

# AD-A234 323

## ENTATION PAGE

Form Approved  
OPM No. 0704-0188

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

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE	3. REPORT TYPE AND DATES COVERED Final: Feb 21 1991 to Mar 01, 1993	
4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: DDC International A/S, DACS VAX/VMS Native Ada Compiler System, Version 4.6, VAX 8530 (Host & Target), 901129S1.11050			5. FUNDING NUMBERS	
6. AUTHOR(S) National Institute of Standards and Technology Gaithersburg, MD USA			8. PERFORMING ORGANIZATION REPORT NUMBER NIST90DDC500_1_1.11	
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			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
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 Native Ada Compiler System, Version 4.6, Gaithersburg, MD, VAX 8530 running VMS Version 5.3 (Host & 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	
17 SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED			16. PRICE CODE	
18 SECURITY CLASSIFICATION UNCLASSIFIED		19 SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED		20. LIMITATION OF ABSTRACT

AVF Control Number: NIST90DDC500\_1\_1.11  
DATE COMPLETED  
BEFORE ON-SITE: February 21, 1991  
AFTER ON-SITE: November 30, 1990  
REVISIONS:

Ada COMPILER  
VALIDATION SUMMARY REPORT:  
Certificate Number: 901129S1.11050  
DDC International A/S  
DACS VAX/VMS Native Ada Compiler System, Version 4.6  
VAX 8530 => VAX 8530

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

AVF Control Number: NIST90DDC500\_1\_1.11

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 Native Ada Compiler System, Version 4.6

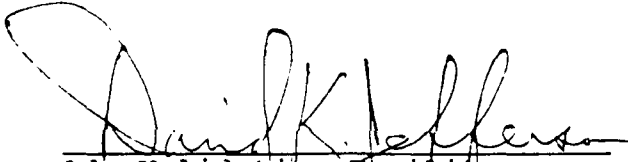
Host Computer System: VAX 8530 running VMS Version 5.3

Target Computer System: VAX 8530 running VMS Version 5.3

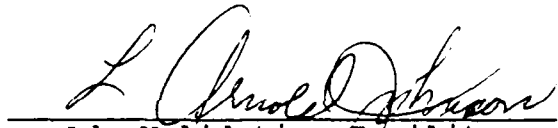
A more detailed description of this Ada implementation is found in section 3.1 of this report.

As a result of this validation effort, Validation Certificate 901129S1.11050 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:

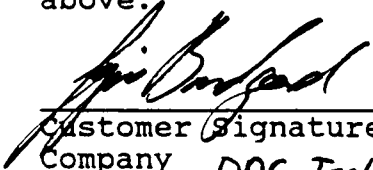
Compiler Name and Version: DACS VAX/VMS Native Ada Compiler  
System, Version 4.6

Host Computer System: VAX 8530 running VMS Version 5.3

Target Computer System: VAX 8530 running VMS Version 5.3

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

Company DDC International A/S  
Title Project Manager

1990-11-29  
Date

## TABLE OF CONTENTS

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

## CHAPTER 1

### INTRODUCTION

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

#### 1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service  
5285 Port Royal Road  
Springfield VA 22161

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

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

#### 1.2 REFERENCES

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

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

### 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 SPRT13, 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 SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK\_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK\_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued. Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

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

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

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

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

#### 1.4 DEFINITION OF TERMS

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



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

## CHAPTER 2

### IMPLEMENTATION DEPENDENCIES

#### 2.1 WITHDRAWN TESTS

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

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

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\_OVERFLOW IS FALSE FOR FLOATING POINT TYPES WITH DIGITS 5. FOR THIS IMPLEMENTATION, MACHINE\_OVERFLOW IS TRUE.

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

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

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

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 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	SFQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	LIRECT_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
CE3102E	CREATE	IN_FILE	TEXT_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

CE2107B..E (4 TESTS), CE2107L, CE2110B AND CE2111D ATTEMPT TO ASSOCIATE MULTIPLE INTERNAL FILES WITH THE SAME EXTERNAL FILE WHEN ONE OR MORE FILES IS WRITING FOR SEQUENTIAL FILES. THE PROPER EXCEPTION IS RAISED WHEN MULTIPLE ACCESS IS ATTEMPTED.

CE2107G..H (2 TESTS), CE2110D, AND CE2111H ATTEMPT TO ASSOCIATE MULTIPLE INTERNAL FILES WITH THE SAME EXTERNAL FILE WHEN ONE OR MORE FILES IS WRITING FOR DIRECT FILES. THE PROPER EXCEPTION IS RAISED WHEN MULTIPLE ACCESS IS ATTEMPTED.

CE2108B, CE2108D, and CE3112B USE THE NAME FUNCTION TO CHECK THAT

TEMPORARY SEQUENTIAL DIRECT, AND TEXT FILES ARE NOT ACCESSIBLE AFTER COMPLETION OF THE PROGRAM THAT CREATES THEM; FOR THIS IMPLEMENTATION TEMPORARY FILES ARE UNNAMED.

