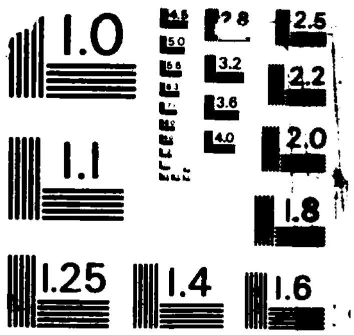
AD-A190 465 ADA COMPILER VALIDATION SUMMARY REPORT ALLIANT COMPUTER SYSTEMS CORPORATI (U) INFORMATION SYSTEMS AND TECHNOLOGY CENTER W-P AFB OH ADA VALI 09 JUN 87 1/1

UNCLASSIFIED AVF-VSR-88 0887 F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A190 465

REPORT DOCUMENTATION PAGE

1. REPORT NUMBER	[illegible] LOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Alliant Computer Systems Corp. Alliant FX/Ada Compiler, Version 1.0 Alliant FX/8 Host and Target	5. TYPE OF REPORT & PERIOD COVERED 9 June 1987 to 9 June 1988
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson AFB	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Ada Validation Facility ASD/SIOL Wright-Patterson AFB OH 45433-6503	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 9 June 1987
	13. NUMBER OF PAGES 39
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson	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DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601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)

87 12 29 158

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Alliant FX/Ada Compiler, Version 1.0, using Version 1.8 of the Ada® Compiler Validation Capability (ACVC). The Alliant FX/Ada Compiler is hosted on an Alliant FX/8 operating under Concentrix , Release 3.0. Programs processed by this compiler may be executed on an Alliant FX/8 operating under Concentrix, Release 3.0.

On-site testing was performed 8 June 1987 through 9 June 1987 at Alliant Computer Systems Corporation, Littleton MA, under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2210 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 170 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2210 tests were processed, results for Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests 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. There were 8 of the processed tests determined to be inapplicable. The remaining 2202 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	102	252	334	244	161	97	138	261	130	32	218	233	2202
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	14	73	86	3	0	0	1	1	0	0	0	0	178
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

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

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

AVF Control Number: AVF-VSR-88.0887
87-02-09-ACS

Ada® COMPILER
VALIDATION SUMMARY REPORT:
Alliant Computer Systems Corporation
Alliant FX/Ada Compiler, Version 1.0
Alliant FX/8 Host and Target

Completion of On-Site Testing:
9 June 1987

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

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

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

```
++++++++++++++++++++++++
+                      +
+ Place NTIS form here +
+                      +
++++++++++++++++++++++++
```

Ada® Compiler Validation Summary Report:

Compiler Name: Alliant FX/Ada Compiler, Version 1.0

Host:
Alliant FX/8 under
Concentrix, Release 3.0

Target:
Alliant FX/8 under
Concentrix, Release 3.0

Testing Completed 9 June 1987 Using ACVC 1.8

This report has been reviewed and is approved.

Ada Validation Facility
Georgeanne Chitwood
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

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

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

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

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Alliant FX/Ada Compiler, Version 1.0, using Version 1.8 of the Ada® Compiler Validation Capability (ACVC). The Alliant FX/Ada Compiler is hosted on an Alliant FX/8 operating under Concentrix , Release 3.0. Programs processed by this compiler may be executed on an Alliant FX/8 operating under Concentrix, Release 3.0.

On-site testing was performed 8 June 1987 through 9 June 1987 at Alliant Computer Systems Corporation, Littleton MA, under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2210 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 170 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2210 tests were processed, results for Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests 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. There were 8 of the processed tests determined to be inapplicable. The remaining 2202 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	102	252	334	244	161	97	138	261	130	32	218	233	2202
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	14	73	86	3	0	0	1	1	0	0	0	0	178
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

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

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

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

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

INTRODUCTION

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 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 direction of the AVF according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was conducted from 8 June 1987 through 9 June 1987 at Alliant Computer Systems Corporation, Littleton MA.

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:

Ada Validation Facility
ASD/SCOL
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 Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, FEB 1983.

