DTIC FILE COPY

(2)

UNCLASSIFIED

Date Entered

ION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
AD-A216 384 12. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
Ada Compiler Validation Summary Report: Alsys, AlsyCOMP_030, Version 4.2, VAX 6210 (host) to Intel iSBC 386/31 (target), 890927A1.10172	5. TYPE OF REPORT & PERIOD COVERED 27 Sept 89 - 1 Dec 90
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) AFNOR, Paris, France.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS AFNOR, Paris, France.	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE
	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) AFNOR, Paris, France.	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 if different from Report)

UNCLASSIFIED

DTIC ELECTE JAN 03 1990 S D D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability (ACVC), Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Alsys, AlsyCOMP_030, Version 4.2; AFNOR; France; VAX 6210 under VMS 5.0 (host) to Intel iSBC 386/31 under ARTK v4.2 (target); ACVC 1.10.

90 01 03 015

AVF Control Number: AVF-VSR-AFNOR-89-07

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890927A1.10172
Alsys
AlsyCOMP_030, Version 4.2
VAX 6210 Host and Intel iSBC 386/31 Target

Completion of On-Site Testing:
27 September 1989

Prepared By:
AFNOR
Tour Europe
Cedex 7
F-92080 Paris la Défense

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

Ada Compiler Validation Summary Report:

Compiler Name: AlsyCOMP_030, Version 4.2

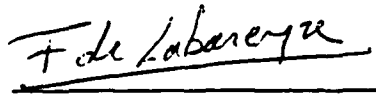
Certificate Number: 890927A1.10172

Host: VAX 6210 under VMS 5.0

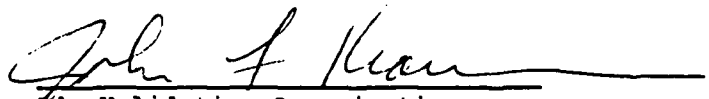
Target: Intel iSBC 386/31 under ARTK v4.2

Testing Completed 27 September 1989 Using ACVC 1.10

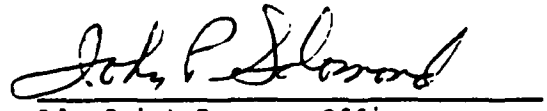
This report has been reviewed and is approved.



AFNOR
Fabrice Garnier de Labareyre
Tour Europe
Cedex 7
F-92080 Paris la Défense



Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311



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

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

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 27 September 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-92080 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 and Guidelines, Ada Joint Program Office, 1 January 1987.

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

4. Ada Compiler Validation Capability User's Guide, December 1986

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 and Guidelines.
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 may legitimately 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_030, Version 4.2

ACVC Version: 1.10

Certificate Number: 890927A1.10172

Host Computer:

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

Target Computer:

Machine:	
Board:	Intel iSBC 386/31
CPU:	Intel 80386
Bus:	Multibus I
I/O:	Intel 8251
Timer:	Intel 8254
Operating System:	ARTK v4.2
Memory Size:	2 Mb
Communications Network:	RS 232 serial connection

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 raise NUMERIC_ERROR or CONSTRAINT_ERROR when a value exceeds SYSTEM.MAX_INT . This implementation raises CONSTRAINT_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) CONSTRAINT_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) CONSTRAINT_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 apparently round to even. (See tests C46012A..Z.) (26 tests)

(2) The method used for rounding to longest integer is apparently round to even. (See tests C46012A..Z.) (26 tests)

(3) The method used for rounding to integer in static universal real expressions is apparently round to even. (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 CONSTRAINT_ERROR . (See test C36003A.)

(2) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

(3) CONSTRAINT_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 CONSTRAINT_ERROR when the array type is declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (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 CONSTRAINT ERROR when the array type is declared.

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 [illegible])

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

1. Input and output.

(3) The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO.

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 584 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 50 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	1131	1750	17	16	46	3089
Inapplicable	0	7	565	0	12	0	584
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	332	137	36	252	259	76	3089
Inappl	14	72	135	3	0	0	5	0	0	0	0	110	245	584
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:

A39005G	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, 584 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.

- 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.

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

. 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

. 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.

. CE2103A is not applicable because USE_ERROR is raised on a CREATE of an instantiation of SEQUENTIAL_IO with an ILLEGAL EXTERNAL FILE NAME.