CE2203A CHECKS FOR SEQUENTIAL IO THAT WRITE RAISES USE\_ERROR IF THE CAPACITY OF THE EXTERNAL FILE IS EXCEEDED. THIS IMPLEMENTATION CANNOT RESTRICT FILE CAPACITY.

EE2401D CHECKS WHETHER READ, WRITE, SET\_INDEX, INDEX, SIZE, AND END\_OF\_FILE ARE SUPPORTED FOR DIRECT FILES FOR AN UNCONSTRAINED ARRAY TYPE. USE\_ERROR WAS RAISED FOR DIRECT CREATE. THE MAXIMUM ELEMENT SIZE SUPPORTED FOR DIRECT\_IO IS 32K BYTES.

CE2403A CHECKS FOR DIRECT\_IO THAT WRITE RAISES USE\_ERROR IF THE CAPACITY OF THE EXTERNAL FILE IS EXCEEDED. THIS IMPLEMENTATION CANNOT RESTRICT FILE CAPACITY.

CE3111B, CE3111D..E (2 TESTS), CE3114B, AND CE3115A ATTEMPT TO ASSOCIATE MULTIPLE INTERNAL FILES WITH THE SAME EXTERNAL FILE WHEN ONE OR MORE FILES IS WRITING FOR TEXT FILES. THE PROPER EXCEPTION IS RAISED WHEN MULTIPLE ACCESS IS ATTEMPTED.

CE3413B CHECKS THAT PAGE RAISES LAYOUT\_ERROR WHEN THE VALUE OF THE PAGE NUMBER EQUALS COUNT'LAST. THE VALUE OF COUNT'LAST IS GREATER THAN 150000 AND THE CHECKING OF THIS OBJECTIVE IS IMPRACTICAL.

## 2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 65 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 C86007A

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

CD2A83A

## 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. Jorgen Bundgaard  
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:

In the U.S.A.:

Mr. Mike Turner  
DDC-I, Inc.  
9630 North 25th Avenue  
Suite #118  
Phoenix, Arizona 85021  
Telephone: 602-944-1883  
Telefax: 602-944-3253

Mailing address:

P.O. Box 37767  
Phoenix, Arizona 85069-7767

In the rest of the world:

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	3805	
b) Total Number of Withdrawn Tests	81	
c) Processed Inapplicable Tests	284	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	284	(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 284 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/target computer system, as appropriate. The results were captured 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:

```
/LIST /OPTIMIZE /LIBRARY
```

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

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

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 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	: VAX11
DELTA_DOC	: 2#1.0#E-31
ENTRY_ADDRESS	: FCNDECL.ENTRY_ADDRESS
ENTRY_ADDRESS1	: FCNDECL.ENTRY_ADDRESS1
ENTRY_ADDRESS2	: FCNDECL.ENTRY_ADDRESS2
FIELD_LAST	: 67
FILE_TERMINATOR	: ' '
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	: 15
ILLEGAL_EXTERNAL_FILE_NAME1	: #1\NODIRECTORY\FILENAME
ILLEGAL_EXTERNAL_FILE_NAME2	:
THIS_FILE_NAME_FAR_IS_TOO_LONG_FOR_THIS_IMPLEMENTATION	:
INAPPROPRIATE_LINE_LENGTH	: 2049
INAPPROPRIATE_PAGE_LENGTH	: -1
INCLUDE_PRAGMA1	:
PRAGMA INCLUDE ("A28006D1.TST")	:
INCLUDE_PRAGMA2	:
PRAGMA INCLUDE ("B28006E1.TST")	:
INTEGER_FIRST	: -32768
INTEGER_LAST	: 32767
INTEGER_LAST_PLUS_1	: 32_768
INTERFACE_LANGUAGE	: VMS
LESS_THAN_DURATION	: -100_000.0
LESS_THAN_DURATION_BASE_FIRST	: -200_000.0
LINE_TERMINATOR	: ' '
LOW_PRIORITY	: 0
MACHINE_CODE_STATEMENT	:
AA INSTR'(AA EXIT SUBPRGRM,0,0,0,0,0);	:
MACHINE_CODE_TYPE	: AA_INSTR
MANTISSA_DOC	: 31

```

MAX_DIGITS                : 15
MAX_INT                   : 2147483647
MAX_INT_PLUS_1           : 2_147_483_648
MIN_INT                   : -2147483648
NAME                      : NO_SUCH_TYPE_AVAILABLE
NAME_LIST                 : VAX11
NAME_SPECIFICATION1      :
    ACVC_VAX$DEVICE:[CHECK.CTESTS]X2120A.DAT;1
NAME_SPECIFICATION2      :
    ACVC_VAX$DEVICE:[CHECK.CTESTS]X2120B.DAT;1
NAME_SPECIFICATION3      :
    ACVC_VAX$DEVICE:[CHECK.CTESTS]X3119A.DAT;1
NEG_BASED_INT            : 16#F000000E#
NEW_MEM_SIZE              : 2_097_152
NEW_STOR_UNIT             : 8
NEW_SYS_NAME              : VAX11
PAGE_TERMINATOR           : ' '
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                      : 0.000_001
VARIABLE_ADDRESS         : FCNDECL.VARIABLE_ADDRESS
VARIABLE_ADDRESS1        : FCNDECL.VARIABLE_ADDRESS1
VARIABLE_ADDRESS2        : FCNDECL.VARIABLE_ADDRESS2
YOUR_PRAGMA               : INTERFACE_SPELLING

```

## 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.

<u>QUALIFIER</u>	<u>DESCRIPTION</u>
/[NO]CHECK	Generates 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.
/[NO]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).
/[NO]TRACEBACK <source-file-spec>	Generate table to print trace of calls. 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.