2. Ada Validation Organization: Procedures and Guidelines, Ada Joint Program Office, 1 JAN 1987.

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 conformity 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 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 nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

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.
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 checks a compiler's conformity regarding a particular feature or 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	A test found to be incorrect and not used to check conformity to the Ada language specification. 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 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 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

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

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

Compiler: Alliant FX/Ada Compiler, Version 1.0

ACVC Version: 1.8

Certificate Expiration Date: 870608W1.08076

Host Computer:

Machine:	Alliant FX/8
Operating System:	Concentrix, Release 3.0
Memory Size:	16 megabytes

Target Computer:

Machine:	Alliant FX/8
Operating System:	Concentrix, Release 3.0
Memory Size:	16 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. This compiler is characterized by the following interpretations of the Ada Standard:

- 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 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 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 during execution. (See test E24101A.)

- 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.

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 subtype 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 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. 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.)

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.)

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 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.

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. 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 validation are given in Appendix D.

- Functions.

  An implementation may allow the declaration of a parameterless function and an enumeration literal having the same profile in the same immediate scope, or it may reject the function declaration. If it accepts the function declaration, the use of the enumeration literal's identifier denotes the function. This implementation rejects the declaration. (See test E66001D.)

- Representation clauses.

  The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it. While the operation of representation clauses is not checked by Version 1.8 of the ACVC, they are used in testing other language features. This implementation accepts 'SIZE and 'STORAGE_SIZE for tasks, 'STORAGE_SIZE for collections, and 'SMALL clauses. Enumeration representation clauses, including those that specify noncontiguous values, appear to be supported. (See tests C55B16A, C87B62A, C87B62B, C87B62C, and BC1002A.)

- Pragmas.

  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 AE2101C, AE2101H, CE2201D, CE2201E, and CE2401D.)

  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).)

  More than one internal file can be associated with each external file for sequential I/O for both reading and writing. (See tests CE2107A..F (6 tests).)

More than one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107A..F (6 tests).)

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

Temporary sequential files are given a name. Temporary direct files are given a name. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

- Generics.

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

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

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.8 of the ACVC contains 2399 tests. When validation testing of Alliant FX/Ada Compiler was performed, 19 tests had been withdrawn. The remaining 2380 tests were potentially applicable to this validation. The AVF determined that 178 tests were inapplicable to this implementation, and that the 2202 applicable tests were passed by the implementation.

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	69	865	1192	17	13	46	2202
Failed	0	0	0	0	0	0	0
Inapplicable	0	2	176	0	0	0	178
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	102	252	334	244	161	97	138	261	130	32	218	233	2202
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	14	73	86	3	0	0	1	1	0	0	0	0	178
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

3.4 WITHDRAWN TESTS

The following 19 tests were withdrawn from ACVC Version 1.8 at the time of this validation:

C32114A	B37401A	B49006A	C92005A
B33203C	C41404A	B4A010C	C940ACA
C34018A	B45116A	B74101B	CA3005A..D (4 tests)
C35904A	C48008A	C87B50A	BC3204C

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. For this validation attempt, 178 tests were inapplicable for the reasons indicated:

- C34001E, B52004D, B55B09C, and C55B07A use LONG_INTEGER which is not supported by this compiler.

- C34001G and C35702B use LONG_FLOAT which is not supported by this compiler.

- 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.

- C96005B checks implementations for which the smallest and largest values in type DURATION are different from the smallest and largest values in DURATION's base type. This is not the case for this implementation.

- The following 170 tests require a floating-point accuracy that exceeds the maximum of 15 supported by the implementation:

C24113L..Y (14 tests)	C35708L..Y (14 tests)	C45421L..Y (14 tests)
C35705L..Y (14 tests)	C35802L..Y (14 tests)	C45424L..Y (14 tests)
C35706L..Y (14 tests)	C45241L..Y (14 tests)	C45521L..Z (15 tests)
C35707L..Y (14 tests)	C45321L..Y (14 tests)	C45621L..Z (15 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 19 Class B tests:

B24204A	B2A003B	B38008A	B67001A	B91003B
B24204B	B2A003C	B41202A	B67001B	B95001A
B24204C	B33301A	B44001A	B67001C	B97102A
B2A003A	B37201A	B64001A	B67001D	

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.8 produced by the Alliant FX/Ada Compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and that the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Alliant FX/Ada Compiler using ACVC Version 1.8 was conducted on-site by a validation team from the AVF. The configuration consisted of an Alliant FX/8 host and target operating under Concentrix, Release 3.0.

