UNCLASSIFIED

DTIC FILE COPY (2)

AD-A211 678

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Content
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	Ada Compiler Validation Summary Report: Harris Corporation, Computer System Division Harris Ada, Version 5.0, Harris H1000 (Host & Target), 89062W1.10103
5. TYPE OF REPORT & PERIOD COVERED	27 June 1989 to 27 June 1990
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	Wright-Patterson AFB Dayton, OH, USA
8. CONTRACT OR GRANT NUMBER(s)	
9. PERFORMING ORGANIZATION AND ADDRESS	Wright-Patterson AFB Dayton, OH, USA
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS	Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081
12. REPORT DATE	
13. NUMBER OF PAGES	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	Wright-Patterson AFB Dayton, OH, USA
15. SECURITY CLASS (of this report)	UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

UNCLASSIFIED

DTIC ELECTE AUG 22 1989 S B D

18. SUPPLEMENTARY NOTES

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

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

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

Harris Corporation, Computer Systems Division Harris Ada, Version 5.0, Harris H1000 under VOS, 8.1 (Host & Target), Wright-Patterson AFB, ACVC 1.10

89 8 21 190

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AVF Control Number: AVF-VSR-263.0689
89-04-10-HAR

Ada COMPILER VALIDATION SUMMARY REPORT:

Certificate Number: 890627W1.10103
Harris Corporation, Computer Systems Division
Harris Ada, Version 5.0
Harris H1000

Completion of On-Site Testing:
June 27, 1989

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

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

Ada Compiler Validation Summary Report:

Compiler Name: Harris Ada, Version 5.0

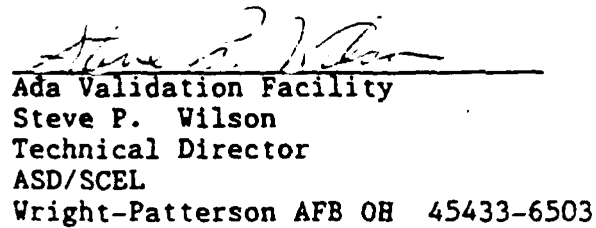
Certificate Number: 890627W1.10103

Host: Harris H1000 under
VOS, 8.1

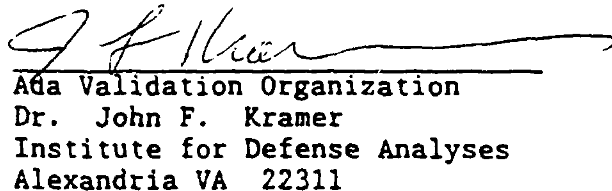
Target: Harris H1000 under
VOS, 8.1

Testing Completed June 27, 1989 using ACVC 1.10

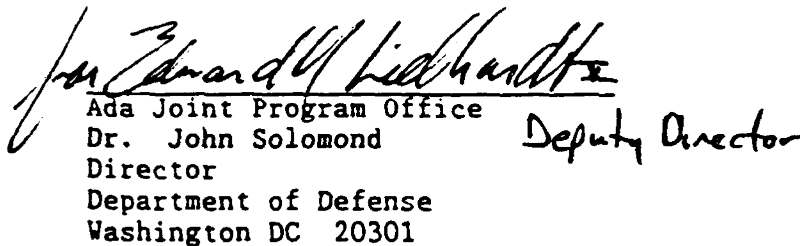
This report has been reviewed and is approved.



Ada Validation Facility
Steve P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503



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



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

Deputy Director

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT 1-2
1.2 USE OF THIS VALIDATION SUMMARY REPORT 1-2
1.3 REFERENCES. 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 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. 3-2
3.4 WITHDRAWN TESTS 3-2
3.5 INAPPLICABLE TESTS. 3-2
3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS. . 3-6
3.7 ADDITIONAL TESTING INFORMATION. 3-6
3.7.1 Prevalidation 3-6
3.7.2 Test Method 3-7
3.7.3 Test Site 3-8

APPENDIX A DECLARATION OF CONFORMANCE

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

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

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

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

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

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed June 27, 1989 at Ft. Lauderdale, FL.

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/SCEL
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, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

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

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

1.4 DEFINITION OF TERMS

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

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

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

Applicant The agency requesting validation.