<u>QUALIFIER</u>	<u>DESCRIPTION</u>
/LIBRARY	The library used in the link.
/[NO]LOG	Produce a log file.
/OBJECT=	Additional object files or object libraries.
/OPTIONS=	Options to be passed on to the native linker.
<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 -128 .. 127;
```

```
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 -16#7.FFFF_C#E31 .. 16#7.FFFF_C#E31;
```

```
type LONG_FLOAT is digits 15  
  range -16#7.FFFF_FFFF_FFFF#E255 .. 16#7.FFFF_FFFF_FFFF#E255;
```

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

```
end STANDARD;
```

Appendix  
User's Guide

F Appendix F of the Ada Reference Manual

F.1 Introduction

This appendix describes the implementation-dependent characteristics of the DDC-I VAX/VMS Ada Compiler, as required in the Appendix F frame of the Ada Reference Manual (ANSI/MIL-STD-1815A).

F.2 Implementation-Dependent Pragmas

There is one implementation defined pragma: pragma interface\_spelling. See the description in Section 4.6.2.2.

F.3 Implementation-Dependent Attributes

No implementation-dependent attributes are defined for the VAX/VMS version.

F.4 Package SYSTEM

The specification of the package SYSTEM:

package SYSTEM is

```
    type ADDRESS          is access INTEGER;
    subtype PRIORITY      is INTEGER range 0..15;
    type NAME              is (VAX11);
    SYSTEM_NAME:          constant NAME    := VAX11;
    STORAGE_UNIT:         constant        := 8;
    MEMORY_SIZE:          constant        := 2048 * 1024;
    MIN_INT:              constant        := -2_147_483_647-1;
    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        := 0.000_001;
```

type interface\_language is (VMS);

end SYSTEM;



Appendix  
User's Guide

F.5 Representation Clauses

The representation clauses that are accepted are described below. Note that representation specifications can be given on derived types too.

F.5.1 Length Clauses

Four kinds of length clauses are accepted.

Size specifications:

The size attribute for a type T is accepted in the following cases :

- If T is a discrete type then the specified size must be greater than or equal to the number of bits needed to represent a value of the type, and less than or equal to 32. Note that when the number of bits needed to hold any value of the type is calculated, the range is extended to include 0 if necessary, i.e. the range 3..4 cannot be represented in 1 bit, but needs 3 bits.
- If T is a fixed point type then the specified size must be greater than or equal to the smallest number of bits needed to hold any value of the fixed point type, and less than 32 bits. Note that the Reference Manual permits a representation, where the lower bound and the upper bound is not representable in the type. Thus the type  

```
type fix is delta 1.0 range -1.0 .. 7.0;
```

is representable in 3 bits. As for discrete types the number of bits needed for a fixed point type is calculated using the range of the fixed point type possibly extended to include 0.0.
- If T is a floating point type, an access type or a task type, the specified size must be equal to the number of bits used to represent values of the type (floating points: 32 or 64, access types : 32 bits and task type : 32 bits).
- If T is a record type the specified size must be greater than or equal to the minimal number of bits used to represent values of the type per default.
- If T is an array type the size of the array must be static, i.e. known at compile time and the specified size must be equal to the minimal number of bits used to represent values of the type per default.

Appendix  
User's Guide

Collection size specifications:

Using the STORAGE\_SIZE attribute on an access type will set an upper limit on the total size of objects allocated in the collection allocated for the access type. If further allocation is attempted, the exception STORAGE\_ERROR is raised. The specified storage size must be less than or equal to INTEGER'LAST.

Task storage size :

When the STORAGE\_SIZE attribute is given on a task type, the task stack area will be of the specified size. There is no upper limit on the given size.

Small specifications :

Any value of the SMALL attribute less than the specified delta for the fixed point type can be given.

F.5.2 Enumeration Representation Clauses

Enumeration representation clauses may specify representation in the range of INTEGER'FIRST+1 .. INTEGER'LAST-1. An enumeration representation clause may be combined with a length clause. If an enumeration representation clause has been given for a type the representational values are considered when the number of bits needed to hold any value of the type is evaluated. Thus the type

