REPORT DOCUMENTATION 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 Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final 08 Dec 1989 to 08 Dec 1990

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: Alsys, AlsyCOMP_018, Version 4.32, VAX 6210 (Host & Target), 891208A1.10228

5. FUNDING NUMBERS

6. AUTHOR(S)

AFNOR, Paris, FRANCE

DTIC
ELECTE
JUL 2 5 1990
S D

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

AFNOR
Tour Europe, Cedex 7
F-92080 Paris la Defense
FRANCE

8. PERFORMING ORGANIZATION REPORT NUMBER

AVF-VSR-AFNOR-89-13

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
United States Department of Defense
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)

Alsys, AlsyCOMP_018, Version 4.32, Paris, France, VAX 6210 under VMS 5.0-2 (Host & Target) ACVC 1.10.

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

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

AD-A224 616

AVF Control Number: AVF-VSR-AFNOR-89-13

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 891208A1.10228
Alsys
AlsyCOMP_018, Version 4.32
VAX 6210 Host and Target

Completion of On-Site Testing:
8 December 1989

Prepared By:
AFNOR
Tour Europe
Cedex 7
F-92049 Paris la Défense

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Accesion For		
NTIS CRA&I	☒	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

Ada Compiler Validation Summary Report:

Compiler Name: AlsyCOMP_018, Version 4.32

Certificate Number: 891208A1.10228

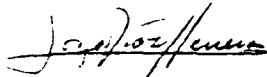Host: VAX 6210 under VMS 5.0-2

Target: VAX 6210 under VMS 5.0-2

Testing Completed 8 December 1989 Using ACVC 1.10

This report has been reviewed and is approved.

Fabrice Garnier de Labareyre

AFNOR
Fabrice Garnier de Labareyre
Tour Europe
Cedex 7
F-92049 Paris la Défense

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

Ada Joint Program Office
Dr. John Solomond
Director
Dep?   :-nse
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

CHAPTER 2 CONFIGURATION INFORMATION

CHAPTER 3 TEST INFORMATION

APPENDIX A DECLARATION OF CONFORMANCE

APPENDIX B TEST PARAMETERS

APPENDIX C WITHDRAWN TESTS

APPENDIX D APPENDIX F OF THE Ada STANDARD

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

. To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

. To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

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

Testing of this compiler was conducted by Alsys under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 8 December 1989 at Alsys SA, in La Celle Saint Cloud, FRANCE.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

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

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

> Ada Information Clearinghouse
> Ada Joint Program Office
> OUSDRE
> The Pentagon, Rm 3D-139 (Fern Street)
> Washington DC 20301-3081

or from:

> AFNOR
> Tour Europe
> cedex 7
> F-92049 Paris la Défense

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

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

1.3 REFERENCES

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

2. Ada Compiler Validation Procedures, Ada Joint Program Office, May 1989.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, January 1989

1.4 DEFINITION OF TERMS

ACVC   The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.

Ada Commentary An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Ada Standard   ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant   The agency requesting validation.

AVF   The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures.

AVO   The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler   A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

Failed test   An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host   The computer on which the compiler resides.

Inapplicable test An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test   An ACVC test for which a compiler generates the expected result.

Target   The computer which executes the code generated by the compiler.

Test   A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn   An ACVC test found to be incorrect and not used to check test conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada 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, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation ma﹍   ﹍nately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It 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 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. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

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

Compiler: AlsyCOMP_018, Version 4.32

ACVC Version: 1.10

Certificate Number: 891208A1.10228

Host Computer:

Machine:	VAX 6210
Operating System:	VMS 5.0-2
Memory Size:	32 Mb

Target Computer:

Machine:	VAX 6210
Operating System:	VMS 5.0-2
Memory Size:	32 Mb

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1)   The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2)   The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3)   The compiler correctly processes a test containing block statements nested to 65 levels. (See test D56001B.)

(4)   The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1)   This implementation supports the additional predefined types, SHORT_SHORT_INTEGER, SHORT_INTEGER, LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Based literals.

(1)   An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR when a value exceeds SYSTEM.MAX_INT . This implementation raises NUMERIC_ERROR during execution. (See test E24201A.)

d. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1)   Apparently no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

(2)   Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3)   This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

(4)   NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5)   NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6)   Underflow is gradual. (See tests C45524A..Z.) (26 tests)

e. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

   (1)   The method used for rounding to integer is round away from zero. (See tests C46012A..Z.) (26 tests)

   (2)   The method used for rounding to longest integer is round away from zero. (See tests C46012A..Z.) (26 tests)

   (3)   The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

f. Array types.

