

AD-A174 959

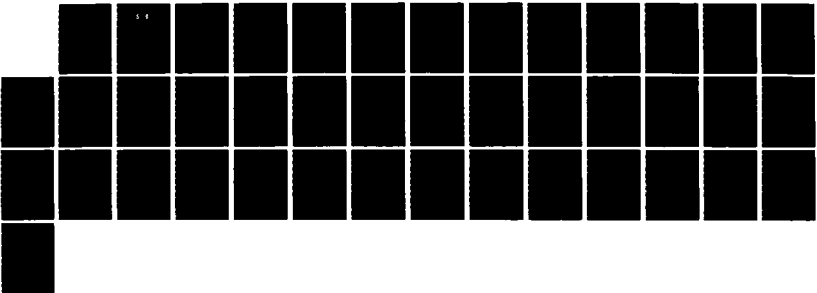
ADA (TRADE NAME) COMPILER VALIDATION SUMMARY REPORT:
VERDIX ADA COMPILER. (U) INFORMATION SYSTEMS AND
TECHNOLOGY CENTER W-P AFB OH ADA VALI.. 02 APR 86
AVF-VSR-23. 0486

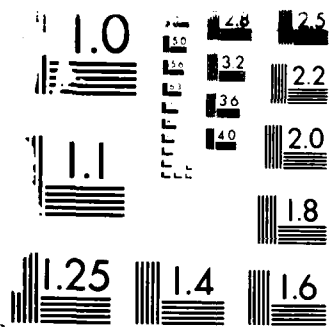
1/1

UNCLASSIFIED

F/G 9/2

NL





RESOLUTION TEST CHART

2

AVF-VSR-23.0486

AD-A174 959

DTIC
ELECTE
DEC 12 1986
S D
D

Ada® COMPILER
VALIDATION SUMMARY REPORT:
Verdix
Verdix Ada Compiler System (VAda-010-0101)
Version 6.0
VAX-11/750

Completion of On-Site Validation:
2 APR 1986

Prepared By:
Ada Validation Facility
ASD/SIOL
Wright-Patterson AFB OH 45433-6503

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

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

Ada is a registered trademark of the United States Government
(Ada Joint Program Office).

CLEARED
FOR OPEN PUBLICATION

NOV 20 1986 3

86 4975

DIRECTORATE FOR FREEDOM OF INFORMATION
AND SECURITY REVIEW (CASD-PA)
DEPARTMENT OF DEFENSE

86 12 11 075

DTIC FILE COPY

7-4

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: VERDIX ADA COMPILER SYSTEM (VALIDATION 010-0101) VERSION 6.0		5. TYPE OF REPORT & PERIOD COVERED April 1986 to April 1987
7. AUTHOR(s) Ada Validation Facility ASD/SIOL Wright Patterson AFB OH 45433-6503		6. PERFORMING ORG. REPORT NUMBER AVF_VSR-23.0486
9. PERFORMING ORGANIZATION AND ADDRESS Ada Validation Facility		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Programming Office United States Department of Defense Washington, D.C. 20301-3081		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Ada Valication Facility ASC/SIOL Wright-Patterson AFB OH 45433-6503		12. REPORT DATE 2 April 1986
		13. NUMBER OF PAGES 42
		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) See Attached.		

DD FORM 1473
1 JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Verdix Ada Compiler System (VAda-010-0101), Version 6.0, using Version 1.7 of the Ada[®] Compiler Validation Capability (ACVC).

The validation process includes submitting a suite of standardized tests (the ACVC) as inputs to an Ada compiler and evaluating the results. The purpose is to ensure conformance of the compiler to ANSI/MIL-STD-1815A Ada by testing that it 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 ANSI/MIL-STD-1815A. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, or during execution.

On-site testing was performed 30 MAR 1986 through 2 APR 1986 at Portland OR under the auspices of the Ada Validation Facility (AVF), according to Ada Validation Organization policies and procedures. The Verdix Ada Compiler System (VAda-010-0101), Version 6.0, is hosted on VAX-11/750 operating under Berkeley UNIX 4.2 BSD.

The results of validation are summarized in the following table:

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	68	820	1073	17	11	23	2012
Failed	0	0	0	0	0	0	0
Inapplicable	0	4	248	0	0	0	252
Withdrawn	0	4	11	0	0	0	15
TOTAL	68	828	1332	17	11	23	2279

[®]Ada is a registered trademark of the United States Government (Ada Joint Program Office).

Ada[®] Compiler Validation Summary Report:

Compiler Name: Verdix Ada Compiler System (VAda-010-0101), Version 6.0

Host Computer:
VAX-11/750
under
Berkeley UNIX 4.2 BSD

Target Computer:
VAX-11/750
under
Berkeley UNIX 4.2 BSD

Testing Completed 2 APR 1986 Using ACVC 1.7

This report has been reviewed and is approved.

Georganne Chitwood

Ada Validation Facility
Georganne Chitwood
ASD/SIOL
Wright-Patterson AFB OH 45433-6503



John F. Kramer

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

Virginia L. Castor

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

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Verdix Ada Compiler System (VAda-010-0101), Version 6.0, using Version 1.7 of the Ada[®] Compiler Validation Capability (ACVC).

The validation process includes submitting a suite of standardized tests (the ACVC) as inputs to an Ada compiler and evaluating the results. The purpose is to ensure conformance of the compiler to ANSI/MIL-STD-1815A Ada by testing that it 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 ANSI/MIL-STD-1815A. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, or during execution.