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 splits during the prevalidation testing were included in their split form on the magnetic tape.

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 was compiled, and all executable tests were linked and run on the Alliant FX/Ada. Results were printed.

The compiler was tested using command scripts provided by Alliant Computer Systems Corporation and reviewed by the validation team. All default options were in effect for testing.

Tests were compiled, linked, and executed (as appropriate) using a single 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 Alliant Computer Systems Corporation, Littleton MA on 8 June 1987, and departed after testing was completed on 9 June 1987.

APPENDIX A

DECLARATION OF CONFORMANCE

Alliant Computer Systems Corporation has submitted the following declaration of conformance concerning the Alliant FX/Ada Compiler.

ALLIANT
Computer Systems Corporation

One Monarch Drive
Littleton, Massachusetts 01460
617-486-4950

DECLARATION OF CONFORMANCE

Compiler Implementor: Alliant Computer Systems Corporation
Ada® Validation Facility: ASD/SCOL, Wright-Patterson AFB, OH
Ada Compiler Validation Capability (ACVC) Version: 1.8

Base Configuration

Base Compiler Name: Alliant FX/Ada Compiler Version: Version 1.0
Host Architecture ISA: Alliant FX/8 OS&VER #: Concentrix, Release 3.0
Target Architecture ISA: Alliant FX/8 OS&VER #: Concentrix, Release 3.0

Implementor's Declaration

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

Andrew F. Halford
Alliant Computer Systems Corp.
Andrew F. Halford

Date: June 10, 1987

Owner's Declaration

I, the undersigned, representing Alliant, take full responsibility for implementation and maintenance of the Ada compiler 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 the Ada language compiler listed, and its host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Andrew F. Halford
Alliant Computer Systems Corp.
Andrew F. Halford

Date: June 10, 1987

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

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 clauses. The implementation-dependent characteristics of the Alliant FX/Ada Compiler, Version 1.0, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A). Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

    ...

 type INTEGER is range -2147483648 .. 2147483647;
 type TINY_INTEGER is range -128 .. 127;
 type SHORT_INTEGER is range -32768 .. 32767;
 type FLOAT is digits 15 range
 -2#0.11111111111111111111111111111111111111111111111111111#E1023
                                  ..
 2#0.11111111111111111111111111111111111111111111111111111#E1023
 type SHORT_FLOAT is digits 6 range
 -2#0.111111111111111111111111#E127
                                  ..
 2#0.111111111111111111111111#E127
 type DURATION is delta 2#1.0#E-14 range -2#100000000000000000.0#
                                  .. 2#11111111111111111.11111111111111#;

    ...