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

   (1)   Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR . (See test C36003A.)

   (2)   NUMERIC_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

   (3)   NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

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

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

   (6)   In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

   (7)   In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared . (See test E52103Y.)

h. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

i. Aggregates.

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

(2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

j. Pragmas.

(1) The pragma INLINE is supported for functions or procedures, but not functions called inside a package specification. (See tests LA3004A..B, EA3004C..D, and CA3004E..F.)

k. Generics.

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

(3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

(4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

(5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

(6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

(7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

(8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

(9)   Generic unit bodies and their subunits can be compiled in separate
        compilations. (See test CA3011A.)

1. Input and output.

(1)   The package SEQUENTIAL_IO can be instantiated with unconstrained
        array types and record types with discriminants without defaults.
        (See tests AE2101C, EE2201D, and EE2201E.)

(2)   The package DIRECT_IO can be instantiated with unconstrained array
        types and record types with discriminants but CREATE will raise
        USE_ERROR. (See tests AE2101H, EE2401D, and EE2401G.)

(4)   Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See
        tests CE2102D..E, CE2102N, and CE2102P.)

(5)   Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO.
        (See tests CE2102F, CE2102I..J, CE2102R, CE2102T, and CE2102V.)

(6)   Modes IN_FILE and OUT_FILE are supported for text files. (See tests
        CE3102E and CE3102I..K.)

(7)   RESET   and DELETE operations are supported for SEQUENTIAL_IO. (See
        tests CE2102G and CE2102X.)

(8)   RESET   and DELETE operations are supported for DIRECT_IO. (See tests
        CE2102K and CE2102Y.)

(9)   RESET and DELETE operations are supported for text files. (See tests
        CE3102F..G, CE3104C, CE3110A, and CE3114A.)

(10)   Overwriting to a sequential file truncates to the last element
        written. (See test CE2208B.)

(11)   Temporary sequential files are   ot given name·   ˙ when
        closed. (See test CE2108A.)

(12)   Temporary direct files are not given names and not deleted when
        closed. (See test CE2108C.)

(13)   Temporary text files are not given names and not deleted when closed.
        (See test CE3112A.)

(14)   Only one internal file can be associated with each external file for
        sequential files when reading or writing (See tests CE2107A..E,
        CE2102L, CE2110B, and CE2111D.)

(15)   Only one internal file can be associated with each external file for
        direct files when reading or writing (See tests CE2107F..I, CE2110D
        and CE2111H.)

(16)   Only one internal file can be associated with each external file for
        text files when reading or writing. (See tests CE3111A..E, CE3114B,
        and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 386 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 52 tests were required. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1130	1939	17	26	46	3287
Inapplicable	0	8	376	0	2	0	386
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	577	545	245	172	99	161	331	137	36	252	259	275	3287
Inappl	14	72	135	3	0	0	5	1	0	0	0	110	46	386
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

```
A39005?  B97102E  BC3009B  C97116A  CD2A62D  CD2A63A  CD2A63B  CD2A63C  CD2A63D
CD2A66A  CD2A66B  CD2A66C  CD2A66D  CD2A73A  CD2A73B  CD2A73C  CD2A73D  CD2A76A
CD2A76B  CD2A76C  CD2A76D  CD2A81G  CD2A83G  CD2A84M  CD2A84N  CD2D11B  CD2B15C
CD5007B  CD5011O  CD7105A  CD7203B  CD7204B  CD7205C  CD7205D  CE2107I  CE3111C
CE3301A  CE3411B  E28005C  ED7004B  ED7005C  ED7005D  ED7006C  ED7006D
```

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 386 tests were inapplicable for the reasons indicated:

- The following 201 tests are not applicable because they 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)

- C35702A and B86001T are not applicable because this implementation supports no predefined type Short_Float.

- C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of System.Max_Mantissa is less than 32.

- The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:
    ```
    C45231C C45304C C45502C C45503C C45504C C45504F C45611C C45613C
    C45614C C45631C C45632C B52004D C55B07A B55B09C B86001W CD7101F
    ```

- C86001F, is not applicable because recompilation of Package SYSTEM is not allowed.

- B86001Y is not applicable because this implementation supports no predefined fixed-point type other than Duration.

. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than Float, Long_Float, or Short_Float.

. B91001H is not applicable because address clause for entries is not supported by this implementation.

. BD5006D is not applicable because address clause for packages is not supported by this implementation.