On-site testing was performed 30 MAR 1986 through 2 APR 1986 at Portland OR under the auspices of the Ada Validation Facility (AVF), according to Ada Validation Organization policies and procedures. The Verdix Ada Compiler System (VAda-010-0101), Version 6.0, is hosted on VAX-11/750 operating under Berkeley UNIX 4.2 BSD.

The results of validation are summarized in the following table:

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	68	820	1073	17	11	23	2012
Failed	0	0	0	0	0	0	0
Inapplicable	0	4	248	0	0	0	252
Withdrawn	0	4	11	0	0	0	15
TOTAL	68	828	1332	17	11	23	2279

[®]Ada is a registered trademark of the United States Government (Ada Joint Program Office).

There were 15 withdrawn tests in ACVC Version 1.7 at the time of this validation attempt. A list of these tests appears in Appendix D.

Some tests demonstrate that some language features are or are not supported by an implementation. For this implementation, the tests determined the following:

- . LONG_INTEGER and LONG_FLOAT are not supported.
- . The additional predefined types SHORT_INTEGER, SHORT_FLOAT, and TINY_INTEGER are supported.
- . Representation specifications for noncontiguous enumeration representations are supported.
- . Generic unit specifications and bodies can be compiled in separate compilations.
- . Pragma INLINE is supported for procedures and functions.
- . The package SYSTEM is used by package TEXT_IO.
- . Modes IN_FILE and OUT_FILE are supported for sequential I/O.
- . Instantiation of the package SEQUENTIAL_IO with unconstrained array types, and with unconstrained record types with discriminants, is supported.
- . RESET and DELETE are supported for sequential and direct I/O.
- . Modes IN_FILE, INOUT_FILE, and OUT_FILE are supported for direct I/O.
- . Instantiation of package DIRECT_IO with unconstrained array types, and unconstrained types with discriminants, is supported.
- . Dynamic creation and deletion of files are supported.
- . More than one internal file can be associated with the same external file.
- . An external file associated with more than one internal file can be reset.
- . Illegal file names can exist.

ACVC Version 1.7 was taken on-site via magnetic tape to Portland OR. All tests, except the withdrawn tests and any executable tests that make use of a floating-point precision greater than SYSTEM.MAX_DIGITS, were compiled on a VAX-11/750. Class A, C, D, and E tests were executed on a VAX-11/750.

On completion of testing, execution results for Class A, C, D, or E tests were examined. Compilation results for Class B were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors.

The AVF identified 2022 of the 2279 tests in Version 1.7 of the ACVC as potentially applicable to the validation of Verdex Ada Compiler System (VAda-010-0101), Version 6.0. Excluded were 242 tests requiring a floating-point precision greater than that supported by the implementation and the 15 withdrawn tests. After the 2022 tests were processed, 10 tests were determined to be inapplicable. The remaining tests were passed by the compiler.

The AVF concludes that these results demonstrate acceptable conformance to ANSI/MIL-STD-1815A.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2	USE OF THIS VALIDATION SUMMARY REPORT	1-2
1.3	RELATED DOCUMENTS	1-3
1.4	DEFINITION OF TERMS	1-3
1.5	ACVC TEST CLASSES	1-4
CHAPTER 2	CONFIGURATION INFORMATION	
2.1	CONFIGURATION TESTED	2-1
2.2	CERTIFICATE INFORMATION	2-2
2.3	IMPLEMENTATION CHARACTERISTICS	2-2
CHAPTER 3	TEST INFORMATION	
3.1	TEST RESULTS	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER	3-1
3.4	WITHDRAWN TESTS	3-2
3.5	INAPPLICABLE TESTS	3-2
3.6	SPLIT TESTS	3-3
3.7	ADDITIONAL TESTING INFORMATION	3-3
3.7.1	Prevalidation	3-3
3.7.2	Test Method	3-4
3.7.3	Test Site	3-4
APPENDIX A	COMPLIANCE STATEMENT	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to 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 (ANSI/MIL-STD-1815A). Any implementation-dependent features must conform to the requirements of the Ada Standard. The entire Ada Standard must be implemented, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to ANSI/MIL-STD-1815A, 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 limitations imposed on a compiler by the operating system and by the hardware. All of the dependencies demonstrated during the process of testing this compiler are given in this report.

VSRs are written according to a standardized format. The reports for several different compilers may, therefore, be easily compared. The information in this report is derived from the test results produced during validation testing. Additional testing information is given in section 3.7 and states problems and details which are unique for a specific compiler. The format of a validation report limits variance between reports, enhances readability of the report, and minimizes the delay between the completion of validation testing and the publication of the report.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

The 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

INTRODUCTION

- . To attempt to identify any unsupported language constructs 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 SofTech Inc., under the supervision of the Ada Validation Facility (AVF) according to policies and procedures established by the Ada Validation Organization (AVO). Testing was conducted from 30 MAR 1986 through 2 APR 1986 at Portland OR.

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 nonconformances to ANSI/MIL-STD-1815A 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
1211 S. Fern, C-107
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SIOL
Wright-Patterson AFB OH 45433-6503

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
 Alexandria VA 22311

1.3 RELATED DOCUMENTS

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, FEB 1983.
2. Ada Validation Organization: Policies and Procedures, MITRE Corporation, JUN 1982, PB 83-110601.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., DEC 1984.

1.4 DEFINITION OF TERMS

ACVC The Ada Compiler Validation Capability. A set of programs that evaluates the conformance of a compiler to the Ada language specification, ANSI/MIL-STD-1815A.

Ada Standard ANSI/MIL-STD-1815A, February 1983.

Applicant The agency requesting validation.

AVF The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established policies and procedures.