end STANDARD;
```

FX/ADA IMPLEMENTATION NOTES (ADA RM APPENDIX F)

FX/ADS Ada provides the full Ada language as specified in the Ada RM. Within the Ada RM, a number of sections contain the annotation *implementation dependent*, meaning that the interpretation is left to the compiler implementor. Alliant has attempted to make those choices that provide the programmer with an essentially unlimited capability to program in Ada. Consequently, an applications programmer can usually program in Ada according to the Ada RM and good engineering practices without consideration of any FX/ADS specifics.

Alliant provides the following Chapter 13 capabilities.

- representation clauses to the bit level and pragma PACK (RM 13.1)
- length clauses and unsigned types (8, 16 bit) (RM 31.2)
- enumeration representation clauses (RM 13.3)
- record representation clauses (RM 13.4)
- interrupt entries (RM 13.5.1)
- representation attributes (RM 13.7.2)
- machine code insertions and pragma IMPLICIT_CODE (RM 13.8)
- interface programming features, including pragma interface, pragma external_name, pragma interface_object, with directives, a.info, and external dependencies capabilities (RM 13.9)
- unchecked deallocations (RM 13.10.1)
- unchecked conversions (RM 13.10.2)
- shared generic bodies
- all-Ada runtime system

6.1 PROGRAM STRUCTURE AND COMPILATION ISSUES

6.1.1 Pragmas and Their Effects

pragma CONTROLLED is recognized by the implementation but has no effect in this release.

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

pragma EXTERNAL_NAME allows the user to specify a *link_name* for an Ada variable or subprogram so that the object can be referenced from other languages.

pragma IMPLICIT_CODE specifies that implicit code generated by the compiler is allowed or disallowed and is used only within a machine code procedure. It takes one of the identifiers ON or OFF as the single argument (default is ON). A warning is issued if OFF is used and any implicit code needs to be generated.

pragma INLINE — This pragma is implemented as described in Appendix B of the RM with the addition that recursive calls can be expanded with the pragma up to a maximum depth of 4. Warnings are produced for too-deep nestings or for bodies that are not available for inline expansion.

pragma INTERFACE supports calls to C and FORTRAN language functions with an optional link name for the subprogram. The Ada specifications can be either functions or procedures. All parameters must have mode IN.

For C, the types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS in SYSTEM. Record and array objects can be passed by reference using the ADDRESS attribute.

For FORTRAN, all parameters are passed by reference; the parameter types must have the type SYSTEM.ADDRESS. The result type for a FORTRAN function must be a scalar type.

The optional link name enables calling a function whose name is defined in another language, allowing characters in the name that are not allowed in an Ada identifier. Case sensitivity can then be preserved. Without the optional link name, the Ada compiler converts all identifiers to upper case. The link name overrides the default transformations that pragma INTERFACE performs on the name to create the unresolved reference name in the object module. For instance, the following example generates a reference for _Var1 with no case or other changes.

