AVF Control Number: AVF-VSR-015
SZT-AVF-015

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 880322I1.09047
Tartan Laboratories Incorporated
Tartan Ada VMS/VMS
Version 2.0V
MicroVax II

Completion of On-Site Testing:
88-03-20

Prepared By:
IABG m.b.H., Dept SZT
Einsteinstrasse 20
8012 Ottobrunn
Federal Republic of Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETEING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A209 752	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Ada Compiler Validation Summary Report: Tartan Laboratories Incorporated, Tartan Ada VMS/VMS Version 2.0V, MicroVAX II (Host and Target).	5. TYPE OF REPORT & PERIOD COVERED 20 March 1988 to 20 March 1989 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) IABG, Ottobrunn, Federal Republic of Germany.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS IABG, Ottobrunn, Federal Republic of Germany.	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 20 March 1988 13. NUMBER OF PAGES 44 p.
14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)* IABG, Ottobrunn, Federal Republic of Germany.	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

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)*

Tartan Ada VMS/VMS Version 2.0V, Tartan Laboratories Incorporated, IABG, MicroVAX II under MicroVMS Version 4.6, ACVC 1.9.

Ada Compiler Validation Summary Report:

Compiler Name: Tartan Ada VMS/VMS
Compiler Version: Version 2.0V

Certificate Number: #880322I1.09047

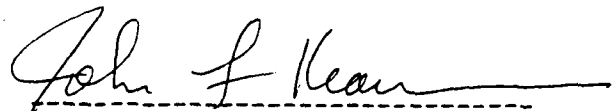
Host: MicroVax II under
microVMS, Version 4.6

Target: same as host

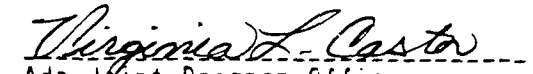
Testing Completed 88-03-20 Using ACVC 1.9

This report has been reviewed and is approved.

IABG m.b.H., Dept SZT
Dr. H. Hummel
Einsteinstrasse 20
8012 Ottobrunn
Federal Republic of Germany



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



Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

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 feature 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 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 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 88-03-20 at Tartan Laboratories Incorporated in Pittsburgh.

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:

IABG m.b.H., Dept SZT
Einsteinstrasse 20
8012 Ottobrunn
Federal Republic of Germany

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 for which a compiler generates code.
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 test	An ACVC test found to be incorrect and not used to check 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 compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. 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 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.

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.

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 the tests in the ACVC follow 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: Tartan Ada VMS/VMS, Version 2.0V

ACVC Version: 1.9

Certificate Number: 880322I1.09047

Host and Target Computer:

Machine:	MicroVax II
Operating System:	microVMS Version 4.6
Memory Size:	9 Megabytes

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:

. Capacities.

The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

. Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

. Predefined types.

This implementation supports the additional predefined type LONG_FLOAT in the package STANDARD. (See tests B86001C and B86001D.)

. Based literals.

An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR (See test E24101A.)

. Expression evaluation.

Apparently some default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

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

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

Sometimes 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.)

No exception 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.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

. Rounding.

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

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

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

. 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:

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

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

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

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

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

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.)

In assigning one-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be 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.)

. Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be 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.)

. Aggregates.

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.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

Not all choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

. Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

. Pragmas.

The pragma INLINE is not supported for procedures. The pragma INLINE is not supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

. Input/output.

The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external file for sequential I/O for both reading and writing. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107F..I (5 tests), CE2110B, and CE2111H.)

An internal sequential access file and an internal direct access file cannot be associated with a single external file for writing. (See test CE2107E.)

An external file associated with more than one internal file cannot be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

Temporary sequential files are not given names. Temporary direct files are not given names. (See tests CE2108A and CE2108C.)

- Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See section 3.5 for restrictions. See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See section 3.5 for restrictions. See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See section 3.5 for restrictions. See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 367 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 290 executable tests that use floating-point precision exceeding that supported by the implementation . Modifications to the code, processing, or grading for 75 tests were required to successfully demonstrate the test objective. (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	108	1044	1505	17	10	44	2728
Inapplicable	2	7	348	0	8	2	367
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

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	182	469	476	242	166	98	141	327	127	36	232	3	229	2728
Inapplicable	22	103	198	6	0	0	2	0	10	0	2	0	24	367
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C	C35904A
C35904B	C35A03E	C35A03R	C37213H	C37213J	C37215C
C37215E	C37215G	C37215H	C38102C	C41402A	C45332A
C45614C	A74106C	C85018B	C87B04B	CC1311B	BC3105A
AD1A01A	CE2401H	CE3208A			

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, 367 tests were inapplicable for the reasons indicated:

- E28002D and E28005D use pragmas LIST and PAGE which are ignored by this compiler. This behaviour was ruled acceptable by the AVO (dated 88-01-14).

. C35702A uses SHORT_FLOAT which is not supported by this implementation.

. The following tests use SHORT_INTEGER, which is not supported by this compiler:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	

. The following tests use LONG_INTEGER, which is not supported by this compiler:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

. C45231D requires a macro substitution for any predefined numeric types other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.

. C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

. C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

. B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

. C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

. CA2009A, CA2009C..D (2 tests), CA2009F, BC3009B..C (2 tests) This compiler enforces the following two rules concerning declarations and proper bodies which are individual compilation units:

  o generic bodies must be compiled and completed before their instantiation.

- o recompilation of a generic body or any of its transitive subunits makes all units obsolete which instantiate that generic body.

These rules are enforced whether the compilation units are in separate compilation files or not. The rules are in conflict with the said tests. AI408 and AI506 allow this behaviour until June 1989.

. CA3004E, EA3004C, and LA3004A use the INLINE pragma for procedures, which is not supported by this compiler.

. CA3004F, EA3004D, and LA3004B use the INLINE pragma for functions, which is not supported by this compiler.

. AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

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

. CE2107B..D (3 tests), CE2107G..I (3 tests), CE2111H, CE3111B..E (4 tests), and CE3114B, and are inapplicable because multiple internal files cannot be associated with the same external file for the operations attempted in these tests. The proper exception is raised when operations are attempted which are not supported.

. CE2108A, CE2108C are inapplicable because temporary sequential and direct files are not given names.

. CE2105A..B (2 tests) are inapplicable because files of mode IN_FILE cannot be created for SEQUENTIAL_IO and DIRECT_IO.

. CE2108A and CE2108C are inapplicable because temporary files do not have names for SEQUENTIAL_IO and DIRECT_IO.

. The following 290 tests require a floating-point accuracy that exceeds the maximum of 9 digits supported by this implementation:

C24113F..Y (20 tests) C35705F..Y (20 tests)

C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (22 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (22 tests)
C45524F..Z (22 tests)	C45621F..Z (22 tests)
C45641F..Y (20 tests)	C46012F..Z (22 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 75 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A B24007A B24009A B25002B B32201A B33301A B34007H
B35701A B36171A B36201A B37101A B37102A B37201A B37202A
B37203A B37302A B38003A B38003B B38008A B38008B B38009A
B38009B B38103A B38103B B38103C B38103D B38103E B43201C
B43202C B44001A B48002A B48002B B48002D B48002E B48002G
B48003E B49003A B49005A B49006A B49007A B49009A B4A010C
B54A20A B54A25A B58002A B58002B B59001A B59001C B59001I
B62006C B64001A B67001A B67001B B67001C B67001D B74103E
B74104A B85007C B91003B B91005A B95001A B95003A B95007B
B95031A B95074E BC1002A BC1109A BC1109C BC1206A BC3005B
BC3009C

For the two tests BC3204C and BC3205D the compilation order was changed to

BC3204C0, ..C1, ..C2, ..C3, ..C4, .. C5, ..C6, ..C3M

and

BC3205D0, ..D2, ..D1M

respectively. This change was necessary because of the compiler's rules for separately compiled generic units. When processed in this order the expected error messages were produced for BC3204CM and BC3205D1M, respectively.