. CE2103B is not applicable because USE_ERROR is raised on a CREATE of an instantiation of DIRECT_IO with an ILLEGAL EXTERNAL FILE NAME.

. CE3107A is not applicable because USE_ERROR is raised on a CREATE of a file of type TEXT_IO.FILE_TYPE with an ILLEGAL EXTERNAL FILE NAME.

The following 242 tests are inapplicable because sequential, text, and direct access files are not supported:

CE2102A..C (3 tests)	CE2102G..H (2 tests)
CE2102K	CE2102N..Y (12 tests)
CE2103C..D (2 tests)	CE2104A..D (4 tests)
CE2105A..B (2 tests)	CE2106A..B (2 tests)
CE2107A..H (8 tests)	CE2107L
CE2108A..H (8 tests)	CE2109A..C (3 tests)
CE2110A..D (4 tests)	CE2111A..I (9 tests)
CE2115A..B (2 tests)	CE2201A..C (3 tests)
EE2201D..E (2 tests)	CE2201F..N (9 tests)
CE2204A..D (4 tests)	CE2205A
CE2208B	CE2401A..C (3 tests)
EE2401D	EE2401G
CE2401E..F (2 tests)	CE2401H..L (5 tests)
CE2404A..B (2 tests)	CE2405B
CE2406A	CE2407A..B (2 tests)
CE2408A..B (2 tests)	CE2409A..B (2 tests)
CE2410A..B (2 tests)	CE2411A
CE3102A..B (2 tests)	EE3102C
CE3102F..H (3 tests)	CE3102J..K (2 tests)
CE3103A	CE3104A..C (3 tests)
CE3107B	CE3108A..B (2 tests)
CE3109A	CE3110A
CE3111A..B (2 tests)	CE3111D..E (2 tests)
CE3112A..D (4 tests)	CE3114A..B (2 tests)
CE3115A	EE3203A
CE3208A	EE3301B
CE3302A	CE3305A
CE3402A	EE3402B
CE3402C..D (2 tests)	CE3403A..C (3 tests)
CE3403E..F (2 tests)	CE3404B..D (3 tests)
CE3405A	EE3405B
CE3405C..D (2 tests)	CE3406A..D (4 tests)
CE3407A..C (3 tests)	CE3408A..C (3 tests)
CE3409A	CE3409C..E (3 tests)
EE3409F	CE3410A
CE3410C..E (3 tests)	EE3410F
CE3411A	CE3411C
CE3412A	
EE3412C	CE3413A
CE3413C	CE3602A..D (4 tests)
CE3603A	CE3604A..B (2 tests)
CE3605A..E (5 tests)	CE3606A..B (2 tests)
CE3704A..F (6 tests)	CE3704M..O (3 tests)
CE3706D	CE3706F..G (2 tests)
CE3804A..P (16 tests)	CE3805A..B (2 tests)
CE3806A..B (2 tests)	CE3806D..E (2 tests)
CE3806G..H (2 tests)	CE3905A..C (3 tests)
CE3905L	CE3906A..C (3 tests)
CE3906E..F (2 tests)	

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 50 tests.

The test EA3004D when run as it is, the implementation fails to detect an error on line 27 of test file EA3004D6M (line 115 of "cat -n ea3004d*"). 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.

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

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.

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_030, Version 4.2 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_030, Version 4.2 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
Target computer:	Intel iSBC 386/31
Target operating system:	ARTK v4.2
Compiler:	AlsyCOMP_030, Version 4.2
Pre-linker:	built-in and Alsys proprietary
Linker:	Phar Lap Linkloc v2.1c
Loader/Downloader:	built-in and Alsys proprietary
Runtime System:	ARTE v4.2

The host and target computers were linked via RS 232 serial connection.

A tape 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 tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the tape.

The contents of the tape were not loaded directly onto the host computer. The loading was made using a network composed of Ethernet and NFS.

After the test files were loaded to disk, the full set of tests was compiled and linked on the VAX 6210, then all executable images were transferred to the Intel iSBC 386/31 via RS 232 serial connection and run. Results were printed from the host computer.