```
pragma INTERFACE (language-name, Var1, "_Var1");
```

pragma INTERFACE_OBJECT allows variables defined in another language to be referenced directly in Ada, replacing all occurrences of *variable_name* with an external reference to *link_name* in the object file.

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

pragma MEMORY_SIZE is recognized by the implementation, but has no effect. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling the SYSTEM package with altered values.

pragma OPTIMIZE is recognized by the implementation, but has no effect in the current release. See the -O option for ada for code optimization options.

pragma OPTIMIZE_CODE specifies that optimizations be allowed or disallowed and is used only within a machine code procedure. It takes one of the identifiers ON or OFF (default is ON).

pragma PACK will cause the compiler to choose a non-aligned representation for composite types. Packing will be to the nearest power of two bits.

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

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

pragma SHARE_BODY provides for the sharing of generic bodies (procedures and packages), when the generic parameters are restricted to enumeration, integer, and floating types. A 'parent' instantiation is created and subsequent generics of the same basic type can share its code, reducing compilation times.

pragma SHARED is recognized by the implementation, but has no effect in the current release.

pragma STORAGE_UNIT is recognized by the implementation, but has no effect. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling the SYSTEM package with altered values.

pragma SUPPRESS — The single parameter form of the **pragma** SUPPRESS is supported; the pragma applies from the point of occurrence to the end of the innermost enclosing block. DIVISION_CHECK and OVERFLOW_CHECK are not suppressible. The double parameter form of the pragma, with a name of an object, type, or subtype is recognized, but has no effect.

pragma SYSTEM_NAME is recognized by the implementation, but has no effect. The implementation does not allow SYSTEM to be modified by means of pragmas; however, the same effect can be achieved by recompiling the SYSTEM package with altered values.

6.1.2 Library Units

Compilation Units — Library Units — FX/ADS requires that a 'main' program must be a non-generic subprogram that is either a procedure or a function returning an Ada STANDARD.INTEGER (the predefined type). While a 'main' program may not be a generic subprogram, it may, however, be an instantiation of a generic subprogram.

Generic Declarations — FX/ADS does not require that a generic declaration and the corresponding body be part of the same compilation, and they are not required to exist in the same FX/ADS library. An error is generated if a single compilation contains two versions of the same unit.

FX/ADS provides for sharing of generic bodies (procedures and packages) when the generic parameters are restricted to enumeration types, integer types, and floating types. The pragma SHARE_BODY is used to require or suppress sharing.

The pragma SHARE_BODY is used to indicate desire to share or not share an instantiation. The pragma may reference either the generic unit or the instantiated unit. When it references a generic unit, it sets sharing on or off for all instantiations of that generic unless overridden by specific SHARE_BODY pragmas for individual instantiations. When it references an instantiated unit, sharing is on or off only for that unit. The default is to share all generics that can be shared unless the unit uses pragma INLINE.

The pragma SHARE_BODY is only allowed in the following places: immediately within a declarative part, immediately within a package specification, or after a library unit in a compilation, but before any subsequent compilation unit. The form of this pragma is

pragma SHARE_BODY (*generic-name, boolean-literal*)

Note that a parent instantiation is independent of any individual instantiation, therefore recompilation of a generic with different parameters has no effect on other compilations that reference it. The unit that caused compilation of a parent instantiation need not be referenced in any way by subsequent units that share the parent instantiation.

Sharing generics causes a slight execution time penalty because all type attributes must be indirectly referenced (as if an extra calling argument were added). However, it substantially reduces compilation time in most circumstances and reduces program size.

Alliant has compiled a unit, SHARED_IO, in the FX/ADS_STANDARD library that instantiates all Ada generic I/O packages. Thus, any instantiation of an Ada I/O generic package will share one of the parent instantiation generic bodies.

6.1.3 Representation Clauses

FX/ADS supports bit-level representation clauses.

pragma PACK — Objects and components are packed to the nearest power of two bits.

Length Clauses — FX/ADS supports all representation clauses to the byte level in the current release. The size specification T'SMALL is not supported except when the representation specification is the same as the value 'SMALL for the base type.

Enumeration Representation Clauses — Enumeration representation clauses are supported.

Record Representation Clauses — The only restriction on record representation specifications is the following: if a field does not start and end on a storage unit boundary, it must be possible to get it into a register with one move instruction. It must fit into 4 bytes starting on a word boundary.

```
for rec use
   record at mod 16;
      a at 0 range 0 .. 9;
      b at 1 range 2 .. 30;
   end record;
```

Note that in the example above a size specification could be given, e.g.,

```
for rec'size use 39;
```

but due to alignment, such a record would always take 5 bytes (i.e., 40 bits).

Interrupts — Interrupts are supported.

Representation Attributes — The ADDRESS attribute is not supported for the following entities:

- static constants
- packages
- tasks
- labels
- entries

Machine Code Insertions — Machine code insertions are supported.

Address Clauses — Address clauses are not supported.

6.1.4 Default Representations

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. Many limits in the FX/ADS implementation are variable as allocated memory is used extensively; therefore, many constructs are limited only by the virtual memory space available to a process.

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

Record and Array Sizes — The maximum size of a statically sized record type is 4,000,000 * STORAGE_UNITs. A record type or array type declaration that exceeds these limits will generate a warning message.

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 STORAGE_UNITs. This is the value returned by T'STORAGE_SIZE for a task type T.

Default Collection Size — In the absence of an explicit STORAGE_SIZE length specification, the default collection size for an access type is 100,000 STORAGE_UNITs. This is the value returned by T'STORAGE_SIZE for an access type T.

Limit on Declared Objects — Declared object size is limited only by available virtual space for the process.

Stack Size— The compiler and other large dynamic compiled programs can occasionally give problems due to the shell's stack limit. Altering the stack size and recompiling is sometimes necessary.

The C shell allows the default stack size of 512K to be reset up to the limit of the process size (usually 6112K bytes). To change the stacksize for the C shell, execute the following command.

