   (FILE  : in FILE_TYPE;
                       ITEM  :  in NUM;
                       WIDTH :  in FIELD := DEFAULT_WIDTH;
                       BASE  :  in NUMBER_BASE := DEFAULT_BASE);
      procedure PUT   (ITEM  :  in NUM;
                       WIDTH :  in FIELD := DEFAULT_WIDTH;
                       BASE  :  in NUMBER_BASE := DEFAULT_BASE);

      procedure GET   (FROM  : in STRING;
                       ITEM  : out NUM;
                       LAST  : out POSITIVE);

      procedure PUT   (TO    : out STRING;
                       ITEM  :  in NUM;
                       BASE  :  in NUMBER_BASE := DEFAULT_BASE);

   end INTEGER_IO;


pragma PAGE;

   -- Generic Packages for Input-Output of Real Types

generic
   type NUM is digits <>;
package FLOAT_IO is

   DEFAULT_FORE : FIELD :=                   2;
   DEFAULT_AFT  : FIELD := NUM'DIGITS - 1;
   DEFAULT_EXP  : FIELD :=                   3;

   procedure GET   (FILE  : in FILE_TYPE;
                    ITEM  : out NUM;
                    WIDTH : in FIELD := 0);
   procedure GET   (ITEM  : out NUM;
                    WIDTH : in FIELD := 0);

   procedure PUT   (FILE : in FILE_TYPE;
                    ITEM : in NUM;
                    FORE : in FIELD := DEFAULT_FORE;
                    AFT  : in FIELD := DEFAULT_AFT;
```

```
                        EXP   : in FIELD := DEFAULT_EXP);
      procedure PUT   (ITEM  : in NUM;
                        FORE  : in FIELD := DEFAULT_FORE;
                        AFT   : in FIELD := DEFAULT_AFT;
                        EXP   : in FIELD := DEFAULT_EXP);


      procedure GET   (FROM  : in STRING;
                        ITEM  : out NUM;
                        LAST  : out POSITIVE);
      procedure PUT   (TO    : out STRING;
                        ITEM  :  in NUM;
                        AFT   :  in FIELD := DEFAULT_AFT;
                        EXP   :  in FIELD := DEFAULT_EXP);


   end FLOAT_IO;


   pragma PAGE;


   generic
      type NUM is delta <>;
   package FIXED_IO is

      DEFAULT_FORE : FIELD := NUM'FORE;
      DEFAULT_AFT  : FIELD := NUM'AFT;
      DEFAULT_EXP  : FIELD := 0;

      procedure GET   (FILE  : in FILE_TYPE;
                        ITEM  : out NUM;
                        WIDTH : in FIELD := 0);
      procedure GET   (ITEM  : out NUM;
                        WIDTH : in FIELD := 0);


      procedure PUT   (FILE : in FILE_TYPE;
                        ITEM : in NUM;
                        FORE : in FIELD := DEFAULT_FORE;
                        AFT  : in FIELD := DEFAULT_AFT;
                        EXP  : in FIELD := DEFAULT_EXP);

      procedure PUT   (ITEM : in NUM;
                        FORE : in FIELD := DEFAULT_FORE;
                        AFT  : in FIELD := DEFAULT_AFT;
                        EXP  : in FIELD := DEFAULT_EXP);

      procedure GET   (FROM : in STRING;
                        ITEM : out NUM;
                        LAST : out POSITIVE);

      procedure PUT   (TO   : out STRING;
                        ITEM :  in NUM;
                        AFT  :  in FIELD := DEFAULT_AFT;
                        EXP  :  in FIELD := DEFAULT_EXP);
```

```
end FIXED_IO;


pragma PAGE;
    -- Generic Package for Input-Output of Enumeration Types

generic
    type ENUM is (<>);
package ENUMERATION_IO is

    DEFAULT_WIDTH   : FIELD      := 0;
    DEFAULT_SETTING : TYPE_SET := UPPER_CASE;

    procedure GET  (FILE : in FILE_TYPE; ITEM : out ENUM);
    procedure GET  (                     ITEM : out ENUM);

    procedure PUT  (FILE  : FILE_TYPE;
                    ITEM  : in ENUM;
                    WIDTH : in FIELD        := DEFAULT_WIDTH;
                    SET   : in TYPE_SET     := DEFAULT_SETTING);
    procedure PUT  (ITEM  : in ENUM;
                    WIDTH : in FIELD        := DEFAULT_WIDTH;
                    SET   : in TYPE_SET     := DEFAULT_SETTING);

    procedure GET  (FROM : in  STRING;
                    ITEM : out ENUM;
                    LAST : out POSITIVE);

    procedure PUT  (TO   : out STRING;
                    ITEM : in  ENUM;
                    SET  : in  TYPE_SET := DEFAULT_SETTING);

end ENUMERATION_IO;

pragma PAGE;

    -- Exceptions

    STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
    MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
    NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
    USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
    DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
    END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
    DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
    LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

pragma page;
private

    type FILE_TYPE is
     record
        FT : INTEGER := -1;
```

```
      end record;

end TEXT_IO;
```

F.7.2   Package IO_EXCEPTIONS

The specification of the package IO_EXCEPTIONS:

```
package IO_EXCEPTIONS is

    STATUS_ERROR : exception;
    MODE_ERROR   : exception;
    NAME_ERROR   : exception;
    USE_ERROR    : exception;
    DEVICE_ERROR : exception;
    END_ERROR    : exception;
    DATA_ERROR   : exception;
    LAYOUT_ERROR : exception;

end IO_EXCEPTIONS;
```

F.7.3   Package BASIC_IO

The specification of package BASIC_IO:

```
with IO_EXCEPTIONS;

package BASIC_IO is

    type count is range 0 .. integer'last;

    subtype positive_count is count range 1 .. count'last;


    function get_integer return string;

    -- Skips any leading blanks, line terminators or page
    -- terminators. Then reads a plus or a minus sign if
    -- present, then reads according to the syntax of an
    -- integer literal, which may be based.  Stores in item
    -- returns a string containing an optional sign and an
    -- integer literal.
    --
    -- The exception DATA_ERROR is raised if the sequence
    -- of characters does not correspond to the syntax
    -- described above.
    --
```

```
-- The exception END_ERROR is raised if the file terminator
-- is read.  This means that the starting sequence of an
-- integer has not been met.
--
-- Note that the character terminating the operation must
-- be available for the next get operation.
--


function get_real return string;

-- Corresponds to get_integer except that it reads according
-- to the syntax of a real literal, which may be based.


function get_enumeration return string;

-- Corresponds to get_integer except that it reads according
-- to the syntax of an identifier, where upper and lower
-- case letters are equivalent to a character literal
-- including the apostrophes.


function get_item (length : in      integer) return string;

-- Reads a string from the current line and stores it in
-- item.  If the remaining number of characters on the
-- current line is less than length then only these
-- characters are returned.  The line terminator is not
-- skipped.


procedure put_item (item : in      string);

-- If the length of the string is greater than the current
-- maximum line (linelength),  the exception LAYOUT_ERROR
-- is raised.
--
-- If the string does not fit on the current line a line
-- terminator is output, then the item is output.


-- Line and page lengths - [DOD-83] 14.3.3.
--

procedure set_line_length (to   : in      count);


procedure set_page_length (to   : in      count);


function line_length return count;
```

```
function page_length return count;


-- Operations on columns, lines and pages - [DOD-83] 14.3.4.
--

procedure new_line;

procedure skip_line;

function end_of_line return boolean;

procedure new_page;

procedure skip_page;

function end_of_page return boolean;

function end_of_file return boolean;

procedure set_col (to    : in      positive_count);

procedure set_line (to    : in      positive_count);

function col return positive_count;

function line return positive_count;

function page return positive_count;


-- Character and string procedures.
-- Corresponds to the procedures defined in [DOD-83] 14.3.6.

procedure get_character (item :     out character);

procedure get_string (item :     out string);

procedure get_line (item :     out string;
```

```
                 last :     out natural );

   procedure put_character (item : in      character);


   procedure put_string (item : in      string);


   procedure put_line (item : in      string);



   --  exceptions:

   USE_ERROR      : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR      : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR     : exception renames IO_EXCEPTIONS.DATA_ERROR;
   LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

end BASIC_IO;
```

F.7.4   Package TERMINAL_DRIVER

The specification of package TERMINAL_DRIVER:

```
package terminal_driver is

   procedure put_character(ch : character);

   procedure flush;

   function get_character return character;

   procedure purge;

private

   pragma interface (AS, put_character);
   pragma interface_spelling(put_character, "Ada_UCC_G$PutByte");

   pragma interface (AS, get_character);
   pragma interface_spelling(get_character, "Ada_UCC_G$GetByte");

   pragma interface (AS, flush);
   pragma interface_spelling(flush, "Ada_UCC_G$FlushOutput");

   pragma interface (AS, purge);
   pragma interface_spelling(purge, "Ada_UCC_G$PurgeInput");
```

```
    pragma initialize("Ada_UCC_G$InitIO");
    pragma rundown    ("Ada_UCC_G$CloseIO");

end terminal_driver;
```

F.7.5   Package SEQUENTIAL_IO

```
-- Source code for SEQUENTIAL_IO

pragma PAGE;

with IO_EXCEPTIONS;

generic

    type ELEMENT_TYPE is private;

package SEQUENTIAL_IO is

  type FILE_TYPE is limited private;

  type FILE_MODE is (IN_FILE, OUT_FILE);


pragma PAGE;
-- File management

    procedure CREATE(FILE : in out FILE_TYPE;
                     MODE : in      FILE_MODE := OUT_FILE;
                     NAME : in      STRING    := "";
                     FORM : in      STRING    := "");

    procedure OPEN  (FILE : in out FILE_TYPE;
                     MODE : in      FILE_MODE;
                     NAME : in      STRING;
                     FORM : in      STRING := "");

    procedure CLOSE (FILE : in out FILE_TYPE);

    procedure DELETE(FILE : in out FILE_TYPE);

    procedure RESET (FILE : in out FILE_TYPE;
                     MODE : in      FILE_MODE);

    procedure RESET (FILE : in out FILE_TYPE);

    function MODE    (FILE : in FILE_TYPE) return FILE_MODE;

    function NAME    (FILE : in FILE_TYPE) return STRING;

    function FORM    (FILE : in FILE_TYPE) return STRING;
```

```
    function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;


pragma PAGE;
-- input and output operations


    procedure READ   (FILE : in      FILE_TYPE;
                       ITEM :     out ELEMENT_TYPE);

    procedure WRITE (FILE : in FILE_TYPE;
                     ITEM : in ELEMENT_TYPE);

    function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

pragma PAGE;
-- exceptions


    STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
    MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
    NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
    USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
    DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
    END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
    DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

pragma PAGE;
private

    type FILE_TYPE is new INTEGER;

end SEQUENTIAL_IO;
```

F.8   Machine Code Insertion

Machine code insertion is allowed using the instruction defined in package MACHINE_CODE. All arguments given in the code statement aggregate must be static.

The machine language defined in package MACHINE_CODE is not VAX assembler, but rather Abstract A-code which is an intermediate language used be the compiler.