```
type enum is (a,b,c);  
for enum use (1,3,5);
```

needs 3 bits not 2 bits to represent any value of the type.

F.5.3 Record Representation Clauses

When component clauses are applied to a record type the following restrictions are imposed :

- All values of the component type must be representable within the specified number of bits in the component clause.

Appendix  
User's Guide

- If the component type is either a discrete type other than `LONG_INTEGER`, a fixed point type, or an array type with a discrete type other than `LONG_INTEGER`, or a fixed point type as element type, then the component is packed into the specified number of bits (see however the restriction in the paragraph above), and the component may start at any bitboundary.
- If the component type is not one of the types specified in the paragraph above, it must start at a storage unit boundary, a storage unit being 8 bits, and the default size calculated by the compiler must be given as the bit width, i.e. the component must be specified as

component at N range 0 .. 8 \* M-1

where N specifies the relative storage unit number (0,1,...) from the beginning of the record, and M the required number of storage units (1,2,...).

- The maximum bit width for components of scalar types is 32.

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.

#### F.5.3.1 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 long 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 long word boundary. If the record type has been associated a different alignment, an error message will be issued.

Appendix  
User's Guide

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

**F.5.3.2 Implementation-Dependent Names for Implementation-Dependent Components**

None defined by the compiler.

**F.6 Address Clauses**

Address clauses for objects are accepted. Note that the definition of the type SYSTEM.ADDRESS (see F.4) does not allow "address literals" to be specified. Address clauses will therefore look like

for X use at Y' ADDRESS;

or

for VAR use at FUNCTION\_RETURNING\_AN\_ADDRESS;

Address clause on subprogram, tasks, package or entries are not supported.

**F.7 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 by the compiler.

**F.8 Interface to Other Languages**

See chapter 14.

Appendix  
User's Guide

F.9 Unchecked Conversion

Unchecked conversion is only allowed between objects of the same "size". In this context the "size" of an array is equal to that of two access values and the "size" of a packed array is equal to two access values and an integer. This is the only restriction imposed on unchecked conversion.

F.10 Input-Output Packages

The implementation supports all requirements of the Ada language. It is an effective interface to the VAX/VMS file system, and in case of text input-output also an effective interface to the VAX/VMS terminal driver.

This section describes the functional aspects of the interface to the VAX/VMS file system and terminal driver. Certain portions of this section is of special interest to the system programmer who needs to control VAX/VMS specific Input-Output characteristics via Ada programs.

The section is organised as follows.

Subsection numbers refer to the equivalent subsections in Chapter 14 of the [DoD 83]. Only subsections of interest to this section are included.

The Ada Input-Output concept as defined in Chapter 14 of the [DoD 83] does not constitute a complete functional specifications of the Input-Output packages. Some aspects are not discussed at all, while others are deliberately left open to an implementation.

These gaps are filled in the appropriate subsections and summarized in subsection F.8.a.

The reader should be familiar with

[DoD 83]                   - The Ada language definition

and certain sections require that the reader is familiar with

[DEC 84a]                 - Guide to VAX/VMS File Applications

[DEC 84b]                 - Record Management Services

[DEC 85]                 - VAX/VMS I/O Users Reference Manual

Appendix  
User's Guide

F.10.1 External Files and File Objects

An external file is either any VAX/VMS file residing on a file-structured device (disk, tape), a record structured device (terminal, lineprinter), or a virtual software device (mailbox). [DoD 83] 14.1(1).

Identification of an external file by a string (the NAME parameter) is described in subsection F.8.2.1.

System-dependent characteristics (the FORM parameter) is described in subsection F.8.2.1

An external file created on a file-structured device will exist after program termination, and may be accessed later from an Ada program, except if the file is a temporary file created by using an empty name parameter. If files corresponding to the external file have not been closed, the external file will also exist upon program completion, and the contents will be the same as if the files had been closed prior to program completion. See further F.8.3. [DoD 83] 14.1(7).

Input-Output of access types will cause input-output of the access value [Dod 83] 14.1(7).

Sharing of an external file is, when using the default system-dependent characteristics, handled as described in the following.

When a file is associated with an external file using the Record Management Services (RMS), and the file is opened with mode IN\_FILE, the implementation will allow the current process and other processes to open files associated with the same external file (e.g. as IN\_FILE in an Ada program).

When a file is opened with mode INOUT\_FILE or OUT\_FILE no file sharing is allowed when using RMS. In particular, trying to gain write access to an external file shared by other files, by OPEN or RESET to mode INOUT\_FILE or OUT\_FILE will raise USE\_ERROR.

When a text file is associated with a terminal device, using the Queue I/O System Services (QIO), there are no restrictions on file sharing.

Appendix  
User's Guide

F.10.2 Sequential and Direct Files

When dealing with sequential and direct input-output only RMS files are used.

In this section, a description of the basic file-mapping is given.

Basic file-mapping concerns the relation between Ada files and (formats of) external RMS files, and the strategy for accessing the external files. When creating new files (with the CREATE procedure), there is a unique mapping onto a RMS file format, the preferred file format. When opening an existing external file (with the OPEN procedure), the mapping is not unique; i.e. several external file formats other than preferred for CREATE may be acceptable. In subsection F.8.2.1 the preferred and acceptable formats are described for sequential and direct input-output. In subsection F.8.3.1 the preferred and acceptable formats are described for text input-output.

F.10.2.1 File Management

This subsection contains information regarding file management :

- Description of preferred and acceptable formats for sequential and direct input-output.
- The NAME parameter.
- The FORM parameter.
- File access.

Preferred and Acceptable Formats

The preferred and acceptable formats for sequential and direct input-output, are described using RMS notation and abbreviations [DEC 84b]. ES is used to denote the element size, i.e. the number of bytes occupied by the element type, or, in case of a varying size type, the maximum size (which must be determinable at the point of instantiation from the value of the SIZE attribute for the element type).

It should be noted that the latter means a type definition like:

```
type large_type is array( integer <> ) of integer;
```

would be mapped onto an element size greater than the maximum allowed size (32 k byte).

Appendix  
User's Guide

SEQUENTIAL\_IO:

An element is mapped into a single record of the external file, or if block-io is used, a number of consecutive virtual blocks of 512 bytes. ES must not be greater than 32767, otherwise USE\_ERROR is raised.

CREATE - preferred file format

- ORG=SEQ, RFM=FIX, MRS=ES  
(note: read and write operations will be done by BLOCK IO if element size is a multiple of 512 bytes)

OPEN - acceptable formats

- ORG=REL, RFM=FIX, MRS=ES
- ORG=SEQ, RFM=FIX, MRS=ES
- ORG=SEQ, RFM=VAR
- ORG=SEQ, RFM=UDF  
(note: BLOCK IO will be used)

(note: a RESET operation to OUT\_FILE mode will give a USE\_ERROR exception, as it is not possible to empty a file of this format).

The detailed setting of the control blocks for sequential\_IO is given below. Note that the user-provided form parameter will override the default specified settings, when used with OPEN or CREATE.

Also note that, when an Ada program contains tasks, asynchronous I/O will be used (ROP = <ASY>).

The following shows the initial setting for OPEN and CREATE (unspecified fields in the control blocks will be cleared to zero).

FAB:

ALQ = 12  
DEQ = 6  
DNM = <.DAT>  
FAC = for block-io, IN\_FILE: <BRO,GET>  
for block-io, OUT\_FILE: <BRO,PUT,UPD,DEL,TRN>  
otherwise, IN\_FILE: <GET>  
otherwise, OUT\_FILE: <PUT,UPD,DEL,TRN>  
FNM = name parameter  
FOP = non-empty name parameter: <MXV,SQO>  
empty name parameter to CREATE: <MXV,SQO,TMP>



Appendix  
User's Guide

MRS = element size (in bytes)  
NAM = address of name-block  
ORG = SEQ  
RAT = <CR>  
RFM = FIX  
SHR = for IN\_FILE: <GET>  
      for OUT\_FILE: <NIL>  
XAB = address of XABFHC block

RAB:

FAB = address of FAB block  
KBF = address of internal longword  
KSZ = 4  
RAC = SEQ  
ROP = for block-io: <BIO>  
      otherwise: <UIF>

NAM:

RSA = address of internal 255 byte buffer  
RSS = 255

XABFHC:

NXT = 0

DIRECT\_IO:

An element is mapped into a single record of the external file, or if block io is used, the smallest possible number of consecutive virtual blocks of 512 bytes. ES must not be greater than 32767, otherwise USE\_ERROR will be raised.

CREATE - preferred file format

- if element size is not a multiple of 512:           ORG=REL,  
      RFM=FIX, MRS=ES
- if element size is a multiple of 512:   ORG=SEQ, RFM=FIX,  
      MRS=ES  
      (note: read and write operations will be done by BLOCK  
      IO)

OPEN - acceptable formats

- ORG=REL, RFM=FIX, MRS=ES
- ORG=SEQ, RFM=FIX, MRS=ES  
      (note: if element size is a multiple of 512, BLOCK IO  
      will be used)
- ORG=SEQ, RFM=UDF  
      (note: BLOCK IO will be used)

## Appendix

### User's Guide

The detailed setting of the control blocks for direct\_IO is given below. Note that the user-provided form parameter will override the default specified settings, when used with OPEN or CREATE.

Also note that, when an Ada program contains tasks, asynchronous I/O will be used (ROP = <ASY>).

The initial setting for OPEN and CREATE (unspecified fields in the control blocks will be cleared to zero) follows:

#### FAB:

ALQ = 12  
DEQ = 6  
DNM = <.DAT>  
FAC = for IN\_FILE: <GET>  
      for OUT\_FILE: <GET,PUT,UPD,DEL,TRN>  
FNM = name parameter  
  
FOP = non-empty name parameter: <MXV,SQO>  
      empty name parameter to CREATE: <MXV,SQO,TMP>  
MRS = 512  
NAM = address of name-block  
ORG = SEQ  
RAT = <CR>  
RFM = VAR  
SHR = for IN\_FILE: <GET>  
      for OUT\_FILE: <NIL>  
XAB = address of XABFHC block

#### RAB:

FAB = address of FAB block  
KBF = address of internal 1cngword  
KSZ = 4  
RAC = SEQ  
ROP = <>  
UBF = address of internal 512 byte buffer  
USZ = 512

#### NAM:

RSA = address of internal 255 byte buffer  
RSS = 255

#### XABFHC:

NXT = 0

Appendix  
User's Guide

Name Parameter

The name parameter, when non null, must be a valid VAX/VMS file specification referring to a file-structured device; a file with that name will then be created.

For a null name parameter, the process' current directory and device must designate a directory on a disk device; a temporary, unnamed file marked for deletion will then be created in that directory. The file will be deleted after closing it, or, if not closed when the program terminates. [DoD 83] 14.2.1(3).

Form Parameter

The FORM string parameter that can be supplied to any OPEN or CREATE procedure is for controlling the external file properties, such as physical organization, allocation etc. In the present implementation this has been achieved by accepting form parameters that specify setting of fields in the RMS control blocks FAB and RAB, used for all open files. This scheme is rather general in that it accepts all settings of the FAB and RAB fields. It opens for modifications of the behaviour required by the Arm, such as being able to open a file for appending data to it. Furthermore, a form parameter for accessing mailboxes is provided.

The following fields can currently not be set explicitly:

FAB:

FNA, FNS (are set by the NAME parameter of OPEN or CREATE)  
DNA, DNS (can be set by DNM=/.../)

The syntax of the form parameter is as follows:

```
form_parameter ::= [ param { , param } ]

param          ::= number_param
                | string_param
                | quotation_param
                | mask_param

number_param   ::= keyword = number
number         ::= digit { digit }
digit          ::= 0 | 1 | ... | 9
string_param   ::= keyword = string
string         ::= / {any character other than slash} /
```

Apper dix  
User's Guide

```
quotation_param ::= keyword = specifier

mask_param      ::= clear_bits
                  | set_bits
                  | define_whole_field

clear_bits      ::= keyword - mask
set_bits        ::= keyword + mask
define_whole_field
                ::= keyword = mask

mask            ::= < [ specifier { , specifier } ] >

keyword         ::= letter letter letter
specifier       ::= letter letter letter [ letter letter ]

letter          ::= A | B | ... | Z | a | b | ... | z
```

Notes:

- . all space characters are ignored.
- . string parameters are converted to uppercase.
- . all keywords and specifiers are 3- or 5-letter words, like the RMS assembly level interface symbolic names. The only exceptions are the RAT=<CR> specifier, which in this implementation must be specified as CAR rather than CR, and the RAB CTX field keyword, which must be specified as CON. There are only 2 5-letter words: the specifiers STMCR and STMLF.

The semantics of the form parameter is (except for the mailbox parameter) to modify the specified FAB and RAB fields just prior to actually calling RMS to open or create a file, i.e. the form parameter overrides the default conventions provided by this implementation ([DoD 83], Section F.5.4). The form parameter is interpreted left to right, and it is legal to respecify fields; in particular a mask field may be manipulated in several turns.

Note that there is no way of modifying fields after an RMS open or create service, in particular it is not possible to set RAB fields on a per record operation basis.

The modifications made are those to be expected from the textually corresponding RMS macro specifications. However, the clear\_bits and set\_bits are particular to this implementation: They serve to either clear individual mask specifiers set by the implementation default, or to set mask specifiers in addition to those specified by the implementation default, respectively.

Appendix  
User's Guide

The mailbox parameter can be either

```
    MBX=TMP  
or  
    MBX=PRM
```

It applies to CREATE only, and causes either a temporary or a permanent mailbox to be created. The NAME parameter will be used to establish a logical name for the mailbox, unless an empty string is specified (in this case, no logical name will be established).

Note that the implementation does in no way check that the form parameter supplied is at all reasonable. The attitude is "you asked for it, you got it". It is discouraged, if other procedures than OPEN, CREATE, and CLOSE will be called, to set ORG, RAC, MRS, NAM, FOP=<NAM>. It is generally discouraged to set XAB.

Examples:

```
-- create a text file  
create(file, out_file, "DATA.TXT");  
  
-- create a temporary text file which will be deleted  
  after completion of the main program  
create(file, out_file);  
  
-- create an empty stream format text file  
create(file, out_file, "DATA.DAT", "ORG=SEQ, RFM=STMLF");  
  
-- create a very big file:  
create(file, out_file, "DATA.DAT", "ALQ=2048, DEQ=256");  
  
-- create a temporary mailbox:  
create(file, out_file, "HELLO", "MBX=TMP");  
  