```
limit stacksize number
```

Most Bourne shell implementations do not allow the stack size to be modified.

6.1.5 Conversions

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

6.1.6 Deallocations

Any objects may be deallocated, but the deallocation succeeds only for constrained objects. No error is reported for objects that cannot be freed. No checks are currently performed on released objects.

6.1.7 Character Set

FX/ADS provides the full *graphic_character* textual representation for programs.

6.1.8 Source File Structure/Restrictions

Lexical Elements, Separators, and Delimiters — FX/ADS Ada uses the normal Concentrix I/O for reading source files. Each line is terminated by a newline character (ASCII.LF). Source lines may contain up to 500 characters, including the terminator. All variable-length Ada elements, such as identifiers and literals, may extend up to the full 499-character limit.

6.2 PREDEFINED ENVIRONMENT

6.2.1 Supported Packages

The following predefined Ada packages given by RM Appendix C(22) are provided.

```
package STANDARD
package CALENDAR
package SYSTEM
generic procedure UNCHECKED_DEALLOCATION
generic function UNCHECKED_CONVERSION
generic package SEQUENTIAL_IO
generic package DIRECT_IO
package TEXT_IO
package IO_EXCEPTIONS
package LOW_LEVEL_IO
package MACHINE_CODE
```

```
-- in package STANDARD

type TINY_INTEGER is      range -128 .. 127;
type SHORT_INTEGER is     range -32768 .. 32767;
type INTEGER is           range -2147483648 .. 2147483647;
type SHORT_FLOAT is       short_float is digits 6 range
                          -2#0.111111111111111111111111#E127
                          ..
                          2#0.111111111111111111111111#E127;
type FLOAT is             float is digits 15 range
-2#0.11111111111111111111111111111111111111111111111111111#E1023
                          ..
2#0.11111111111111111111111111111111111111111111111111111#E1023;
type DURATION is          delta 2#1.0#E-14 range
-2#100000000000000000.0# .. 2#11111111111111111.11111111111111#;
```

```
-- in package DIRECT_IO

type COUNT is       range 0 .. 2_147_483_647;


-- in package TEXT_IO

type COUNT is       range 0 .. 2_147_483_647;
subtype FIELD is    INTEGER range 0 .. 132;
```

6.2.2 Input/output

The Ada I/O system is implemented on top of basic Concentrix I/O. Both formatted and binary I/O are available. There are no restrictions on the types with which DIRECT_IO and SEQUENTIAL_IO can be instantiated, except that the element size must be less than a maximum given by the variable SYSTEM.MAX_REC_SIZE. This variable may be set to any value prior to the generic instantiation; thus, the user may use any element size. DIRECT_IO may be instantiated with unconstrained types, but each element will be padded out to the maximum possible for that type or to SYSTEM.MAX_REC_SIZE, whichever is smaller. No checking — other than normal static Ada type checking — is done to ensure that values from files are read into correctly sized and typed objects.

Input-output under Concentrix is similar to the C language implementation. FX/ADS file and terminal input-output are identical in most respects and differ only in the frequency of buffer flushing. Output is buffered (buffer size is 1024 bytes). The buffer is always flushed after each write request if the destination is a terminal. The procedure FILE_SUPPORT.ALWAYS_FLUSH (*file_ptr*) is provided for flushing the buffer for nonterminal output. A single call to this procedure will cause flushing of the buffer after all subsequent output requests. See the source code for file_io_body.a in the standard library.

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 a 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 before instantiating DIRECT_IO to provide an upper limit on the record size. The maximum size supported is 1024 * 1024 * 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.

6.2.3 Package system.a

```
package SYSTEM
is
        type NAME is ( fx_unix );

        SYSTEM_NAME         : constant NAME := fx_unix;

        STORAGE_UNIT        : constant := 8;
        MEMORY_SIZE         : constant := 16_777_216;

        -- System-Dependent Named Numbers

        MIN_INT             : constant := -2_147_483_648;
        MAX_INT             : constant := 2_147_483_647;
        MAX_DIGITS          : constant := 15;
        MAX_MANTISSA        : constant := 31;
        FINE_DELTA          : constant := 2.0**(-30);
        TICK      : constant := 0.01;

        -- Other System-dependent Declarations

        subtype PRIORITY is INTEGER range 0 .. 99;