AVO The Ada Validation Organization. In the context of this report, the AVO is responsible for setting policies and procedures for compiler validations.

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 A test for which the compiler generates a result that demonstrates nonconformance to the Ada Standard.

Host The computer on which the compiler resides.

INTRODUCTION

Inapplicable test	A 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.
LMC	The Language Maintenance Committee, whose function is to resolve issues concerning the Ada language.
Passed test	A test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that evaluates the conformance of a compiler to a language specification. In the context of this report, the term is used to designate a single ACVC test. The text of a program may be the text of one or more compilations.
Withdrawn test	A test found to be inaccurate in checking conformance to the Ada language specification. A withdrawn test 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

Conformance to ANSI/MIL-STD-1815A is measured using the Ada Compiler Validation Capability (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. Special program units are used to report the results of the Class A, C, D, and E tests during execution. Class B tests are expected to produce compilation errors, and Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. (However, no checks are performed during execution to see if the test objective has been met.) 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 message indicating that it has passed.

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 syntactical 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.

INTRODUCTION

Class D tests check the compilation and execution capacities of a compiler. Since there are no requirements placed on a compiler by the Ada Standard for some parameters (e.g., the number of identifiers permitted in a compilation, the number of units in a library, and the number of nested loops in a subprogram body), 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 results. It also provides a set of identity functions used to defeat some compiler optimization strategies and force computations to be made by the target computer instead of by the compiler on the host computer. 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 these units 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.

Some of the conventions followed in the ACVC 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. The values used for this validation are listed in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformance 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. Any test that was determined to contain an illegal

INTRODUCTION

language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The nonconformant tests 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: Verdex Ada Compiler System (VAda-010-0101), Version 6.0

Test Suite: Ada Compiler Validation Capability, Version 1.7

Host Computer:

Machine(s):	VAX-11/750
Operating System:	Berkeley UNIX 4.2 BSD
Memory Size:	6 megabytes

Target Computer:

Machine(s):	VAX-11/750
Operating System:	Berkeley UNIX 4.2 BSD
Memory Size:	6 megabytes

CONFIGURATION INFORMATION

2.2 CERTIFICATE INFORMATION

Compiler: Verdex Ada Compiler System (VAda-010-0101), Version 6.0

Test Suite: Ada Compiler Validation Capability, Version 1.7

Host Computer:

Machine(s):	VAX-11/750
Operating System:	Berkeley UNIX 4.2 BSD
Memory Size:	6 megabytes

Target Computer:

Machine(s):	VAX-11/750
Operating System:	Berkeley UNIX 4.2 BSD
Memory Size:	6 megabytes

2.3 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. This compiler is characterized by the following interpretations of the Ada Standard:

- Nongraphic characters.

Nongraphic characters are defined in the ASCII character set but are not permitted in Ada programs, even within character strings. The compiler correctly recognizes these characters as illegal in Ada compilations. The characters are printed in the output listing. (See test B26005A.)

- Capacities.

The compiler correctly processes compilations containing loop statements nested to 65 levels, block statements nested to 65 levels, recursive procedures nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A through D55A03H, D56001B,

CONFIGURATION INFORMATION

D64005E through D64005G, and D29002K.)

- Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed `SYSTEM.MAX_INT`. This implementation does not reject such calculations and processes them correctly. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

This implementation supports the additional predefined types `SHORT_INTEGER`, `SHORT_FLOAT`, and `TINY_INTEGER` in the package `STANDARD`. (See tests B86001CR, B86001CP, and B86001DT.)

- 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` during execution. This implementation raises `NUMERIC_ERROR` during execution. (See test E24101A.)

- Array types.

When an array type is declared with an index range exceeding the `INTEGER'LAST` values and with a component that is a null `BOOLEAN` array, this compiler raises `NUMERIC_ERROR` when the type is declared (See tests E36202A and E36202B.)

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` either when declared or assigned. Alternately, 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 entire expression appears to be evaluated 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 entire expression does not appear to be evaluated before `CONSTRAINT_ERROR` is raised when checking whether the expression's subtype is compatible with

CONFIGURATION INFORMATION

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 during compilation. (See test E38104A.)

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

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.)

- . Functions.

The declaration of a parameterless function with the same profile as an enumeration literal in the same immediate scope is rejected by the implementation. (See test E66001D.)

- . Representation clauses.

Enumeration representation clauses are supported. (See test BC1002A.)

- . Tasks.

The storage size of a task object is allowed to change after the task is activated. (See test C92005A.)

CONFIGURATION INFORMATION

- . Generics.

When given a separately compiled generic unit specification, some illegal instantiations, and a body, the compiler rejects the body because of the instantiations. (See test BC3204D.)

- . Pragas.

The pragma `INLINE` is supported for procedures and functions. (See tests CA3004E and CA3004F.)

- . Input/output.

The package `SEQUENTIAL_IO` can be instantiated with unconstrained array types and record types with discriminants. The package `DIRECT_IO` can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests CE2201D, CE2201E, and CE2401D.)

More than one internal file can be associated with each external file for sequential I/O for both reading and writing. (See tests CE2107A through CE2107F.)

More than one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107A through CE2107F.)

An external file associated with more than one internal file can be deleted. (See test CE2110B.)

More than one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE3111A through CE3111E.)

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

Temporary sequential files and temporary direct files are given a name. Temporary files given names are deleted when they are closed. (See test CE2108A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

The AVF identified 2022 of the 2279 tests in Version 1.7 of the ACVC as potentially applicable to the validation of Verdix Ada Compiler System (VAda-010-0101), Version 6.0. Excluded were 242 tests requiring a floating-point precision greater than that supported by the implementation and the 15 withdrawn tests. After they were processed, 10 tests were determined to be inapplicable. The remaining tests were passed by the compiler.