. The following 10 tests are not applicable because size clause on float is not supported by this implementation:

    CD1009C   CD2A41A..B (2 tests)
    CD2A41E   CD2A42A..B (2 tests)
    CD2A42E..F (2 tests)   CD2A42I..J (2 tests)

. CD1C04B, CD1C04E, CD4051A..D (4 tests) are not applicable because representation clause on derived records or derived tasks is not supported by this implementation.

. CD2A84B..I (8 tests), CD2A84K..L (2 tests) are not applicable because size clause on access type is not supported by this implementation.

. The following 28 tests are not applicable because size clause for derived private type is not supported by this implementation:

    CD1C04A   CD2A21C..D (2 tests)
    CD2A22C..D (2 tests)   CD2A22G..H (2 tests)
    CD2A31C..D (2 tests)   CD2A32C..D (2 tests)
    CD2A32G..H (2 tests)   CD2A41C..D (2 tests)
    CD2A42C..D (2 tests)   CD2A42G..H (2 tests)
    CD2A51C..D (2 tests)   CD2A52C..D (2 tests)
    CD2A52G..H (2 tests)   CD2A53D
    CD2A54D   CD2A54H

. The following 29 tests are not applicable because of the way this implementation allocates storage space for one component, size specification clause for an array type or for a record type requires compression of the storage space needed for all the components (without gaps).

    CD2A61A..D (4 tests)   CD2A61F
    CD2A61H..L (5 tests)   CD2A62A..C (3 tests)
    CD2A71A..D (4 tests)   CD2A72A..D (4 tests)
    CD2A74A..D (4 tests)   CD2A75A..D (4 tests)

. CD4041A is not applicable because alignment "at mod 8" is not supported by this implementation.

. The following 21 tests are not applicable because address clause for a constant is not supported by this implementation:

    CD5011B,D,F,H,L,N,R (7 tests)   CD5012C,D,G,H,L (5 tests)
    CD5013B,D,F,H,L,N,R (7 tests)   CD5014U,W (2 tests)

. CD5012J, CD5013S, CD5014S are not applicable because address clause for a task is not supported by this implementation.

. CE2102D is inapplicable because this implementation supports create with in_file mode for SEQUENTIAL_IO.

- CE2102E is inapplicable because this implementation supports create with out_file mode for SEQUENTIAL_IO.

- CE2102F is inapplicable because this implementation supports create with inout_file mode for DIRECT_IO.

- CE2102I is inapplicable because this implementation supports create with in_file mode for DIRECT_IO.

- CE2102J is inapplicable because this implementation supports create with out_file mode for DIRECT_IO.

- CE2102N is inapplicable because this implementation supports open with in_file mode for SEQUENTIAL_IO.

- CE2102O is inapplicable because this implementation supports RESET with in_file mode for SEQUENTIAL_IO.

- CE2102P is inapplicable because this implementation supports open with out_file mode for SEQUENTIAL_IO.

- CE2102Q is inapplicable because this implementation supports RESET with out_file mode for SEQUENTIAL_IO.

- CE2102R is inapplicable because this implementation supports open with inout_file mode for DIRECT_IO.

- CE2102S is inapplicable because this implementation supports RESET with inout_file mode for DIRECT_IO.

- CE2102T is inapplicable because this implementation supports open with in_file mode for DIRECT_IO.

- CE2102U is inapplicable because this implementation supports RESET with in_file mode for DIRECT_IO.

- CE2102V is inapplicable because this implementation supports open with out_file mode for DIRECT_IO.

- CE2102W is inapplicable because this implementation supports RESET with out_file mode for DIRECT_IO.

- EE2401D and EE2401G are not applicable because USE_ERROR is raised when the CREATE of an instantiation of DIRECT_IO with unconstrained type is called.

- CE2401H is not applicable because create with inout_file mode for unconstrained records with default discriminants is not supported by this implementation.

- CE3102E is inapplicable because this implementation supports create with in_file mode for text files.

- CE3102F is inapplicable because this implementation supports reset for text files, for out_file, in_file and from out_file to in_file mode.

- CE3102G is inapplicable because this implementation supports deletion of an external file for text files.

. CE3102I is inapplicable because this implementation supports create with out_file mode for text files.

. CE3102J is inapplicable because this implementation supports open with in_file mode for text files.

. CE3102K is inapplicable because this implementation supports open with out_file mode for text files.

. The following 7 tests are not applicable because no name is associated to a temporary file.