The compilation files for BC3204D and BC3205C consist of several compilation units each. The compilation units for the main procedures are near the beginning of the files. When processing these files unchanged a link error is reported instead of the expected compilation error because of the compiler's rules for separately compiled generic units. Therefore, the compilation files were changed by appending copies of the main procedures to the end of these files. When processing these second occurences of the main procedures the expected error messages were generated by the compiler.

Test E28002B checks that predefined or unrecognized pragmas may have arguments involving overloaded identifiers without enough contextual information to resolve the overloading. It also checks the correct processing of pragma LIST. This compiler ignores pragma LIST so that this part of the test was not taken into account when grading the test as passed.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the Tartan Ada VMS/VMS 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 Tartan Ada VMS/VMS using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a MicroVax II operating under microVMS, Version 4.6.

A magnetic 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 magnetic tape. Tests requiring modifications during the prevalidation testing were not included in their modified form on the magnetic tape. They were provided by Tartan Laboratories Incorporated and reviewed by the validation team.

The contents of the magnetic tape were not loaded directly onto the host computer. The tests were read on a VAX-750 and transferred to the host computer using an ether-net connection.

After the test files were loaded to disk, the full set of tests was compiled on the MicroVax II, and all executable tests were linked and run. Results were transferred to the VAX-750 by ether-net where they could be printed, checked and archived.

The compiler was tested using command scripts provided by Tartan Laboratories Incorporated and reviewed by the validation team. The compiler was tested using all default switch settings.

Tests were compiled, linked, and executed (as appropriate) using a single host computer. Test output, compilation listings, and job logs were captured on magnetic tape 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 Tartan Laboratories Incorporated in Pittsburgh and was completed on 88-03-20.

APPENDIX A

DECLARATION OF CONFORMANCE

Tartan Laboratories Incorporated has submitted the following Declaration of Conformance concerning the Tartan Ada VMS/VMS.

DECLARATION OF CONFORMANCE

DECLARATION OF CONFORMANCE

Compiler Implementor: Tartan Laboratories Incorporated
Ada Validation Facility: IABG m.b.H., Dept. SZT
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name: Tartan Ada VMS/VMS
Base Compiler Version: Version 2.0V
Host and Target Computer: MicroVAX II under MicroVMS 4.6

Implementor's Declaration

I, the undersigned, representing Tartan Laboratories Incorporated, 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 Tartan Laboratories Incorporated 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.

Donald L Evans Date: 2 April 88
Tartan Laboratories Incorporated
Donald L. Evans, President

Owner's Declaration

I, the undersigned, representing Tartan Laboratories Incorporated, 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 further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. 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.

Donald L Evans Date: 2 April 88
Tartan Laboratories Incorporated
Donald L. Evans, President