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
GENERIC=STUBS	Code of generic instantiation is placed in separate units
CALLS=INLINE	The pragma INLINE are taken into account

Tests were compiled, linked, and executed (as appropriate) using a single host and target computer. Test output, compilation listings, and job logs were captured on data cartridge 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 27 September 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Alsys has submitted the following Declaration of Conformance concerning the AlsyCOMP_030, Version 4.2 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_030, Version 4.2
Host Architecture: VAX 6210
Host OS and Version: VMS 5.0
Target Architecture: Intel iSBC 386/31
Target OS and Version: ARTK v4.2

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 shall be made only in the owner's corporate name.

Date: 21 Nov. 89

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: 21 Nov. 89

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.	655360
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	I_80X86
$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_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAD_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.	2_097_151.999_023_437_51
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	3_000_000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	10
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name specifying a non existent directory	ILLEGAL\!@#$%^&*()/_+~
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name different from $ILLEGAL_EXTERNAL_FILE_NAME1	!@#$%^&*()?/)(*&^\!@#$%^
$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.	-2_097_152.5
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-3_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.	2147483648
$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.	I_80X86
$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.	655360
$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.	I_80X86
$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.	1.0/18.2

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).

B971C2E
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
These 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_030, Version 4.2 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+127
..
   2#1.111_1111_1111_1111_1111_1111#E+127;

 type LONG_FLOAT is digits 15 range
-2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E1023
 ..
 2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E1023;

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

 ...

 end STANDARD;
```

Alsys Ada

VAX/VMS to 80386 Compiler

APPENDIX F

Version 4.2

Alsys S.A.
29, Avenue de Versailles
78170 La Celle St. Cloud, France

Alsys Inc.
One Burlington Business Center
67 South Bedford Street
Burlington, MA 01803-5152

Alsys Ltd.
Partridge House, Newtown Road
Henley-on-Thames,
Oxfordshire RG9 1EN, England

Printed: August 1989

TABLE OF CONTENTS

Section 1

Implementation-Dependent Pragmas

1.1 INLINE

Pragma INLINE is fully supported; however, it is not possible to inline a subprogram in a declarative part.

1.2 INTERFACE

Ada programs can interface with subprograms written in Assembler and other languages through the use of the predefined pragma INTERFACE and the implementation-defined pragma INTERFACE_NAME.

Pragma INTERFACE specifies the name of an interfaced subprogram and the name of the programming language for which parameter passing conventions will be generated. Pragma INTERFACE takes the form specified in the RM:

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

where,

- *language_name* is ASSEMBLER or Ada.
- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.

The only language names accepted by pragma INTERFACE are ASSEMBLER, and Ada. The full implementation requirements for writing pragma INTERFACE subprograms are described in the *Application Developer's Guide.*

The language name used in the pragma INTERFACE does not have to have any relationship to the language actually used to write the interfaced subprogram. It is used only to tell the Compiler how to generate subprogram calls; that is, what kind of parameter passing techniques to use. The programmer can interface Ada programs with subroutines written in any other (compiled) language by understanding the mechanisms used for parameter passing by the Alsys Ada Compiler and the corresponding mechanisms of the chosen external language.

1.3 INTERFACE_NAME

Pragma INTERFACE_NAME associates the name of the interfaced subprogram with the external name of the interfaced subprogram. If pragma INTERFACE_NAME is not used, then the two names are assumed to be identical. This pragma takes the form:

pragma INTERFACE_NAME (*subprogram_name*, *string_literal*);

where,

- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.
- *string_literal* is the name by which the interfaced subprogram is referred to at link time.

The pragma INTERFACE_NAME is used to identify routines in other languages that are not named with legal Ada identifiers. Ada identifiers can only contain letters, digits, or underscores, whereas the different linkers allow external names to contain other characters, for example, the dollar sign ($) or commercial at sign (@). These characters can be specified in the *string_literal* argument of the pragma INTERFACE_NAME.

The pragma INTERFACE_NAME is allowed at the same places of an Ada program as the pragma INTERFACE. (Location restrictions can be found in section 13.9 of the RM.) However, the pragma INTERFACE_NAME must always occur after the pragma INTERFACE declaration for the interfaced subprogram.

The *string_literal* of the pragma INTERFACE_NAME is passed through unchanged to the object file. However, only the first 31 characters are significant, so it is recommended that the *string_literal* be restricted to 31 characters.

The *Runtime Executive* contains several external identifiers. All such identifiers begin with either the string "ADA@" or the string "ADAS@". Accordingly, names prefixed by "ADA@" or "ADAS@" should be avoided by the user.

Example