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

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	68	820	1073	17	11	23	2012
Failed	0	0	0	0	0	0	0
Inapplicable	0	4	248	0	0	0	252
Withdrawn	0	4	11	0	0	0	15
TOTAL	68	828	1332	17	11	23	2279

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

TEST INFORMATION

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	96	204	272	244	161	97	158	199	105	28	216	232	2012
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	20	103	122	3	0	0	3	1	0	0	0	0	252
Withdrawn	0	1	4	0	0	0	1	1	6	0	1	1	15
TOTAL	116	308	398	247	161	97	162	201	111	28	217	233	2279

3.4 WITHDRAWN TESTS

The following tests have been withdrawn from the ACVC Version 1.7:

B4A010C	C35904A	C92005A
B83A06B	C41404A	CA1003B
BA2001E	C48008A	CA3005A through CA3005D (4 tests)
BC3204C	C4A014A	CE2107E

See Appendix D for the test descriptions.

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. For this validation attempt, 252 tests were inapplicable for the reasons indicated:

- . C34001G, C35702B, and B86001CQ use LONG_FLOAT which is not supported by this compiler.
- . C34001E, B52004D, C55B07A, B55B09C, and B86001CS use LONG_INTEGER which is not supported by this compiler.
- . C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation.
- . C96005B uses DURATION'BASE values above maximum and below minimum which are not supported by this implementation.

TEST INFORMATION

- 242 tests were not processed because SYSTEM.MAX_DIGITS was 9.
These tests were:

C24113F through C24113Y (20 tests)
C35705F through C35705Y (20 tests)
C35706F through C35706Y (20 tests)
C35707F through C35707Y (20 tests)
C35708F through C35708Y (20 tests)
C35802F through C35802Y (20 tests)
C45241F through C45241Y (20 tests)
C45321F through C45321Y (20 tests)
C45421F through C45421Y (20 tests)
C45424F through C45424Y (20 tests)
C45521F through C45521Z (21 tests)
C45621F through C45621Z (21 tests)

3.6 SPLIT TESTS

If one or more errors do not appear to have been detected in a Class B test because of compiler error recovery, then the test is split into a set of smaller tests that contain the undetected errors. These splits are then compiled and examined. The splitting process continues until all errors are detected by the compiler or until there is exactly one error per split. Any Class A, Class C, or Class E test that cannot be compiled and executed because of its size is split into a set of smaller subtests that can be processed.

Splits were required for 18 Class B tests.

B24104A	B33004A	B67001B
B24104B	B37201A	B67001C
B24104C	B38008A	B67001D
B2A003A	B44001A	B910ABA
B2A003B	B64001A	B95001A
B2A003C	B67001A	B97101E

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.7 produced by Verdix Ada Compiler System (VAda-010-0101), Version 6.0, was submitted to the AVF by the applicant for pre-validation review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests.

TEST INFORMATION

3.7.2 Test Method

Testing of Verdix Ada Compiler System (VAda-010-0101) using ACVC Version 1.7 was conducted on-site by a validation team. The configuration consisted of a VAX-11/750 host and target operating under Berkeley UNIX 4.2 BSD.

A magnetic tape containing ACVC Version 1.7 was taken on-site by the validation team. The magnetic tape contained all tests applicable to this validation, as well as all tests inapplicable to this validation except for any Class C tests that require floating-point precision exceeding the maximum value supported by the implementation. Tests that make use of values that are specific to an implementation were customized before being written to the magnetic tape. Tests requiring splits during the prevalidation testing were included in their split form on the test magnetic tape. No editing of the test files was necessary when the validation team arrived on-site.

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests were compiled on the VAX-11/750, and all executable tests were run on the VAX-11/750. Results were printed from the VAX-11/750. Tests that were withdrawn from ACVC Version 1.7 were not run.

The compiler was tested using command scripts provided by Verdix. These scripts were reviewed by the validation team.

Tests were run in batch mode using a single host and target 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

The validation team arrived at Portland OR on 30 MAR 1986 and departed after testing was completed on 2 APR 1986.

APPENDIX A
COMPLIANCE STATEMENT

Verdix has submitted the following compliance statement concerning the Verdix Ada Compiler System (VAda-010-0101).

COMPLIANCE STATEMENT

Compliance Statement

Base Configuration:

Compiler: Verdex Ada Compiler System (VAda-010-0101), Version 6.0

Test Suite: Ada Compiler Validation Capability, Version 1.7

Host Computer:

Machine(s): VAX-11/750

Operating System: Berkeley UNIX 4.2 BSD

Target Computer:

Machine(s): VAX-11/750

Operating System: Berkeley UNIX 4.2 BSD

Verdex has made no deliberate extensions to the Ada language standard.

Verdex agrees to the public disclosure of this report.

Verdex agrees to comply with the Ada trademark policy, as defined by the Ada Joint Program Office.

Dave Nomura Date: 6/30/86

Verdex
Dave Nomura
Manager
Ada Products Division

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 MIL-STD-1815A, and to certain allowed restrictions on representation classes. The implementation-dependent characteristics of the Verdex Ada Compiler System (VAda-010-0101), Version 6.0, are described in the following sections, which discuss topics one through eight as stated in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A). Package STANDARD is also included in this appendix.

1. Implementation-Dependent Pragmas

1.1. SHARE_BODY Pragma

The SHARE_BODY pragma takes the name of a generic instantiation or a generic unit as the first argument and one of the identifiers TRUE or FALSE as the second argument. This pragma is only allowed immediately at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

When the first argument is a generic unit the pragma applies to all instantiations of that generic. When the first argument is the name of a generic instantiation the pragma applies only to the specified instantiation, or overloaded instantiations.