CE2107C..D (2 tests)	CE2107H
CE2107L	CE2108B
CE2108D	CE3112B

. The following 5 tests are not applicable because multiple internal files cannot be associated with the same external file for sequential files. The proper exception is raised when multiple access is attempted.

CE2107A..B (2 tests)	CE2107E
CE2110B	CE2111D

. The following 4 tests are not applicable because multiple internal files cannot be associated with the same external file for direct files. The proper exception is raised when multiple access is attempted.

CE2107F..G (2 tests)	CE2110D
CE2111H	

. The following 6 tests are not applicable because multiple internal files cannot be associated with the same external file for text files. The proper exception is raised when multiple access is attempted.

CE3111A..B (2 tests)	CE3111D..E (2 tests)
CE3114B	. CE3115A

3.6 TEST, PROCESSING, AND   EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 52 tests.

The following 27 tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:
B23004A B24007A B24009A B28003A B32202A B32202B B32202C B33001A B36307A B37004A
B49003A B49005A B61012A B62001B B74304B B74304C B74401F B74401R B91004A B95032A
B95069A B95069B BA1101B BC2001D BC3009A BC3009C BD5005B

The following 21 tests were split in order to show that the compiler was able to find the representation clause indicated by the comment --N/A =>ERROR :

CD2A61A CD2A61B CD2A61F CD2A61I CD2A61J CD2A62A CD2A62B CD2A71A CD2A71B CD2A72A CD2A72B CD2A75A CD2A75B CD2A84B CD2A84C CD2A84D CD2A84E CD2A84F CD2A84G CD2A84H CD2A84I

The test EA3004D when run as it is, the implementation fails to detect an error on line 27 of test file EA3004D6M . This is because the pragma INLINE has no effect when its object is within a package specification. However, the results of running the test as it is does not confirm that the pragma had no effect, only that the package was not made obsolete. By re-ordering the compilations so that the two subprograms are compiled after file D5 (the re-compilation of the "with"ed package that makes the various earlier units obsolete), we create a test that shows that indeed pragma INLINE has no effect when applied to a subprogram that is called within a package specification: the test then executes and produces the expected NOT_APPLICABLE result (as though INLINE were not supported at all). The re-ordering of EA3004D test files is 0-1-4-5-2-3-6.

BA2001E requires that duplicate names of subunits with a common ancestor be detected and rejected at compile time. This implementation detects the error at link time, and the AVO ruled that this behavior is acceptable.

Modified version was produced for C87B62B, in order to have the test run to completion and fully exhibit the test behavior:
An explicit STORAGE_SIZE clause was added for the access type declared at line 68. This allows the test to execute without raising STORAGE_ERROR and to meet its objective (test overloading resolution in expression within length clause). The test then produces the expected PASSED result.

Modified version was produced for CC1223A, in order to have the test run to completion and fully exhibit the test behavior:
This test uses an expression within a generic body that cause the test to raise an NUMERIC_OVERFLOW unexpected exception at line 262. The expression "2**T'MANTISSA -1" on line 262 was changed to the equivalent form "(2**(T'MANTISSA-1)-1 + 2**(T'MANTISSA-1))" in order to avoid generating the exception-raising value 2**31. The test then produces the expected PASSED result.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the AlsyCOMP_018, Version 4.32 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the AlsyCOMP_018, Version 4.32 compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	VAX 6210
Host operating system:	VMS 5.0-2
Target computer:	VAX 6210
Target operating system:	VMS 5.0-2
Compiler:	AlsyCOMP_018, Version 4.32

A Cartridge TK50 containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the Cartridge TK50. Tests requiring modifications during the prevalidation testing were included in their modified form on the Cartridge TK50.

The contents of the Cartridge TK50 were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled, linked and all executable tests were run on the VAX 6210.

The compiler was tested using command scripts provided by Alsys and reviewed by the validation team. The compiler was tested using all default option settings except for the following:

OPTION   EFFECT

CALLS=INLINED   The pragma INLINE are taken into account

REDUCTION=EXTENSIVE   Perform some high level optimization

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on Cartridge TK50 and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Alsys SA, in La Celle Saint Cloud, FRANCE and was completed on 8 December 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Alsys has submitted the following Declaration of Conformance
concerning the AlsyCOMP_018, Version 4.32 compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: Alsys

Ada Validation Facility: AFNOR, Tour Europe Cedex 7,
                              F-92080 Paris la Défense

Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name: AlsyCOMP_018, Version 4.32
Host Architecture: VAX 6210
Host OS and Version: VMS 5.0-2
Target Architecture: VAX 6210
Target OS and Version: VMS 5.0-2

Implementor's Declaration

I, the undersigned, representing Alsys, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Alsys is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration s̲   ̲ ̲   ̲ only in the owner's corporate name.