-- open a mailbox; at reading, do not wait for  
  messages:  
open(file, in_file, "HELLO", "ROP+<TMO>, TMO=0");
```

File Access

The OPEN and CREATE procedures utilize the normal RMS defaulting mechanism to determine the exact file to open or create.

Device and directory (when not specified) defaults to the process' current device (SYSSDISK) and directory.

Appendix  
User's Guide

The version number (when not specified), defaults for OPEN to highest existing, or for CREATE, one higher than the highest existing, or 1 when no version exists.

The implementation provides .DAT as the default file type.

External files, which are not to be accessed via block-io (as described in formats), will be accessed via standard RMS access methods. For SEQUENTIAL\_IO, sequential record access mode will be used. For DIRECT\_IO, random access by record number will be used.

Creation of a file with mode IN\_FILE will raise USE\_ERROR, when referring to an RMS file.

For sequential and direct io, files created by SEQUENTIAL\_IO for a given type T, may be opened (and processed) by DIRECT\_IO for the same type and vice-versa. In the latter case, however, the function END\_OF\_FILE (14.2.2(8)) may fail to produce TRUE in cases where the file has been written at random, leaving "holes" in the file. See [DoD 83] 14.2.1(7).

For a sequential or text file associated with an RMS file, a RESET operation to OUT\_FILE mode will cause deletion of any elements in the file, i.e. the file is emptied. Likewise, a sequential file or text file opened (by OPEN) with mode OUT\_FILE, will be emptied. For any other RESET operation, the contents of the file is not affected.

For a text file, any RESET operation will cause USE\_ERROR to be raised, when QIO services are used.

Appendix  
User's Guide

F.10.2.2 Sequential Input-Output

The implementation omits type checking for DATA\_ERROR, in case the element type is of an unconstrained type, [DoD 83] 14.2.2(4), i.e.:

```
... f : FILE_TYPE
type et is 1..100;
type eat is array( et range <> ) of integer;

X : eat( 1..2 );
Y : eat( 1..4 );
...
-- write X, Y:

write( f, X); write( f, Y); reset( f, IN_FILE);

-- read X into Y and Y into X:

read( f, Y); read( f, X);
```

This should have given DATA\_ERROR, but will instead give undefined values in the last 2 elements of Y.

F.10.2.3 Specification of the Package Sequential\_IO

```
with BASIC_IO_TYPES;
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);

-- 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;
```

Appendix  
User's Guide

```
        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;

-- 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;

-- 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;

private

    type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end SEQUENTIAL_IO;
```

#### F.10.2.4 Direct Input-Output

The implementation omits type checking for DATA\_ERROR, in case the element type is of an unconstrained type, [Dod 83] 14.2.4(4), see F.8.2.2.



Appendix  
User's Guide

**F.10.2.5      Specification of the Package Direct\_IO**

```
with BASIC_IO_TYPES;  
with IO_EXCEPTIONS;
```

```
generic
```

```
    type ELEMENT_TYPE is private;
```

```
package DIRECT_IO is
```

```
    type FILE_TYPE is limited private;
```

```
    type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
```

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

```
    subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;
```

```
-- File management
```

```
procedure CREATE(FILE : in out FILE_TYPE;  
                  MODE : in      FILE_MODE      := INOUT_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;
```

```
-- input and output operations
```

Appendix  
User's Guide

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

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

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

function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;
function SIZE (FILE : in FILE_TYPE) return COUNT;
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

-- 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;

private

type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end DIRECT_IO;
```

### F.10.3 Text Input-Output

When utilizing text input-output, RMS is used when an external file is residing on a file-structured device, or is a virtual software device. When an external file that is a terminal device is opened or created, the queue I/O services (QIO) are used by default.

If a text file of mode `OUT_FILE` corresponds to an external RMS file, the external file will also exist upon program completion, and a pending linebuffer will be flushed before the text file is closed.

Appendix  
User's Guide

F.10.3.1 File Management

This subsection contains information regarding file management, where it differs from the file management described in F.8.2.1.

- Description of preferred and acceptable formats for text input-output.
- The FORM parameter.
- File access.

Preferred and Acceptable Formats

Lines of text are mapped into records of external files.

For output, the following rules apply.

The Ada line terminators and file terminators are never explicitly stored (however, for stream format files, RMS forces line terminators to trail each record). Page terminators, except the last, are mapped into a form feed character trailing the last line of the page. (In particular, an empty page (except the last) is mapped into a single record containing only a form feed character). The last page terminator in a file is never represented in the external file. It is not possible to write records containing more than 512 characters. That is, the maximum line length is 511 or 512, depending on whether a page terminator (form feed character) must be written or not. If output is more than 512 characters, USE\_ERROR will be raised.

On input, a FF trailing a record indicates that the record contains the last line of a page and that at least one more page exists. The physical end of file indicates the end of the last page.

CREATE - preferred file format

- ORG=SEQ, RFM=VAR, MRS=512

OPEN - acceptable file formats

Appendix  
User's Guide

- all formats except
  - ORG=IDX
  - RFM=UDF

(Note: for stream files (RFM=STM...) any sequence of the LF, CR, and VT control characters at the end of a line will be stripped off at input. At output, line terminators will be provided by RMS defaults). (Note: input of any record containing more than 512 characters will raise a USE\_ERROR exception).

The detailed setting of the control blocks for TEXT\_IO is given below. Note that the user-provided form parameter will override the default specified settings, when used with OPEN or CREATE.