APPENDIX B

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 Tartan Ada VMS/VMS, Version 2.0V, are described in the following sections, which discuss topics in Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is
    ...
    type INTEGER is range -2147483648 .. 2147483647;
    type FLOAT is digits 6 range -16#0.7FFF_FF8#E+32 .. 16#0.7FFF_FF8#E+32;
    type LONG_FLOAT is digits 9 range -16#0.7FFF_FFFF_FFFF_FF8#E+32 ..
       16#0.7FFF_FFFF_FFFF_FF8#E+32 ;
    type DURATION is delta 0.02 range -86400.0 .. 86400.0;
       -- DURATION'SMALL = 2#1.0#E-6
    ...
end STANDARD;
```

CHAPTER 5
APPENDIX F TO MIL-STD-1815A

This chapter contains the required Appendix F to Military Standard, Ada Programming Language, ANSI/MIL-STD-1815A (American National Standards Institute, Inc., February 17, 1983).

5.1. PRAGMAS

5.1.1. Predefined Pragmas

This section summarizes the effects of and restrictions on predefined pragmas.

- Access collections are not subject to automatic storage reclamation so pragma CONTROLLED has no effect. Space deallocated by means of UNCHECKED_DEALLOCATION will be reused by the allocation of new objects.

- Pragma ELABORATE is fully supported.

- Pragma INLINE, is supported but has an effect on the generated code only when the call appears within the same compilation unit as the body of the in-lined subprogram.

- Pragma INTERFACE, is not supported. The implementation-defined pragma FOREIGN_BODY (see Section 5.1.2.2) can be used to interface to subprograms written in other languages.

- For global control over listings, compiler option switches should be used rather than pragma LIST.

- Tartan compilers currently optimize both the time and space aspects based on what is best in the local context. Future releases of the compiler will have option switches to decrease the level of sophistication of the optimizations. Because it is generally very difficult to establish global time and space tradeoffs, pragma OPTIMIZE cannot be effectively supported in the form suggested in the LRM.

- Pragma PACK is fully supported.

- The effect of pragma PAGE can be achieved by inserting form-feed characters into the Ada source file.

- Pragma PRIORITY is fully supported.

- The effect of pragma SUPPRESS can be achieved by a global compiler option

switch.

- Future releases of the compiler will support the following pragmas: MEMORY_SIZE, SHARED, STORAGE_UNIT and SYSTEM_NAME.

A warning message will be issued if an unsupported pragma is supplied.

5.1.2. Implementation-Defined Pragmas

Tartan provides the following pragmas.

5.1.2.1. Pragma LINKAGE_NAME

The pragma LINKAGE_NAME associates an Ada entity with a string that is meaningful externally; e.g., to a linkage editor. It takes the form

pragma LINKAGE_NAME (Ada-simple-name, string-constant)

The Ada-simple-name must be the name of an Ada entity declared in a package specification. This entity must be one that has a runtime representation; e.g., a subprogram, exception or object. It may not be a named number or string constant. The pragma must appear after the declaration of the entity in the same package specification.

The effect of the pragma is to cause the string-constant to be used in the generated assembly code as an external name for the associated Ada entity. It is the responsibility of the user to guarantee that this string constant is meaningful to the linkage editor and that no illegal linkname clashes arise.

5.1.2.2. Pragma FOREIGN_BODY

A subprogram written in another language can be called from an Ada program. Pragma FOREIGN_BODY is used to indicate that the body for a non-generic top-level package specification is provided in the form of an object module. The bodies for several subprograms may be contained in one object module.

Use of the pragma FOREIGN_BODY dictates that all subprograms, exceptions and objects in the package are provided by means of a foreign object module.

The pragma is of the form:

pragma FOREIGN_BODY (language_name [, elaboration_routine_name])

The parameter language_name is intended to allow the compiler to identify the calling convention used by the foreign module (but this functionality is not yet in operation). The programmer must ensure that the calling convention and data representation of the foreign body procedures are compatible with those used by the Tartan Ada compiler. Subprograms called by tasks should be reentrant.

The optional elaboration_routine_name argument provides a means to initialize

the package. The routine specified as the elaboration_routine_name, which will be called for the elaboration of this package body, must be a global routine in the object module provided by the user.

A specification that uses this pragma may contain only subprogram declarations, object decarations that use an unconstrained type mark, and number declarations. Pragmas may also appear in the package. The type mark for an object cannot be a task type, and the object declaration must not have an initial value expression. The pragma must be given prior to any declarations within the package specification. If the pragma is not located before the first declaration, or any restriction on the declarations is violated, the pragma is ignored and a warning is generated.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma FOREIGN_BODY. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

Pragma LINKAGE_NAME should be used for all declarations in the package, including any declarations in a nested package specification to be sure that there are no conflicting link names. If pragma LINKAGE_NAME is not used, the cross-reference qualifier, /CROSS_REFERENCE, (see Section 3.2) should be used when invoking the compiler and the resulting cross-reference table of linknames inspected to identify the linknames assigned by the compiler and determine that there are no conflicting linknames (see also Section 3.5).

Example:

```
package Fortran_RunTimes is
  pragma FOREIGN_BODY ("fortran");
  --The language name is a comment only, it is not checked for validity.
end Fortran_RunTimes;

with Fortran_Runtimes;
package Fortran_Library is
  pragma FOREIGN_BODY ("fortran", "init_seed");
  function SquareRoot(x:float) return float;
  function Exp  (x:float) return float;
  function Random return float;
private
  pragma LINKAGE_NAME (SquareRoot, "Sqrt");
  pragma LINKAGE_NAME (Exp, "Exp");
  pragma LINKAGE_NAME (Random, "Rnd");
end Fortran_Library;
```

The user may compile a body written in Ada for a specification into the library, regardless of the language specified in the pragma contained in the specification. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a

foreign body with an Ada body without recompiling the specification.

The user can either compile an Ada body into the library, or use the command ALIB FOREIGN_BODY (See Section 4.7) to use an Ada body from another library. The Ada body from another library must have been compiled under an identical specification. The pragma LINKAGE_NAME must have been applied to all entities declared in the specification. The only way to specify the linkname for the elaboration routine of an Ada body is with the pragma FOREIGN_BODY.

5.2. IMPLEMENTATION-DEPENDENT ATTRIBUTES

No implementation-dependent attributes are currently supported.

5.3. SPECIFICATION OF THE PACKAGE SYSTEM

The parameter values specified for the VAX in package SYSTEM [LRM 13.7.1 and Appendix C] are:

```
package system is
   type address is new integer;
   type name is (VAX, MIL_STD_1750A, MC68000);
   system_name  : constant name := VAX;
   storage_unit : constant := 8;
   memory_size  : constant := 1000000;
   max_int      : constant := 2147483647;
   min_int      : constant := -max_int - 1;
   max_digits   : constant := 9;
   max_mantissa : constant := 31;
   fine_delta   : constant := 2#1.0#e-31;
   tick         : constant := 0.01667;
   subtype priority is integer range 10 .. 200;
   default_priority: constant priority := priority'first;
   runtime_error    : exception;
end system;
```

5.4. RESTRICTIONS ON REPRESENTATION CLAUSES

Restrictions on representation specifications:

- Length clauses [LRM 13.2]:

  * A length clause for T'SIZE is permitted for any type or first subtype T for which the size can be computed at compile time. A length clause for a composite type cannot be used to force a smaller size for components than established by the default type mapping or by length clauses for the component types.

  * There are no restrictions on other forms of length clauses except the restrictions specified in LRM 13.2. The size specified for

T'STORAGE_SIZE of an access type or task type T is assumed to include a small amount of hidden administrative storage.

- Enumeration representation clauses [LRM 13.3]:

  * All integer codes in the representation aggregate must be between INTEGER'FIRST and INTEGER'LAST.

- Record representation clauses [LRM 13.4]:

  * Record representation clauses are permitted only for record types all of whose components have a size known at compile time.

  * Representation specifications may be specified for some components of a record without supplying representation specifications for all components. The compiler will freely allocate the unspecified components.

- Address clauses [LRM 13.5]:

  * When applied to an object, an address clause becomes a linker directive to allocate the object at the given logical address. For any object not declared immediately within a top-level library package, the address clause is meaningless, with the possible exception of objects declared inside a task, if the target permits a task to run in a separate address space.

  * Address clauses applied to local packages are not supported by Tartan Ada.

  * Address clauses applied to subprograms and tasks are implemented according to the LRM rules.

  * When applied to an entry, the specified value identifies an interrupt in a manner customary for the target. Immediately after a task is created, a runtime call is made for each of its entries having an address clause, establishing the proper binding between the entry and the interrupt.

5.5. IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record. These components cannot be named by the user.

5.6. INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES

Section 13.5.1 of the Ada Language Reference Manual describes a syntax for associating interrupts with task entries. Tartan Ada implements the address clause