        MAX_REC_SIZE : integer := 64*1024;

        type ADDRESS is private;

        NO_ADDR : constant ADDRESS;

        function PHYSICAL_ADDRESS(I: INTEGER) return ADDRESS;
        function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
        function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
        function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
        function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
        function ADDR_DIFF(A, B: ADDRESS) return INTEGER;
        function INCR_ADDR(A: ADDRESS;
                 INCR: INTEGER) return ADDRESS;
        function DECR_ADDR(A: ADDRESS;
                 DECR: INTEGER) return ADDRESS;

        function  ">"(A,  B:  ADDRESS)  return  BOOLEAN  renames
ADDR_GT;
```

```
        function "<"(A, B: ADDRESS) return BOOLEAN
                renames ADDR_LT;
        function ">="(A, B: ADDRESS) return BOOLEAN
                renames ADDR_GE;
        function "<="(A, B: ADDRESS) return BOOLEAN
                renames ADDR_LE;
        function "-"(A, B: ADDRESS) return INTEGER
                renames ADDR_DIFF;
        function "+"(A: ADDRESS; INCR: INTEGER) return ADDRESS
                renames INCR_ADDR;
        function "-"(A: ADDRESS; DECR: INTEGER) return ADDRESS
                renames DECR_ADDR;

        pragma inline(ADDR_GT);
        pragma inline(ADDR_LT);
        pragma inline(ADDR_GE);
        pragma inline(ADDR_LE);
        pragma inline(ADDR_DIFF);
        pragma inline(INCR_ADDR);
        pragma inline(DECR_ADDR);
        pragma inline(PHYSICAL_ADDRESS);

private

        type ADDRESS is new integer;

        NO_ADDR : constant ADDRESS := 0;

end SYSTEM;
```

6.2.4 Other Packages in standard

package MACHINE_CODE

The general definition of **package** MACHINE_CODE provides an assembly language interface for the target machine including the necessary record types needed in the code statement, an enumeration type containing all the opcode mneumonics, a set of register definitions, and a set of addressing mode functions. Also supplied (for use only in units that WITH MACHINE_CODE) are **pragma** IMPLICIT_CODE and the attribute 'REF.

Machine code statements take operands of **type** OPERAND, a private type that forms the basis of all machine code address formats for the target.

The general syntax of a machine code statement is

```
CODE_n' ( opcode, operand {, operand}) );
```

where *n* indicates the number of operands in the aggregate.

When there is a variable number of operands, they are listed within a subaggregate using the syntax shown below.

```
CODE_n' ( opcode, ( operand {, operand} );
```

In the example shown below, code_2 is a record 'format' whose first arguement is an enumeration value of type OPCODE followed by two operands of type OPERAND.

```
CODE_2'(add, a'ref, b'ref);
```

For those opcodes requiring no operands, named notation must be used. (See Ada RM 4.3(4).)