If the second argument is TRUE the compiler will try to share code generated for a generic instantiation with code generated for other instantiations of the same generic. When the second argument is FALSE each instantiation will get a unique copy of the generated code. The extent to which code is shared between instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit.

1.2. EXTERNAL_NAME Pragma

The EXTERNAL_NAME pragma takes the name of a subprogram or variable defined in Ada and allows the user to specify a different external name that may be used to reference the entity from other languages. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification.

1.3. INTERFACE_OBJECT Pragma

The `INTERFACE_OBJECT` pragma takes the name of a variable defined in another language and allows it to be referenced directly in Ada. The pragma will replace all occurrences of the variable name with an external reference to the second, `link_argument`. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar or an access type. The object *cannot* be any of the following:

- a loop variable,
- a constant,
- an initialized variable,
- an array, or
- a record.

2. Implementation of Predefined Pragmas

2.1. CONTROLLED

This pragma is recognized by the implementation but has no effect.

2.2. ELABORATE

This pragma is implemented as described in Appendix B of the Ada RM.

2.3. INLINE

This pragma is implemented as described in Appendix B of the Ada RM.

2.4. INTERFACE

This pragma supports calls to 'C' and FORTRAN functions. The Ada subprograms can be either functions or procedures. The types of parameters and the result type for functions must be scalar, access or the predefined type `ADDRESS` in `SYSTEM`. An optional third argument overrides the default link name. All parameters must have mode `IN`. Record and array objects can be passed by reference using the `ADDRESS` attribute.

2.5. LIST

This pragma is implemented as described in Appendix B of the Ada RM.

2.6. MEMORY_SIZE

This pragma is recognized by the implementation. The implementation does not allow `SYSTEM` to be modified by means of pragmas, the `SYSTEM` package must be recompiled.

2.7. OPTIMIZE

This pragma is recognized by the implementation but has no effect.

2.8. PACK

This pragma will cause the compiler to choose a non-aligned representation for composite types. Components that are smaller than a `STORAGE_UNIT` are packed into a number of bits that is a power of two level.

2.9. PAGE

This pragma is implemented as described in Appendix B of the Ada RM.

2.10. PRIORITY

This pragma is implemented as described in Appendix B of the Ada RM.

(3) Package SYSTEM

2.11. SHARED

This pragma is recognized by the implementation but has no effect.

2.12. STORAGE_UNIT

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

2.13. SUPPRESS

This pragma is implemented as described, except that RANGE_CHECK and DIVISION_CHECK cannot be suppressed.

2.14. SYSTEM_NAME

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

3. Implementation-Dependent Attributes

NONE.

4. Specification Of Package SYSTEM

```

package SYSTEM
is
    type NAME is ( vax_unix );

    SYSTEM_NAME      : constant NAME := vax_unix;

    STORAGE_UNIT    : constant := 8;
    MEMORY_SIZE     : constant := 6_291_456;

    -- System-Dependent Named Numbers

    MIN_INT          : constant := -2_147_483_647 - 1;
    MAX_INT          : constant := 2_147_483_647;
    MAX_DIGITS       : constant := 9;
    MAX_MANTISSA     : constant := 31;
    FINE_DELTA       : constant := 2.0*(-14);
    TICK             : constant := 0.01;

    -- Other System-dependent Declarations

    subtype PRIORITY is INTEGER range 0 .. 7;

    MAX_REC_SIZE : integer := 64*1024;

    type ADDRESS is private;
    no_addr: constant address;
    function incr_addr(a: address; incr: integer) return address;
    function decr_addr(a: address; decr: integer) return address;
    function addr_diff(a, b: address) return integer;
function "+"(a: address; incr: integer) return address renames incr_addr;
function "-"(a: address; decr: integer) return address renames decr_addr;
function "-"(a, b: address) return integer renames addr_diff;
pragma inline(incr_addr, decr_addr, addr_diff);

```

APPENDIX F OF THE Ada STANDARD

`private`

```
type ADDRESS is new INTEGER;  
no_addr: constant address := address(0);
```

`end SYSTEM;`

5. Restrictions On Representation Clauses

5.1. Pragma PACK

Array and record components that are smaller than a `STORAGE_UNIT` are packed into a number of bits that is a power of two. Objects and larger components are packed to the nearest whole `STORAGE_UNIT`.

5.2. Size Specification

The size specification `TSMALL` is not supported.

5.3. Record Representation Clauses

Components not aligned on `STORAGE_UNIT` boundaries may not span more than four `STORAGE_UNITS`.

5.4. Address Clauses

Address clauses are not supported.

5.5. Interrupts

Interrupts are not supported.

5.6. Representation Attributes

The `ADDRESS` attribute is not supported for the following entities:

- Packages
- Tasks
- Labels
- Entries

5.7. Machine Code Insertions

Machine code insertions are not supported.

6. Conventions for Implementation-generated Names

There are no implementation-generated names.

7. Interpretation of Expressions in Address Clauses

Address clauses are not supported.

8. Restrictions on Unchecked Conversions

The predefined generic function `UNCHECKED_CONVERSION` cannot be instantiated with a target type which is an unconstrained array type or an unconstrained record type with discriminants.