```
for toentry use at intID;
```

by associating the interrupt specified by intID with the toentry entry of the task containing this address clause. The interpretation of intID is both machine and compiler dependent.

5.7. RESTRICTIONS ON UNCHECKED CONVERSIONS

Tartan supports UNCHECKED_CONVERSION with a restriction that requires the sizes of both source and target types to be known at compile time. The sizes need not be the same. If the value in the source is wider than that in the target, the source value will be truncated. If narrower, it will be zero-extended. Calls on instantiations of UNCHECKED_CONVERSION are made inline automatically.

5.8. IMPLEMENTATION-DEPENDENT CHARACTERISTICS OF INPUT-OUTPUT PACKAGES

Tartan Ada supports all predefined input/output packages [LRM Chapter 14] with the exception of LOW_LEVEL_IO (which is planned for a future release).

SEQUENTIAL_IO and DIRECT_IO may not be instantiated on types whose representation size is greater than 32255 bytes. Any attempt to read or write values of such types raises USE_ERROR.

SEQUENTIAL_IO and DIRECT_IO may not be instantiated on unconstrained array types, nor on record record types with discriminants without default values.

An attempt to delete an external file while more than one internal file refers to this external file raises USE_ERROR.

When an external file is referenced by more than one internal file, an attempt to reset one of those internal files to OUT_FILE raises USE_ERROR.

An attempt to create a file with FILE_MODE IN_FILE raises USE_ERROR.

Since the implementation of the input-output packages uses buffers, output to one file cannot necessarily be read immediately from another file associated with the same external file.

The FORM parameter of file management subprograms is ignored.

An attempt to read a non-existent data record through the operations of SEQUENTIAL_IO or DIRECT_IO raises DATA_ERROR, except that END_ERROR is raised when reading beyond the end of file.

If the VMS record management services (RMS) return a status value that cannot be mapped onto a predefined Ada exception, the exception DEVICE_ERROR is raised.

5.9. OTHER IMPLEMENTATION CHARACTERISTICS

The following information is supplied in addition to that required by Appendix F to MIL-STD-1815A.

5.9.1. Definition of a Main Program

Any Ada library subprogram unit may be designated the main program for purposes of linking (using the ALIB command) provided that the subprogram has no parameters.

Tasks initiated in imported library units follow the same rules for termination as other tasks [described in LRM 9.4 (6-10)]. Specifically, these tasks are not terminated simply because the main program has terminated. Terminate alternatives in selective wait statements in library tasks are therefore strongly recommended.

5.9.2. Implementation of Generic Units

All instantiations of generic units, except the predefined generic UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION subprograms, are implemented by code duplications. No attempt at sharing code by multiple instantiations is made in this release of Tartan Ada. (Code sharing will be implemented in a later release.)

Tartan Ada enforces the restriction that the body of a generic unit must be compiled before the unit can be instantiated. It does not impose the restriction that the specification and body of a generic unit must be provided as part of the same compilation. A recompilation of the body of a generic unit will obsolete any units that instantiated this generic unit.

5.9.3. Attributes of Type Duration

The type DURATION is defined with the following characteristics:

```
DURATION'DELTA is 0.02 sec
DURATION'SMALL is  0.015625 sec
DURATION'FIRST is -86400.0 sec
DURATION'LAST  is 86400.0 sec
```

5.9.4. Values of Integer Attributes

Tartan Ada supports the predefined integer type INTEGER. The range bounds of the predefined type INTEGER are:

```
INTEGER'FIRST is -2**31
INTEGER'LAST is 2**31-1
```

The range bounds for subtypes declared in package TEXT_IO are:

```
COUNT'FIRST is 0
COUNT'LAST is INTEGER'LAST - 1

POSITIVE_COUNT'FIRST is 1
POSITIVE_COUNT'LAST is INTEGER'LAST - 1

FIELD'FIRST is 0
FIELD'LAST is 20
```

The range bounds for subtypes declared in packages DIRECT_IO are:

```
COUNT'FIRST is 0
COUNT'LAST is INTEGER'LAST

POSITIVE_COUNT'FIRST is 1
POSITIVE_COUNT'LAST is COUNT'LAST
```

5.9.5. Values of Floating-Point Attributes .

Tartan Ada supports the predefined floating-point types FLOAT and LONG_FLOAT. FLOAT maps onto the VAX F-format floating-point representation; LONG_FLOAT, onto the D-format. Future versions of Tartan Ada will support all four VAX formats (F-, D-, G-, and H-format) in an implementation-dependent library package.

Attribute	Value for FLOAT
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	16#0.1000_000#E-4
approximately	9.53673E-07
SMALL	16#0.8000_000#E-21
approximately	2.58494E-26
LARGE	16#0.FFFF_F80#E+21
approximately	1.93428E+25
SAFE_EMAX	127
SAFE_SMALL	16#0.1000_000#E-31
approximately	2.93874E-39
SAFE_LARGE	16#0.7FFF_FC0#E+32
approximately	1.70141E+38

FIRST	-16#0.7FFF_FF8#E+32
approximately	-1.70141E+38
LAST	16#0.7FFF_FF8#E+32
approximately	1.70141E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	127
MACHINE_EMIN	-127
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

Attribute	Value for LONG_FLOAT
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON	16#0.4000_0000_0000_000#E-7
approximately	9.3132257461548E-10
SMALL	16#0.8000_0000_0000_000#E-31
approximately	2.3509887016446E-38
LARGE	16#0.FFFF_FFFE_0000_000#E+31
approximately	2.1267647922655E+37
SAFE_EMAX	127
SAFE_SMALL	16#0.1000_0000_0000_000#E-31
approximately	2.9387358770557E-39
SAFE_LARGE	16#0.7FFF_FFFF_0000_000#E+32
approximately	1.7014118338124E+38
FIRST	-16#0.7FFF_FFFF_FFFF_FF8#E+32
approximately	-1.7014118346047E+38
LAST	16#0.7FFF_FFFF_FFFF_FF8#E+32
approximately	1.7014118346047E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	56
MACHINE_EMAX	127
MACHINE_EMIN	-127
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

APPENDIX C

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. Ada aggregate notation is used to denote long strings. The multiplication operator is meant to be overloaded to achieve repetition so that e.g. 239 * 'A' is equivalent to (1..239 => 'A').

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	239 * 'A' & '1'
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	239 * 'A' & '2'
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	120 * 'A' & '3' & 119 * 'A'
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	120 * 'A' & '4' & 119 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	237 * '0' & "298"

TEST PARAMETERS

Name and Meaning	Value
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	235 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 120 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 119 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	220 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483646
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	20
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	X}J!@#$^&~Y
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	XYZ*
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100000.0

Name and Meaning	Value
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	100000000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BAD-CHAR*^
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	MUCH_TOO_LONG.. (truncated)
$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
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-100000000.0
$MAX_DIGITS Maximum digits supported for floating-point types.	9
$MAX_IN_LEN Maximum input line length permitted by the implementation.	240
$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

TEST PARAMETERS

Name and Meaning	Value
$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:" & 235*'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:" & 233*'0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 238 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	$NAME
$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.	8#7777777777776#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 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.

- B28003A: A basic declaration (line 36) wrongly follows a later declaration.

- E28005C: This test requires that 'PRAGMA LIST (ON);' not appear in a listing that has been suspended by a previous "pragma LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the ARG.

- C34004A: The expression in line 168 wrongly yields a value outside of the range of the target type T, raising CONSTRAINT_ERROR.

- C35502P: Equality operators in lines 62 & 69 should be inequality operators.

- A35902C: Line 17's assignment of the nomimal upper bound of a fixed-point type to an object of that type raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

- C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

- C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may in fact raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

- C35A03E, C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard doesn't support this assumption.

- C37213H: The subtype declaration of SCONS in line 100 is wrongly expected to raise an exception when elaborated.

- C37213J: The aggregate in line 451 wrongly raises CONSTRAINT_ERROR.

- C37215C, C37215E, C37215G, C37215H: Various discriminant constraints are wrongly expected to be incompatible with type CONS.

- C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

- C41402A: 'STORAGE_SIZE is wrongly applied to an object of an access type.

- C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

- C45614C: REPORT.IDENT_INT has an argument of the wrong type (LONG_INTEGER).

- A74106C, C85018B, C87B04B, CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur in lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively (and possibly elsewhere).

- BC3105A: Lines 159..168 are wrongly expected to be illegal; they are legal.

- AD1A01A: The declaration of subtype INT3 raises CONSTRAINT_ERROR for implementations that select INT'SIZE to be 16 or greater.

- CE2401H: The record aggregates in lines 105 & 117 contain the wrong values.

- CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.