Also note that, when an Ada program contains tasks, asynchronous I/O will be used. When RMS files ROP = <ASY>, or asynchronous QIO when terminal devices.

The following shows the initial setting for OPEN and CREATE (unspecified fields in the control blocks will be cleared to zero):

FAB:

ALQ = 12  
DEQ = 6  
DNM = <.DAT>  
FAC = for IN\_FILE: <GET>  
for OUT\_FILE: <GET, PUT, UPD, DEL, TRN>  
FNM = name parameter  
FOP = non-empty name parameter <MXV, SQO>  
empty name parameter to CREATE: <MXV, SQO, TMP>  
MRS = 512  
NAM = address of name-block  
ORG = SEQ  
RAT = <CR>  
RFM = VAR  
SHR = for IN\_FILE: <GET>  
for OUT\_FILE: <NIL>  
XAB = address of XABFHC block

RAB:

FAB = address of FAB block  
KBF = address of internal longword  
KSZ = 4  
RAC = SEQ  
ROP = <>  
UBF = address of internal 512 byte buffer  
USZ = 512

Appendix  
User's Guide

NAM:

RSA = address of internal 255 byte buffer  
USZ = 255

XABFHC:

NXT = 0

Form parameter

If any form parameter, except for the empty string or a string containing only blanks, is supplied to OPEN or CREATE, RMS services will always be used. In this case, the file operations on external files as terminal-devices will use buffered input- output.

File\_access

External RMS files are accessed via sequential record access methods.

Files associated with terminal devices, using QIO services, do not contain page terminators. This means that calling SKIP\_PAGE will raise USE\_ERROR. Furthermore, trying to RESET a file in this category will cause USE\_ERROR.

Files associated with the same external file, using QIO services, share the standard values (page-, line, and column-number), e.g. standard values for STANDARD\_OUTPUT are implicitly updated after reading from STANDARD\_INPUT.

F.10.3.2 Specification of the Package Text\_IO

with BASIC\_IO\_TYPES;  
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 .. LONG\_INTEGER'LAST;  
subtype POSITIVE\_COUNT is COUNT range 1 .. COUNT'LAST;  
UNBOUNDED: constant COUNT:= 0; -- line and page length

subtype FIELD is INTEGER range 0 .. 67;

subtype NUMBER\_BASE is INTEGER range 2 .. 16;

Appendix  
User's Guide

```
type TYPE_SET is (LOWER_CASE, UPPER_CASE);

-- 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;

-- 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;

-- 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);
```

Appendix  
User's Guide

```
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;
```

-- 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);
```

Appendix  
User's Guide

```
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;
```

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

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);
```

-- 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;
```



Appendix  
User's Guide

```

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;

```

-- 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);

```

Appendix  
User's Guide

```
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;

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;
```

Appendix  
User's Guide

```
-- 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  : in 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;

-- 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;

private

  type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end TEXT_IO;
```

Appendix  
User's Guide

F.10.4 Low Level Input-Output

The package LOW\_LEVEL\_IO is empty.

F.10.4.1 Clarifications of Ada Input-Output Requirements Summary

The Ada Input-Output concepts as presented in Chapter 14 of [DoD 83] do not constitute a complete functional specification of the Input-Output packages. Some aspects are not discussed at all, while others are deliberately left open to an implementation. These gaps are filled in below, with reference to sections of the [DoD 83].

F.10.4.2 Assumptions

- 14.2.1(15): For a sequential or text file, a RESET operation to OUT\_FILE mode deletes any elements in the file, i.e. the file is emptied. Likewise, a sequential or text file opened (by OPEN) as an OUT\_FILE, will be emptied. For any other RESET operation, the contents of the file is not affected.
- 14.2.1(7) : For sequential and direct io, files created by SEQUENTIAL\_IO for a given type T, may be opened (and processed) by DIRECT\_IO for the same type and vice-versa. In the latter case, however, the function END\_OF\_FILE (14.2.2(8)) may fail to produce TRUE in the case where the file has been written at random, leaving "holes" in the file.

F.10.4.3 Implementation Choices

- 14.1(1) : An external file is either any VAX/VMS file residing on a file-structured device (disk,tape), a record structured device (terminal, lineprinter), or a virtual software device (mailbox).
- 14.1(7) : An external file created on a file-structured device will exist after program termination, and may later be accessed from an Ada program.
- 14.1(13) : See Section F.8.2.1 File Management.

## Appendix

### User's Guide

14.2.1(3) : The name parameter, when non-null, must be a valid VAX/VMS file specification referring to a file-structured device; a file with that name will then be created. For a null name parameter, the process' current directory and device must designate a directory on a disk device; a temporary, unnamed file marked for deletion will then be created in that directory.

The form and effect of the form parameter is discussed in Sections F.8.2.1 and F.8.3.1.

Creation of a file with mode IN\_FILE will raise USE\_ERROR.

14.2.1(13): Deletion of a file is only supported for files on a disk device, and requires deletion access right to the file.

14.2.2(4): No check for DATA\_ERROR is performed in case the element type is of an unconstrained type.