```
CODE_0' ( op => opcode );
```

The *opcode* must be an enumeration literal (i.e., it cannot be an object, attribute, or a rename). An *operand* can only be an entity defined in MACHINE_CODE of the 'REF attribute.

The arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE. The 'REF attribute may not be used as an argument in any of these functions.

'REF

The attribute 'REF denotes the effective address of the first of the storage units allocated to its object. For a subprogram, package, task unit, or label, it refers to the address of the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, it refers to the corresponding hardware interrupt. The attribute is of type OPERAND defined in package MACHINE_CODE and is allowed only within a machine code procedure. 'REF is not supported for a package, task unit, or entry. (See Section F.4.8 for more information on the use of this attribute.)

pragma IMPLICIT_CODE

pragma IMPLICIT_CODE specifies that implicit code generated by the compiler is allowed or disallowed and is used only within a machine code procedure. It takes one of the identifiers ON or OFF as the single argument (default is ON). A warning is issued if OFF is used and any implicit code needs to be generated.

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.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..498 =>'A', 499 =>'1')
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..498 =>'A', 499 =>'2')
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..248 => 'A', 249 => '3', 250..499 => 'A')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..248 => 'A', 249 => '4', 250..499 => '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.	(1..496 => '0', 497..499 => "298")

Name and Meaning	Value
$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 the size of the maximum line length.	(1..493 => '0', 494..499 => "69.0E1")
$BLANKS A sequence of blanks twenty characters fewer than the size of the maximum line length.	(1..479 =>' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
$EXTENDED_ASCII_CHARS A string literal containing all the ASCII characters with printable graphics that are not in the basic 55 Ada character set.	"abcdefghijklmnopqrstuvwxyz" & "&!$%?@[]^`{}~"
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2_147_483_647
$FILE_NAME_WITH_BAD_CHARS An illegal external file name that either contains invalid characters, or is too long if no invalid characters exist.	"/illegal/file_name/2{]$%2102c.DAT"
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character, or is too long if no wild card character exists.	"illegal/file_name/CE2102C*.DAT"
$GREATER_THAN_DURATION A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST if any, otherwise any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST The universal real value that is greater than DURATION'BASE'LAST, if such a value exists.	10_000_000.0

Name and Meaning	Value
$ILLEGAL_EXTERNAL_FILE_NAME1 An illegal external file name.	"""no/such/directory/" & "ILLEGAL_EXTERNAL_FILE_NAME1"
$ILLEGAL_EXTERNAL_FILE_NAME2 An illegal external file name that is different from $ILLEGAL_EXTERNAL_FILE_NAME1.	"""no/such/directory/" & "ILLEGAL_EXTERNAL_FILE_NAME2"
$INTEGER_FIRST The universal integer literal expression whose value is INTEGER'FIRST.	-(2**31)
$INTEGER_LAST The universal integer literal expression whose value is INTEGER'LAST.	(2**31)-1
$LESS_THAN_DURATION A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST if any, otherwise any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST The universal real value that is less than DURATION'BASE'FIRST, if such a value exists.	-10_000_000.0
$MAX_DIGITS The universal integer literal whose value is the maximum digits supported for floating-point types.	15
$MAX_IN_LEN The universal integer literal whose value is the maximum input line length permitted by the implementation.	499
$MAX_INT The universal integer literal whose value is SYSTEM.MAX_INT.	(2**31)-1

Name and Meaning	Value
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER if one exists, otherwise any undefined name.	TINY_INTEGER
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFD#
$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.	(NON_NULL)

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 19 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.

- C32114A: An unterminated string literal occurs at line 62.
- B33203C: The reserved word "IS" is misspelled at line 45.
- C34018A: The call of function G at line 114 is ambiguous in the presence of implicit conversions.
- C35904A: The elaboration of subtype declarations SFX3 and SFX4 may raise NUMERIC_ERROR instead of CONSTRAINT_ERROR as expected in the test.
- B37401A: The object declarations at lines 126 through 135 follow subprogram bodies declared in the same declarative part.
- C41404A: The values of 'LAST and 'LENGTH are incorrect in the *if* statements from line 74 to the end of the test.
- B45116A: ARRPRIBL1 and ARRPRIBL2 are initialized with a value of the wrong type--PRIBOOL_TYPE instead of ARRPRIBOOL_TYPE--at line 41.
- C48008A: The assumption that evaluation of default initial values occurs when an exception is raised by an allocator is incorrect according to AI-00397.
- B49006A: Object declarations at lines 41 and 50 are terminated incorrectly with colons, and *end case*; is missing from line 42.
- B4A010C: The object declaration in line 18 follows a subprogram body of the same declarative part.

- B74101B: The <u>begin</u> at line 9 causes a declarative part to be treated as a sequence of statements.
- C87B50A: The call of "/=" at line 31 requires a use clause for package A.
- C92005A: The "/=" for type PACK.BIG_INT at line 40 is not visible without a use clause for the package PACK.
- C940ACA: The assumption that allocated task TT1 will run prior to the main program, and thus assign SPYNUMB the value checked for by the main program, is erroneous.
- CA3005A..D (4 tests): No valid elaboration order exists for these tests.
- BC3204C: The body of BC3204C0 is missing.

END
DATE
FILMED
DTIC
4/88