```
package SAMPLE_DATA is
    function SAMPLE_DEVICE (X: INTEGER) return INTEGER;
    function PROCESS_SAMPLE (X: INTEGER) return INTEGER;
private
    pragma INTERFACE (ASSEMBLER, SAMPLE_DEVICE);
    pragma INTERFACE (ADA, PROCESS_SAMPLE);
    pragma INTERFACE_NAME (SAMPLE_DEVICE, "DEVIO$GET_SAMPLE");
end SAMPLE_DATA;
```

1.4 INDENT

Pragma INDENT is only used with *AdaReformat*. *AdaReformat* is the Alsys reformatter which offers the functionalities of a pretty-printer in an Ada environment.

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

```
pragma INDENT(OFF);
```

causes *AdaReformat* not to modify the source lines after this pragma, while

```
pragma INDENT(ON);
```

causes *AdaReformat* to resume its action after this pragma.

1.5 Other Pragmas

Pragmas IMPROVE and PACK are discussed in detail in the section on representation clauses and records (Chapter 4).

Pragma PRIORITY is accepted with the range of priorities running from 1 to 10 (see the definition of the predefined package SYSTEM in Section 3). Undefined priority (no pragma PRIORITY) is treated as though it were less than any defined priority value.

In addition to pragma SUPPRESS, it is possible to suppress all checks in a given compilation by the use of the Compiler option CHECKS. (See the *User's Guide*.)

Section 2

Implementation-Dependent Attributes

2.1 P'IS_ARRAY

For a prefix P that denotes any type or subtype, this attribute yields the value TRUE if P is an array type or an array subtype; otherwise, it yields the value FALSE.

2.2 P'RECORD_DESCRIPTOR, P'ARRAY_DESCRIPTOR

These attributes are used to control the representation of implicit components of a record. (See Section 4.8 for more details.)

2.3 E'EXCEPTION_CODE

For a prefix E that denotes an exception name, this attribute yields a value that represents the internal code of the exception. The value of this attribute is of the type INTEGER.

Section 3

Specification of the package SYSTEM

The implementation does not allow the recompilation of package SYSTEM.

```
package SYSTEM is

   --     ******************************
   --     * (1) Required Definitions. *
   --     ******************************

   type NAME is (I_80x86);
   SYSTEM_NAME : constant NAME := I_80x86;

   STORAGE_UNIT : constant := 8;
   MEMORY_SIZE  : constant := 640 * 1024;

   -- System-Dependent Named Numbers:

   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;

   -- For the high-resolution timer, the clock resolution is
   -- 1.0 / 1024.0.
   TICK         : constant := 1.0 / 18.2;

   -- Other System-Dependent Declarations:
   subtype PRIORITY is INTEGER range 1 .. 10;
```

```
-- The type ADDRESS is, in fact, implemented as a
--  386 bit offset²

type ADDRESS is private;
NULL_ADDRESS: constant ADDRESS;


--     **********************************
--     * (2) MACHINE TYPE CONVERSIONS *
--     **********************************

-- If the word / double-word operations below are used on
-- ADDRESS, then MSW yields the segment and LSW yields the
-- offset.
-- In the operations below, a BYTE_TYPE is any simple type
-- implemented on 8-bits (for example, SHORT_SHORT_INTEGER), a WORD_TYPE is
-- any simple type implemented on 16-bits (for example, SHORT_INTEGER), and
-- a DOUBLE_WORD_TYPE is any simple type implemented on
-- 32-bits (for example, INTEGER, FLOAT, ADDRESS).