_____ Date: _____2 NOV 1989_____

Alsys
Etienne Morel, Managing Director

Owner's Declaration

I, the undersigned, representing Alsys, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

_____ Date: _____2 NOV 1989_____

Alsys
Etienne Morel, Managing Director

APPENDIX B

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(254 * 'A') & '1'
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(254 * 'A) & '2'
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(126 * 'A') & '3' & (128 * 'A')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(126 * 'A') & '4' & (128 * 'A')

Name and Meaning	Value
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(252 * '0') & '298'
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(250 * '0') & '690.0'
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & (127 * 'A') & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & (127 * 'A') & '1"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(235 * ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2**31
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	VAX_VMS
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31

Name and Meaning	Value
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is 'RATION'BASE'LAST	100_000_000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	127
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name specifying a non existent directory	/~/*/f1
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name different from $ILLEGAL_EXTERNAL_FILE_NAME1	/~/*/f2
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648

Name and Meaning	Value
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-100_000_000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	1
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	255
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2:11: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	'2:' & (250 * '0') & '11:'
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16: F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	'16:' & (248 * '0') & 'F.E:'

Name and Meaning	Value
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & (253 * 'A') & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	SHORT_SHORT_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	VAX_VMS
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma memory_size, other than DEFAULT_MEM_SIZE. If there is no other value, then use DEFAULT_MEM_SIZE.	2**31
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma storage_unit, other than DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	VAX_VMS
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one inout parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX C

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

E28005C
This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

A39005G
This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

B97102E
This test contains an unitended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

C97116A
This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

BC3009B
This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

CD2A62D
This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests]
These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests]
These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

CD2B15C & CD7205C

This tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

CD2D11B

This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

CD5007B

This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

ED7004B, ED7005C & D, ED7006C & D [5 tests]

These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

CD7105A

This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

CD7203B, & CD7204B

These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD7205D

This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

CE2107I

This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

CE3111C

This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

CE3301A

This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

CE3411B

This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX D

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 the AlsyCOMP_018, Version 4.32 compiler, as described in this Appendix, are provided by Alsys. 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_SHORT_INTEGER is range -128 .. 127;

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

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

 type FLOAT is digits 6 range
  -2#1.111_1111_1111_1111_1111_1111#E+126
..
   2#1.111_1111_1111_1111_1111_1111#E+126;

 type LONG_FLOAT is digits 15 range
-2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E1022
 ..
 2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E1022;

 type DURATION is delta 2#0.000_000_000_000_01# range -86_400.0 .. 86_400.0;

 ...

 end STANDARD;
```

Alsys Ada

VAX/VMS and VAX/Ultrix Compiler

APPENDIX F

Version 4.32

TABLE OF CONTENTS

APPENDIX F

1 INTERFACING THE LANGUAGE ADA WITH OTHER LANGUAGES

Programs written in Ada can interface with external subprograms written in another language, by use of the INTERFACE pragma. The format of the pragma is:

pragma INTERFACE (*language_name* , *Ada_subprogram_name*) ;

where the *language_name* can be any of

■ Assembler

■ C (VAX/Ultrix only)

The *Ada_subprogram_name* is the name by which the subprogram is known in Ada. For example, to call a subprogram known as FAST_FOURIER in Ada, written in C, the INTERFACE pragma is:

pragma INTERFACE (C, FAST_FOURIER);

To relate the name used in Ada with the name used in the original language, the Alsys Ada compiler converts the latter name to lower case and truncates it to 32 significant characters.

To avoid naming conflict with routines of the Alsys Ada Executive, *external routine* names should not begin with the letters *alsy* (whether in lower or upper case or a combination of both).

To allow the use of non-Ada naming conventions, such as special characters, or case sensitivity, an implementation-dependent pragma INTERFACE_NAME has been introduced:

pragma INTERFACE_NAME (*Ada_subprogram_name. name_string*) ;

so that, for example,

pragma INTERFACE_NAME (FAST_FOURIER, "fft");

will associate the FAST_FOURIER subprogram in Ada with the C subprogram fft.

VAX/VMS:

In order to conform to the naming conventions of the VMS linker, the name is truncated to 255 characters.

VAX/Ultrix:

In order to conform to the naming conventions of the Ultrix Linker, an underscore is prepended to the name given by name_string, the result is then truncated to 255 characters.

The pragma INTERFACE_NAME may be used anywhere in an Ada program where INTERFACE is allowed (see [13.9]). INTERFACE_NAME must occur after the corresponding pragma INTERFACE and within the same declarative part.

2   IMPLEMENTATION-DEPENDENT PRAGMAS

Pragma INTERFACE

This pragma has been described in the previous section.

Pragma IMPROVE and Pragma PACK

These pragmas are discussed in sections 5.7 and 5.8 on representation clauses for arrays and records.

Note that packing of record types is done systematically by the compiler. The pragma pack will affect the mapping of each component onto storage. Each component will be allocated on the logical size of the subtype.

Example:

```
type R is
    record
        C1 : BOOLEAN; C2 : INTEGER range 1 .. 10;
    end record;
pragma PACK(R);
-- the attribute R'SIZE returns 5
```

Pragma INDENT

This pragma is only used with the Alsys *Reformatter*, this tool offers the functionalities of a pretty-printer in an Ada environment.

The pragma is placed in the source file and interpreted by the Reformatter.

pragma INDENT(OFF)

causes the Reformatter not to modify the source lines after this pragma.

pragma INDENT(ON)

causes the Reformatter to resume its action after this pragma.

Pragmas not implemented

The following pragmas are not implemented:

```
CONTROLLED
MEMORY_SIZE
OPTIMIZE
STORAGE_UNIT
SYSTEM_NAME
```

3   IMPLEMENTATION-DEPENDENT ATTRIBUTES

In addition to the Representation Attributes of [13.7.2] and [13.7.3], there are four attributes which
are listed under F.5 below, for use in record representation clauses.

Limitations on the use of the attribute ADDRESS

The attribute ADDRESS is implemented for all prefixes that have meaningful addresses. The
following entities do not have meaningful addresses and will therefore cause a compilation error if
used as prefix to ADDRESS:

- A constant that is implemented as an immediate value i.e., does not have any space allocated for
  it.

- A package specification that is not a library unit.

- A package body that is not a library unit or a subunit.

4   PACKAGES SYSTEM AND STANDARD

This section contains information on two predefined library packages:

- a listing of the specification of the package SYSTEM

- a list of the implementation-dependent declarations in the package STANDARD.

package SYSTEM is

-- Standard Ada definitions

```
    type NAME is (VAX_ULTRIX) ;                              -- or VAX_VMS
      SYSTEM_NAME      : constant NAME := VAX_ULTRIX;        -- or VAX_VMS
      STORAGE_UNIT     : constant := 8 ;
      MEMORY_SIZE      : constant := 2**31;
      MIN_INT          : constant := -(2**31) ;
      MAX_INT          : constant := 2**31-1 ;
      MAX_DIGITS       : constant := 15 ;
      MAX_MANTISSA     : constant := 31 ;
      FINE_DELTA       : constant := 2#1.0#e-31 ;
      TICK             : constant := 0.01 ;

    type ADDRESS is private;
    NULL_ADDRESS : constant ADDRESS;

    subtype PRIORITY is INTEGER range 1..127;


    -- Address arithmetic

    function TO_INTEGER (LEFT : ADDRESS)
        return INTEGER;
    function TO_ADDRESS (LEFT : INTEGER)
        return ADDRESS;
```

```
function "+" (LEFT : INTEGER; RIGHT : ADDRESS)
    return ADDRESS;
function "+" (LEFT : ADDRESS; RIGHT : INTEGER)
    return ADDRESS;

function "-" (LEFT : ADDRESS; RIGHT : ADDRESS)
    return INTEGER;
function "-" (LEFT : ADDRESS; RIGHT : INTEGER)
    return ADDRESS;
function "mod" (LEFT : ADDRESS; RIGHT : POSITIVE)
    return NATURAL;

function "<"  (LEFT : ADDRESS; RIGHT : ADDRESS)
    return BOOLEAN;
function "<=" (LEFT : ADDRESS; RIGHT : ADDRESS)
    return BOOLEAN;
function ">"  (LEFT : ADDRESS; RIGHT : ADDRESS)
    return BOOLEAN;
function ">=" (LEFT : ADDRESS; RIGHT : ADDRESS)
    return BOOLEAN;
function IS_NULL (LEFT : ADDRESS)
    return BOOLEAN;
function LONG_WORD_ALIGNED (LEFT : ADDRESS)
    return BOOLEAN;
function ROUND (LEFT : ADDRESS)
    return ADDRESS;
-- Return the given address rounded to the next lower even value


end SYSTEM;
```

The package STANDARD

The following are the implementation-dependent parts of the package STANDARD:

```
type SHORT_SHORT_INTEGER is range -(2**7) .. (2**7 -1);
type SHORT_INTEGER is range -(2**15) .. (2**15 -1);
type INTEGER is range -(2**31) .. (2**31 -1);

type FLOAT is digits 6 range
    -(2.0 - 2.0**(-23)) * 2.0**127 ..
    +(2.0 - 2.0**(-23)) * 2.0**127;

type LONG_FLOAT is digits 15 range
    -(2.0 - 2.0**(-51)) * 2.0**1023 ..
    +(2.0 - 2.0**(-51)) * 2.0**1023;


type DURATION is delta 2.0**(-14) range -86_400.0 .. 86_400.0;
```

5 TYPE REPRESENTATION CLAUSES

The representation of an object is closely connected with its type. For this reason this section addresses successively the representation of enumeration, integer, floating point, fixed point, access, task, array and record types. For each class of type the representation of the corresponding objects is described.

Except in the case of array and record types, the description for each class of type is independent of the others. To understand the representation of an array type it is necessary to understand first the representation of its components. The same rule applies to record types.

Apart from implementation defined pragmas, Ada provides three means to control the size of objects:

- a (predefined) pragma PACK, when the object is an array, an array component, a record or a record component

- a record representation clause, when the object is a record or a record component

- a size specification, in any case.

For each class of types the effect of a size specification alone is described. Interference between size specifications, packing and record representation clauses is described under array and record types.

5.1 Enumeration Types

Size of the objects of an enumeration subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an enumeration subtype has the same size as its subtype.

Alignment of an enumeration subtype

An enumeration subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, it is otherwise even byte aligned.

Address of an object of an enumeration subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an enumeration subtype is even when its subtype is even byte aligned.

5.2 Integer Types

Size of the objects of an integer subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an integer subtype has the same size as its subtype.

Alignment of an integer subtype

An integer subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, it is word aligned if the size of the subtype is less than or equal to a word, it is long-word aligned otherwise.

Address of an object of an integer subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an integer subtype is byte, word or long-word depending on the previous alignment.

5.3 Floating Point Types

Size of the objects of a floating point subtype

An object of a floating point subtype has the same size as its subtype.

Alignment of a floating point subtype

A floating point subtype is always long-word aligned.

Address of an object of a floating point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a floating point subtype is always long-word aligned.

5.4 Fixed Point Types

Size of the objects of a fixed point subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of a fixed point type has the same size as its subtype.

Alignment of a fixed point subtype

A fixed point subtype is byte aligned if its size is less than or equal to 8 bits, it is word aligned if the size of the subtype is less than or equal to a word, it is long-word aligned otherwise.

Address of an object of a fixed point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a fixed point subtype is byte, word or long-word depending on the previous alignment.

5.5 Access Types

Size of an object of an access subtype

An object of an access subtype has the same size as its subtype, thus an object of an access subtype is always 32 bits long.

Alignment of an access subtype.

An access subtype is always long-word aligned.

Address of an object of an access subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an access subtype is always long-word aligned.

5.6 Task Types

Size of the objects of a task subtype

An object of a task subtype has the same size as its subtype. Thus an object of a task subtype is always 32 bits long.

Alignment of a task subtype

A task subtype is always long-word aligned.

Address of an object of a task subtype

Provided its alignment is not constrained by a record representation clause, the address of an object of a task subtype is always long-word aligned.

5.7 Array Types

Size of the objects of an array subtype

The size of an object of an array subtype is always equal to the size of the subtype of the object.

Alignment of an array subtype

If no pragma PACK applies to an array subtype and no size specification applies to its components, the array subtype is aligned according to the type of the components.

Address of an object of an array subtype

Provided its alignment is not constrained by a record representation clause, the address of an object of an array subtype is aligned according to the previous alignment.

5.8 Record Types

Size of an object of a record subtype

An object of a constrained record subtype has the same size as its subtype.

An object of an unconstrained record subtype has the same size as its subtype if this size is less than or equal to 8 kb. If the size of the subtype is greater than this, the object has the size necessary to store its current value; storage space is allocated and released as the discriminants of the record change.

Alignment of a record subtype

When no record representation clause applies to its base type, a record subtype is byte, word or long-word aligned according to the components.