9. Implementation Characteristics of I/O Packages

Instantiations of `DIRECT_IO` use the value `MAX_REC_SIZE` as the record size (expressed in `STORAGE_UNITS`) when the size of `ELEMENT_TYPE` exceeds that value. For example for unconstrained arrays such as string where `ELEMENT_TYPE'SIZE` is very large, `MAX_REC_SIZE` is used instead. `MAX_RECORD_SIZE` is defined in `SYSTEM` and can be changed by a program before instantiating `DIRECT_IO` to provide an upper limit on the record size. In any case the maximum size supported is $1024 \times 1024 \times \text{STORAGE_UNIT}$ bits. `DIRECT_IO` will raise `USE_ERROR` if `MAX_REC_SIZE` exceeds this absolute limit.

Instantiations of `SEQUENTIAL_IO` use the value `MAX_REC_SIZE` as the record size (expressed in `STORAGE_UNITS`) when the size of `ELEMENT_TYPE` exceeds that value. For example for unconstrained arrays such as string where `ELEMENT_TYPE'SIZE` is very large, `MAX_REC_SIZE` is used instead. `MAX_RECORD_SIZE` is defined in `SYSTEM` and can be changed by a program before instantiating `INTEGER_IO` to provide an upper limit on the record size. `SEQUENTIAL_IO` imposes no limit on `MAX_REC_SIZE`.

10. Implementation Limits

The following limits are actually enforced by the implementation. It is not intended to imply that resources up to or even near these limits are available to every program.

10.1. Line Length

The implementation supports a maximum line length of 500 characters including the end of line character.

10.2. Record and Array Sizes

The maximum size of a statically sized array type is $4,000,000 \times \text{STORAGE_UNITS}$. The maximum size of a statically sized record type is $4,000,000 \times \text{STORAGE_UNITS}$. A record type or array type declaration that exceeds these limits will generate a warning message.

10.3. Default Stack Size for Tasks

In the absence of an explicit `STORAGE_SIZE` length specification every task except the main program is allocated a fixed size stack of $10,240 \text{ STORAGE_UNITS}$. This is the value returned by `T'STORAGE_SIZE` for a task type `T`.

10.4. Default Collection Size

In the absence of an explicit `STORAGE_SIZE` length attribute the default collection size for an access type is $100,000 \text{ STORAGE_UNITS}$. This is the value returned by `T'STORAGE_SIZE` for an access type `T`.

10.5. Limit on Declared Objects

There is an absolute limit of $6,000,000 \times \text{STORAGE_UNITS}$ for objects declared statically within a compilation unit. If this value is exceeded the compiler will terminate the compilation of the unit with a `FATAL` error message.

`DURATION'SMALL` = 6.103515625000000E-05 seconds

APPENDIX F OF THE Ada STANDARD

```

package standard is
  type boolean is (false, true);
  function "=" (left, right: boolean) return boolean;
  function "/=" (left, right: boolean) return boolean;
  function "<" (left, right: boolean) return boolean;
  function "<=" (left, right: boolean) return boolean;
  function ">" (left, right: boolean) return boolean;
  function ">=" (left, right: boolean) return boolean;

  function "and" (left, right: boolean) return boolean;
  function "or" (left, right: boolean) return boolean;
  function "xor" (left, right: boolean) return boolean;
  function "not" (right: boolean) return boolean;

  type tiny_integer is range -128 .. 127;
  function "=" (left, right: tiny_integer) return boolean;
  function "/=" (left, right: tiny_integer) return boolean;
  function "<" (left, right: tiny_integer) return boolean;
  function "<=" (left, right: tiny_integer) return boolean;
  function ">" (left, right: tiny_integer) return boolean;
  function ">=" (left, right: tiny_integer) return boolean;
  function "+" (right: tiny_integer) return tiny_integer;
  function "-" (right: tiny_integer) return tiny_integer;
  function "abs" (right: tiny_integer) return tiny_integer;
  function "+" (left, right: tiny_integer) return tiny_integer;
  function "-" (left, right: tiny_integer) return tiny_integer;
  function "*" (left, right: tiny_integer) return tiny_integer;
  function "/" (left, right: tiny_integer) return tiny_integer;
  function "rem" (left, right: tiny_integer) return tiny_integer;
  function "mod" (left, right: tiny_integer) return tiny_integer;
  function "***" (left, right: tiny_integer) return tiny_integer;

  type short_integer is range -32768 .. 32767;
  function "=" (left, right: short_integer) return boolean;
  function "/=" (left, right: short_integer) return boolean;
  function "<" (left, right: short_integer) return boolean;
  function "<=" (left, right: short_integer) return boolean;
  function ">" (left, right: short_integer) return boolean;
  function ">=" (left, right: short_integer) return boolean;
  function "+" (right: short_integer) return short_integer;
  function "-" (right: short_integer) return short_integer;
  function "abs" (right: short_integer) return short_integer;
  function "+" (left, right: short_integer) return short_integer;
  function "-" (left, right: short_integer) return short_integer;
  function "*" (left, right: short_integer) return short_integer;
  function "/" (left, right: short_integer) return short_integer;
  function "rem" (left, right: short_integer) return short_integer;
  function "mod" (left, right: short_integer) return short_integer;
  function "***" (left, right: short_integer) return short_integer;

  type integer is range -2147483648 .. 2147483647;
  function "=" (left, right: integer) return boolean;
  function "/=" (left, right: integer) return boolean;
  function "<" (left, right: integer) return boolean;
  function "<=" (left, right: integer) return boolean;
  function ">" (left, right: integer) return boolean;
  function ">=" (left, right: integer) return boolean;
  function "+" (right: integer) return integer;
  function "-" (right: integer) return integer;
  function "abs" (right: integer) return integer;
  function "+" (left, right: integer) return integer;
  function "-" (left, right: integer) return integer;
  function "*" (left, right: integer) return integer;
  function "/" (left, right: integer) return integer;
  function "rem" (left, right: integer) return integer;
  function "mod" (left, right: integer) return integer;