-- Byte <==> Word conversions:

-- Get the most significant byte:
generic
   type BYTE_TYPE is private;
   type WORD_TYPE is private;
function MSB (W: WORD_TYPE) return BYTE_TYPE;

-- Get the least significant byte:
generic
   type BYTE_TYPE is private;
   type WORD_TYPE is private;
function LSB (W: WORD_TYPE) return BYTE_TYPE;

-- Compose a word from two bytes:
generic
```

```
   type BYTE_TYPE is private;
   type WORD_TYPE is private;
function WORD (MSB, LSB: BYTE_TYPE) return WORD_TYPE;

-- Word <==> Double-Word conversions:

-- Get the most significant word:
generic
   type WORD_TYPE is private;
   type DOUBLE_WORD_TYPE is private;
function MSW (W: DOUBLE_WORD_TYPE) return WORD_TYPE;

-- Get the least significant word:
generic
   type WORD_TYPE is private;
   type DOUBLE_WORD_TYPE is private;
function LSW(W: DOUBLE_WORD_TYPE) return WORD_TYPE;

-- Compose a DATA double word from two words.
generic
   type WORD_TYPE is private;
   -- The following type must be a data type
   -- (for example, LONG_INTEGER):
   type DATA_DOUBLE_WORD is private;
function DOUBLE_WORD (MSW, LSW: WORD_TYPE) return DATA_DOUBLE_WORD;


--     ******************************
--     * (3) OPERATIONS ON ADDRESS *
--     ******************************

-- You can get an address via 'ADDRESS attribute or by
-- Some addresses are used by the Compiler.  For example,
-- the display is located at the low end of the DS segment.
-- Note that no operations are defined to get the values of
-- the segment registers, but if it is necessary an
```

```
   -- interfaced function can be written.

   generic
      type OBJECT is private;
   function FETCH_FROM_ADDRESS (FROM: ADDRESS) return OBJECT;

   generic
      type OBJECT is private;
   procedure ASSIGN_TO_ADDRESS (OBJ: OBJECT; TO: ADDRESS);

private

   ...