When a record representation clause that does not contain an alignment clause applies to its base type, a record subtype is byte, word or long-word aligned according to the components.

When a record representation clause that contains an alignment clause applies to its base type, a record subtype has an alignment that obeys the alignment clause.

Address of an object of a record subtype

Provided its alignment is not constrained by a representation clause, the address of an object of a record subtype is byte, word or long-word aligned according to the previous alignment.

6 ADDRESS CLAUSES

6.1 Address Clauses for Objects

An address clause can be used to specify an address for an object as described in RM 13.5. When such a clause applies to an object no storage is allocated for it in the program generated by the compiler. The program accesses the object using the address specified in the clause.

An address clause is not allowed for task objects, nor for unconstrained records whose size is greater than 8 kb.

6.2 Address Clauses for Program Units

Address clauses for program units are not implemented in the current version of the compiler.

6.3 Address Clauses for Entries

Address clauses for entries are not implemented in the current version of the compiler.

6.4 Address Clauses for Constants

Address clauses for constants are not implemented in the current version of the compiler.

7 UNCHECKED CONVERSIONS

Unconstrained arrays are not allowed as target types. Unconstrained record types without defaulted discriminants are not allowed as target types. Access to unconstrained arrays are not allowed as target or source types.

If the source and the target types are each scalar or access types, the sizes of the objects of the source and target types must be equal.

If a composite type is used either as source type or as target type this restriction on the size does not apply.

If the source and the target types are each of scalar or access type or if they are both of composite type, the effect of the function is to return the operand.

In other cases the effect of unchecked conversion can be considered as a copy:

- if an unchecked conversion is achieved of a scalar or access source type to a composite target type, the result of the function is a copy of the source operand: the result has the size of the source.

- if an unchecked conversion is achieved of a composite source type to a scalar or access target type, the result of the function is a copy of the source operand: the result has the size of the target.

8 INPUT-OUTPUT CHARACTERISTICS

In this part of the Appendix the implementation-specific aspects of the input-output system are described.

8.1 Introduction

In Ada, input-output operations (IO) are considered to be performed on *objects* of a certain file type rather than being performed directly on external files. An external file is anything external to the program that can produce a value to be read or receive a value to be written. Values transferred for a given file must be all of one type.

Generally, in Ada documentation, the term *file* refers to an object of a certain file type, whereas a physical manifestation is known as an *external file*. An external file is characterized by

- Its NAME, which is a string defining a legal path name under the current version of the operating system.

- Its FORM, which gives implementation-dependent information on file characteristics.

Both the NAME and THE FORM appear explicitly as parameters of the Ada CREATE and OPEN procedures. Though a file is an object of a certain file type, ultimately the object has to correspond to an external file. Both CREATE and OPEN associate a NAME of an external file (of a certain FORM) with a program file object.

Ada IO operations are provided by means of standard packages [14].

SEQUENTIAL_IO	A generic package for sequential files of a single element type.
DIRECT_IO	A generic package for direct (random) access files.
TEXT_IO	A generic package for human readable (text, ASCII) files.
IO_EXCEPTIONS	A package which defines the exceptions needed by the above three packages.

The generic package LOW_LEVEL_IO is not implemented in this version.

The upper bound for index values in DIRECT_IO and for line, column and page numbers in TEXT_IO is given by

COUNT'LAST = 2**31 -1

The upper bound for field widths in TEXT_IO is given by

FIELD'LAST = 255

8.2 The FORM Parameter

The FORM parameter of both the CREATE and OPEN procedures in Ada specifies the characteristics of the external file involved.

The CREATE procedure establishes a new external file, of a given NAME and FORM, and associates it with a specified program file object. The external file is created (and the file object set) with a specified (or default) file mode. If the external file already exists, the file will be erased. The exception USE_ERROR is raised if the file mode is IN_FILE.

Example:

```
CREATE(F, OUT_FILE, "MY_FILE",
          FORM =>
          "WORLD => READ, OWNER => READ_WRITE");
```

The OPEN procedure associates an existing external file, of a given NAME and FORM, with a specified program file object. The procedure also sets the current file mode. If there is an inadmissible change of mode, then the Ada exception USE_ERROR is raised.

The FORM parameter is a string, formed from a list of attributes, with attributes separated by commas (,). The string is not case sensitive (so that, for example, *HERE* and *here* are treated alike). (FORM attributes are distinct from Ada attributes.)

The general form of each attribute is a keyword followed by => and then a qualifier. The arrow and qualifier may sometimes be omitted. The format for an attribute specifier is thus either of

KEYWORD

KEYWORD => QUALIFIER