```

APPENDIX F OF THE Ada STANDARD

```
function "***" (left, right: integer) return integer;
```

```
type short_float is digits 6 range
```

```
-2#0.11111111111111111111111111111111#E127 ..
2#0.11111111111111111111111111111111#E127";
```

```
function "=" (left, right: short_float) return boolean;
function "/=" (left, right: short_float) return boolean;
function "<" (left, right: short_float) return boolean;
function "<=" (left, right: short_float) return boolean;
function ">" (left, right: short_float) return boolean;
function ">=" (left, right: short_float) return boolean;
function "+" (right: short_float) return short_float;
function "-" (right: short_float) return short_float;
function "abs" (right: short_float) return short_float;
function "+=" (left, right: short_float) return short_float;
function "-=" (left, right: short_float) return short_float;
function "**" (left, right: short_float) return short_float;
function "/" (left, right: short_float) return short_float;
function "***" (left, right: short_float) return short_float;
```

```
type float is digits 9 range
```

```
-2#0.1111111111111111111111111111111111111111111111111111111111111111#E127" ..
2#0.1111111111111111111111111111111111111111111111111111111111111111#E127";
```

```
function "=" (left, right: float) return boolean;
function "/=" (left, right: float) return boolean;
function "<" (left, right: float) return boolean;
function "<=" (left, right: float) return boolean;
function ">" (left, right: float) return boolean;
function ">=" (left, right: float) return boolean;
function "+" (right: float) return float;
function "-" (right: float) return float;
function "abs" (right: float) return float;
function "+=" (left, right: float) return float;
function "-=" (left, right: float) return float;
function "**" (left, right: float) return float;
function "/" (left, right: float) return float;
function "***" (left, right: float) return float;
```

```
function "**" (left: univ_integer; right: univ_real) return univ_real;
function "**" (left: univ_real; right: univ_integer) return univ_real;
function "/" (left: univ_real; right: univ_integer) return univ_real;
```

```
function "**" (left: any_fixed; right: any_fixed) return univ_fixed;
function "/" (left: any_fixed; right: any_fixed) return univ_fixed;
```

```
type character is
```

```
(nul, soh, stx, etx, eot, enq, eck, bel,
bs, ht, lf, vt, ff, cr, so, si,
dle, dcl, dc2, dc3, dc4, nak, syn, etb,
can, em, sub, esc, fs, gs, rs, us,
' ', '!', '"', '#', '$', '%', '&', '\'',
'(', ')', '*', '+', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', ':', ';', '<', '>', '?',
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',
'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '(', '|', '|', '~', del);
```

```
for character use
```

```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, .... 127);
```

APPENDIX F OF THE Ada STANDARD

```

package ascii is
  nul: constant character := nul; soh: constant character := soh;
  stx: constant character := stx; etx: constant character := etx;
  eot: constant character := eot; enq: constant character := enq;
  ack: constant character := ack; bel: constant character := bel;
  lf : constant character := lf; vt : constant character := vt;
  ff : constant character := ff; cr : constant character := cr;
  so : constant character := so; si : constant character := si;
  dle: constant character := dle; dc1: constant character := dc1;
  dc2: constant character := dc2; dc3: constant character := dc3;
  dc4: constant character := dc4; nak: constant character := nak;
  syn: constant character := syn; etb: constant character := etb;
  sub: constant character := sub; esc: constant character := esc;
  rs : constant character := rs; us : constant character := us;
  del: constant character := del;

  exclam      : constant character := '!';
  quotation   : constant character := '"';
  sharp       : constant character := '#';
  dollar      : constant character := '$';
  percent     : constant character := '%';
  ampersand   : constant character := '&';
  colon       : constant character := ':';
  semicolon   : constant character := ';';
  query       : constant character := '?';
  at_sign     : constant character := '@';
  l_bracket   : constant character := '[';
  back_slash  : constant character := '\';
  r_bracket   : constant character := ']';
  underline   : constant character := '_';
  grave       : constant character := '`';
  l_brace     : constant character := '{';
  bar         : constant character := '|';
  r_brace     : constant character := '}';
  tilde       : constant character := '~';

  lc_a: constant character := 'a';
  ...
  lc_z: constant character := 'z';

end ascii;

subtype natural is integer range 0 .. integer'last;
subtype positive is integer range 1 .. integer'last;

type string is array(positive range <>) of character;
pragma pack(string);
function "=" (left, right: string) return boolean;
function "/=" (left, right: string) return boolean;
function "<" (left, right: string) return boolean;
function "<=" (left, right: string) return boolean;
function ">" (left, right: string) return boolean;
function ">=" (left, right: string) return boolean;
function "&" (left: string; right: string) return string;
function "&" (left: character; right: string) return string;
function "&" (left: string; right: character) return string;
function "&" (left: character; right: character) return string;

type duration is delta 2#1.0#E-14 range
  -2#10000000000000000.0# ..
  2#1111111111111111.111111111111#;
function "=" (left, right: duration) return boolean;
function "/=" (left, right: duration) return boolean;
function "<" (left, right: duration) return boolean;
function "<=" (left, right: duration) return boolean;

```

APPENDIX F OF THE Ada STANDARD

```
function ">" (left, right: duration) return boolean;
function ">=" (left, right: duration) return boolean;
function "**" (left: duration; right: integer) return duration;
function "**" (left: integer; right: duration) return duration;
function "/" (left: duration; right: integer) return duration;

constraint_error : exception;
numeric_error : exception;
program_error : exception;
storage_error : exception;
tasking_error : exception;
end standard;
```

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 identified by names that begin with a dollar sign. A value is substituted for each of these names before the test is run. The values used for this validation are given below.