end SYSTEM;
```

Section 4

Support for 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.

Representation clauses on derived record types or derived tasks types are not supported.

Size representation clauses on types derived from private types are not supported when the derived type is declared outside the private part of the defining package.

4.1 Enumeration Types

Minimum size of an enumeration subtype

The minimum possible size of an enumeration subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

Size of an enumeration subtype

When no size specification is applied to an enumeration type or first named subtype, the objects of that type or first named subtype are represented as signed machine integers.

When a size specification is applied to an enumeration type, this enumeration type and each of its subtypes has the size specified by the length clause.

The Alsys compiler fully implements size specifications. Nevertheless, as enumeration values are coded using integers, the specified length cannot be greater than 32 bits.

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.

4.2 Integer Types

Minimum size of an integer subtype

The minimum possible size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

Size of an integer subtype

The sizes of the predefined integer types SHORT_SHORT_INTEGER, SHORT_INTEGER and INTEGER are respectively 8, 16 and 32 bits.

When a size specification is applied to an integer type, this integer type and each of its subtypes has the size specified by the length clause.

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.

4.3 Floating Point Types

The minimum possible size of a floating point subtype is 32 bits if its base type is FLOAT or a type derived from FLOAT; it is 64 bits if its base type is LONG_FLOAT or a type derived from LONG_FLOAT.

The sizes of the predefined floating point types FLOAT and LONG_FLOAT are respectively 32 and 64 bits.

The size of a floating point type and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

The only size that can be specified for a floating point type or first named subtype using a size specification is its usual size (32 or 64 bits).

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

4.4 Fixed Point Types

Minimum size of a fixed point subtype

The minimum possible size of a fixed point subtype is the minimum number of binary digits that is necessary for representing the values of the range of the subtype using the small of the base type.

Size of a fixed point subtype

The sizes of the predefined fixed point types SHORT_FIXED, FIXED and LONG_FIXED are respectively 8, 16 and 32 bits.

When no size specification is applied to a fixed point type or to its first named subtype, its size and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

The Alsys compiler fully implements size specifications. Nevertheless, as fixed point objects are represented using machine integers, the specified length cannot be greater than 32 bits.

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.

4.5 Access Types and Collections

Access Types and Objects of Access Types

The only size that can be specified for an access type using a size specification is its usual size (32 bits).

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.

Collection Size

As described in RM 13.2, a specification of collection size can be provided in order to reserve storage space for the collection of an access type.

When no STORAGE_SIZE specification applies to an access type, no storage space is reserved for its collection, and the value of the attribute STORAGE_SIZE is then 0.

The maximum size is limited by the amount of memory available.

4.6 Task Types

Storage for a task activation

As described in RM 13.2, a length clause can be used to specify the storage space (that is, the stack size) for the activation of each of the tasks of a given type. Alsys also allows the task stack size, for all tasks, to be established using a Binder option. If a length clause is given for a task type, the value indicated at bind time is ignored for this task type, and the length clause is obeyed. When no length clause is used to specify the storage space to be reserved for a task activation, the storage space indicated at bind time is used for this activation.

A length clause may not be applied to a derived task type. The same storage space is reserved for the activation of a task of a derived type as for the activation of a task of the parent type.

The minimum size of a task subtype is 32 bits.

A size specification has no effect on a task type. The only size that can be specified using such a length clause is its usual size (32 bits).

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.

4.7 Array Types

Each array is allocated in a contiguous area of storage units. All the components have the same size. A gap may exist between two consecutive components (and after the last one). All the gaps have the same size.

Size of an array subtype

The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of the gaps (if any). If the subtype is unconstrained, the maximum number of components is considered.

The size of an array subtype cannot be computed at compile time

- if it has non-static constraints or is an unconstrained array type with non-static index subtypes (because the number of components can then only be determined at run time).
- if the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).

The consequence of packing an array type is thus to reduce its size.

If the components of an array are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static, the compiler ignores any pragma PACK applied to the array type but issues a warning message. Apart from this limitation, array packing is fully implemented by the Alsys compiler.

A size specification applied to an array type or first named subtype has no effect. The only size that can be specified using such a length clause is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of an array is as expected by the application.

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.

4.8 Record Types

Size of a record subtype

Unless a component clause specifies that a component of a record type has an offset or a size which cannot be expressed using storage units, the size of a record subtype is rounded up to a whole number of storage units.

The size of a constrained record subtype is obtained by adding the sizes of its components and the sizes of its gaps (if any). This size is not computed at compile time

- when the record subtype has non-static constraints,
- when a component is an array or a record and its size is not computed at compile time.

The size of an unconstrained record subtype is obtained by adding the sizes of the components and the sizes of the gaps (if any) of its largest variant. If the size of a component or of a gap cannot be evaluated exactly at compile time an upper bound of this size is used by the compiler to compute the subtype size.

A size specification applied to a record type or first named subtype has no effect. The only size that can be specified using such a length clause is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of a record is as expected by the application.

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.

Section 5

Conventions for Implementation-Generated Names

The Alsys Ada Compiler may add fields to record objects and have descriptors in memory for record or array objects. These fields are accessible to the user through implementation-generated attributes (See Section 2.3).

Section 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 the compiler does not cause storage to be allocated for the object. The program accesses the object using the address specified in the clause. It is the responsibility of the user therefore to make sure that a valid allocation of storage has been done at the specified address.

An address clause is not allowed for task objects, for unconstrained records whose size is greater than 8k bytes or for a constant.

There are a number of ways to compose a legal address expression for use in an address clause. The most direct ways are:

- For the case where the memory is defined in Ada as another object, use the 'ADDRESS attribute to obtain the argument for the address clause for the second object.
- For the case where an absolute address is known to the programmer, instantiate the generic function SYSTEM.REFERENCE on a 16 bit unsigned integer type (either from package UNSIGNED, or by use of a length clause on a derived integer type or subtype) and on type SYSTEM.ADDRESS. Then the values of the desired segment and offset can be passed as the actual parameters to the instantiated function in the simple expression part of the address clause. See Section 3 for the specification of package SYSTEM.
- For the case where the desired location is memory defined in assembly or another non-Ada language (is relocatable), an interfaced routine may be used to obtain the appropriate address from referencing information known to the other language.

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 Interrupt Entries

Address clauses for interrupt entries are supported.

Section 7

Unchecked Conversions

Unchecked conversions are allowed between any types provided the instantiation of UNCHECKED_CONVERSION is legal Ada. It is the programmer's responsibility to determine if the desired effect is achieved.

If the target type has a smaller size than the source type then the target is made of the least significant bits of the source.

Section 8

Input-Output Packages

The *RM* defines the predefined input-output packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO, and describes how to use the facilities available within these packages. The *RM* also defines the package IO_EXCEPTIONS, which specifies the exceptions that can be raised by the predefined input-output packages.

In addition the *RM* outlines the package LOW_LEVEL_IO, which is concerned with low-level machine-dependent input-output, such as would possibly be used to write device drivers or access device registers. LOW_LEVEL_IO has not been implemented. However the user provides low level drivers to support a specific hardware.

8.1 Accessing Devices

The user must provide a description of the input-output devices and provide drivers for them. These drivers will permit the various devices to be accessed through the standard IO packages, namely TEXT_IO, SEQUENTIAL_IO and DIRECT_IO.

All necessary information to describe the devices and build the drivers is available in the *Cross Development Guide.*

8.2 File Names and Form

The function FORM always returns a null string.

The FORM parameter of the OPEN procedures must be a null string.

8.3 Sequential Files

For sequential access the file is viewed as a sequence of values that are transferred in the order of their appearance (as produced by the program or run-time environment). This is sometimes called a *stream* file in other operating systems. Each object in a sequential file has the same binary representation as the Ada object in the executable program.

8.4 Direct Files

For direct access the file is viewed as a set of elements occupying consecutive positions in a linear order. The position of an element in a direct file is specified by its index, which is an integer of subtype POSITIVE_COUNT.

DIRECT_IO only allows input-output for constrained types. If DIRECT_IO is instantiated for an unconstrained type, all calls to CREATE or OPEN will raise USE_ERROR. Each object in a direct file will have the same binary representation as the Ada object in the executable program. All elements within the file will have the same length.

8.5 Text Files

Text files are used for the input and output of information in ASCII character form. Each text file is a sequence of characters grouped into lines, and lines are grouped into a sequence of pages.

All text file column numbers, line numbers, and page numbers are values of the subtype POSITIVE_COUNT.

Note that due to the definitions of line terminator, page terminator, and file terminator in the *RM*, and the method used to mark the end of file, some ASCII files do not represent well-formed TEXT_IO files.

A text file is buffered by the *Alsys Runtime Executive* unless it names a device, as indicated by the function ADA@IO_IOCTL (See the *Cross Development Guide*).

If STANDARD_INPUT and STANDARD_OUTPUT are the console input and output devices (as indicated by the function ADA@IO_IOCTL), prompts written to STANDARD_OUTPUT with the procedure PUT will appear before (or when) a GET (or GET_LINE) occurs.

The functions END_OF_PAGE and END_OF_FILE always return FALSE when the file is a device, which includes the use of STANDARD_INPUT when it corresponds to the console input device. Programs which would like to check for end of file when the file may be a device should handle the exception END_ERROR instead, as in the following example:

Example

```
begin
    loop
        -- Display the prompt:
        TEXT_IO.PUT ("--> ");
        -- Read the next line:
        TEXT_IO.GET_LINE (COMMAND, LAST);
        -- Now do something with COMMAND (1 .. LAST)
    end loop;
exception
    when TEXT_IO.END_ERROR =>
        null;
end;
```

END_ERROR is raised for STANDARD_INPUT when ^Z (ASCII.SUB) is entered at the console input device.

8.6 The Need to Close a File Explicitly

The *Alsys Runtime Executive* will flush all buffers and close all open files when the program is terminated, either normally or through some exception.

However, the *RM* does not define what happens when a program terminates without closing all the opened files. Thus a program which depends on this feature of the *Alsys Runtime Executive* might have problems when ported to another system.

8.7 Limitation on the procedure RESET

An internal file opened for input cannot be RESET for output. However, an internal file opened for output can be RESET for input, and can subsequently be RESET back to output.

8.8 Sharing of External Files and Tasking Issues

Several internal files can be associated with the same external file only if all the internal files are opened with mode IN_MODE. However, if a file is opened with mode OUT_MODE and then changed to IN_MODE with the RESET procedure, it cannot be shared.

Care should be taken when performing multiple input-output operations on an external file during tasking because the order of calls to the I/O primitives is unpredictable. For example, two strings output by TEXT_IO.PUT_LINE in two different tasks may appear in the output file with interleaved characters. Synchronization of I/O in cases such as this is the user's responsibility.

The TEXT_IO files STANDARD_INPUT and STANDARD_OUTPUT are shared by all tasks of an Ada program.

If TEXT_IO.STANDARD_INPUT corresponds to the console input device, it will not block a program on input. All tasks not waiting for input will continue running.