<u>Name and Meaning</u>	<u>Value</u>
\$BIG_ID1 Identifier of size MAX_IN_LEN with varying last character.	(1 .. 498 => 'A', 499 => '1')
\$BIG_ID2 Identifier of size MAX_IN_LEN with varying last character.	(1 .. 498 => 'A', 499 => '2')
\$BIG_ID3 Identifier of size MAX_IN_LEN with varying middle character.	(1 .. 249 => 'A', 250 => '1' , (251 .. 499 => 'A')
\$BIG_ID4 Identifier of size MAX_IN_LEN with varying middle character.	(1 .. 249 => 'A', 250 => '1' , (251 .. 499 => 'A')
\$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is MAX_IN_LEN characters long.	(1 .. 496 => '0', 497 .. 499 => "298")

TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
<p>\$BIG_REAL_LIT A real literal that can be either of floating- or fixed-point type, has value 690.0, and has enough leading zeroes to be MAX_IN_LEN characters long.</p>	(1 .. 493 => '0', 494 .. 499 => "69.OE1")
<p>\$BLANKS Blanks of length MAX_IN_LEN - 20</p>	(1 .. 479 => ' ')
<p>\$COUNT_LAST Value of COUNT'LAST in TEXT_IO package.</p>	2_147_483_647
<p>\$EXTENDED_ASCII_CHARS A string literal containing all the ASCII characters with printable graphics that are not in the basic 55 Ada character set.</p>	"abcdefghijklmnopqrstuvwxyz!\$%?@[]^`{}~"
<p>\$FIELD_LAST Value of FIELD'LAST in TEXT_IO package.</p>	2_147_483_647
<p>\$FILE_NAME_WITH_BAD_CHARS An illegal external file name that either contains invalid characters or is too long.</p>	"/illegal/file_name/2{}}\$%2102C.DAT"
<p>\$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.</p>	"/illegal/file_name/CE2102C*.DAT"
<p>\$GREATER_THAN_DURATION A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.</p>	100_000.0
<p>\$GREATER_THAN_DURATION_BASE_LAST The universal real value that is greater than DURATION'BASE'LAST.</p>	10_000_000.0
<p>\$ILLEGAL_EXTERNAL_FILE_NAME1 Illegal external file name.</p>	"/no/such/directory/ILLEGAL_EXTERNAL_FILE_NAME1"
<p>\$ILLEGAL_EXTERNAL_FILE_NAME2 Illegal external file names.</p>	"/no/such/directory/ILLEGAL_EXTERNAL_FILE_NAME2"

TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
<p>\$INTEGER_FIRST The universal integer literal expression whose value is INTEGER'FIRST.</p>	-2_147_483_648
<p>\$INTEGER_LAST The universal integer literal expression whose value is INTEGER'LAST.</p>	2_147_483_647
<p>\$LESS_THAN_DURATION A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.</p>	-100_000.0
<p>\$LESS_THAN_DURATION_BASE_FIRST The universal real value that is less than DURATION'BASE'FIRST.</p>	-10_000_000.0
<p>\$MAX_DIGITS Maximum digits supported for floating-point types.</p>	9
<p>\$MAX_IN_LEN Maximum input line length permitted by the implementation.</p>	499
<p>\$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.</p>	TINY_INTEGER
<p>\$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.</p>	16#FFFFFFFD#
<p>\$NON_ASCII_CHAR_TYPE An enumerated type definition for a character type whose literals are the identifier NON_NULL and all non-ASCII characters with printable graphics.</p>	(NON_NULL)

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. When testing was performed, the following 15 tests had been withdrawn at the time of validation testing for the reasons indicated:

- . B4A010C: The object declaration in line 18 follows a subprogram body of the same declarative part.
- . B83A06B: The Ada Standard 8.3(17) and AI-00330 permit the label LAB_ENUMERAL of line 80 to be considered a homograph of the enumeration literal in line 25.
- . BA2001E: The Ada Standard 10.2(5) states: "Simple names of all subunits that have the same ancestor library unit must be distinct identifiers." This test checks for the above condition when stubs are declared. However, the Ada Standard does not preclude the check being made when the subunit is compiled.
- . BC3204C: The file BC3204C4 should contain the body for BC3204C0 as indicated in line 25 of BC3204C3M.
- . C35904A: The elaboration of subtype declarations SFX3 and SFX4 may raise NUMERIC_ERROR (instead of CONSTRAINT_ERROR).
- . C41404A: The values of 'LAST and 'LENGTH are incorrect in IF statements from line 74 to the end of the test.
- . C48008A: This test requires that the evaluation of default initial values not occur when an exception is raised by an allocator. However, the Language Maintenance Committee (LMC) has ruled that such a requirement is incorrect (AI-00397/01).

WITHDRAWN TESTS

- . C4A014A: The number declarations in lines 19-22 are incorrect because conversions are not static.
- . C92005A: At line 40, "/"=" for type PACK.BIG_INT is not visible without a USE clause for package PACK.
- . CA1003B: This test requires all of the legal compilation units of a file containing some illegal units to be compiled and executed. According to AI-00255, such a file may be rejected as a whole.
- . CA3005A..D (4 tests): No valid elaboration order exists for these tests.
- . CE2107E: This test has a variable, TEMP_HAS_NAME, that needs to be given an initial value of TRUE.

END

1-87

Dt i c