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

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

Compiler A processor for the Ada language. In the context of this report, a compiler is any language processor, including

cross-compilers, translators, and interpreters.

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

Host — The computer on which the compiler resides.

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

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

Target — The computer for which a compiler generates code.

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

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

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors because of the way in which a program library is used at link time.

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

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

illegal construct that it contains is detected by the compiler.

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

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

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

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

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate

tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

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

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

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

Compiler: Harris Ada, Version 5.0

ACVC Version: 1.10

Certificate Number: 890627W1.10103

Host Computer:

Machine:	Harris H1000
Operating System:	VOS 8.1
Memory Size:	12 MB Physical Memory

Target Computer:

Machine:	Harris H1000
Operating System:	VOS 8.1
Memory Size:	12 MB Physical Memory

2.2 IMPLEMENTATION CHARACTERISTICS

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

a. Capacities.

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

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

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

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

b. Predefined types.

(1) There are no additional predefined types in package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

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

(1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

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

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

(4) Sometimes NUMERIC_ERROR is raised when an integer literal

operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) NUMERIC_ERROR is raised when a literal operand in a fixed-point membership test is outside the range of the base type. No exception is raised when a literal operand in a fixed-point comparison test is outside the range of the base type. (See test C45252A.)

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

d. Rounding.

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

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

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

e. Array types.

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

For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

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

(3) NUMERIC_ERROR is raised when 'LENGTH is applied to a null array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

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

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

(6) 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.)

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

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

f. Discriminated types.

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

g. Aggregates.

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

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

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

h. Pragmas.

(1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B, EA3004C..D, and CA3004E..F.)

i. Generics

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

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

j. Input and output

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

(2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) CREATE with modes IN_FILE and OUT_FILE is supported for SEQUENTIAL_IO. Open with mode IN_FILE and OUT_FILE is supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

(4) CREATE with modes IN_FILE, OUT_FILE, and INOUT_FILE is supported for DIRECT_IO. Open with modes (N_FILE, OUT_FILE, and INOUT_FILE is supported for DIRECT_IO. (See tests CE2102F, CE2102I..J, CE2102R, CE2102T, and CE2102V.)

(5) CREATE with modes IN_FILE and OUT_FILE is supported for text files. OPEN with modes IN_FILE and OUT_FILE is supported for text files. (See tests CE3102E and CE3102I..K.)

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

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

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

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

(10) Temporary sequential files are given names and deleted when closed. (See test CE2108A.)

(11) Temporary direct files are given names and deleted when closed. (See test CE2108C.)

(12) Temporary text files are given names and deleted when closed. (See test CE3112A.)

(13) More than one internal file cannot be associated with each external file for sequential files when writing or reading. (See tests CE2107A..E, CE2102L, CE2110B, and CE2111D.)

(14) More than one internal file cannot be associated with each external file for direct files when writing or reading. (See tests CE2107F..H (3 tests), CE2110D, and CE2111H.)

(15) More than one internal file cannot be associated with each external file for text files when writing or reading. (See tests CE3111A,B,D,E, CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 496 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 285 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 12 tests were required to successfully demonstrate the test objective. (See section 3.6.)

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

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1127	1831	17	27	46	3177
Inapplicable	0	11	484	0	1	0	496
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	192	546	485	242	172	99	158	331	137	36	252	246	281	3177
Inappl	20	103	195	6	0	0	8	1	0	0	0	123	40	496
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

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

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84M	CD2A84N	CD2B15C	CD2D11B	CD5007B	CD50110
ED7004B	ED7005C	ED7005D	ED7006C	ED7006D	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

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

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 496 tests were inapplicable for the reasons indicated:

a. The following 285 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113F..Y	C35705F..Y	C35706F..Y	C35707F..Y
C35708F..Y	C35802F..Z	C45241F..Y	C45321F..Y
C45421F..Y	C45521F..Z	C45524F..Z	C45621F..Z

C45641F..Y C46012F..Z

b. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

c. C35702B and B86001U are not applicable because this implementation supports no predefined type LONG_FLOAT.

d. The following 16 tests are not applicable because this implementation does not support a predefined type SHORT_INTEGER:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	B86001V
CD7101E				

e. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

f. C45231D, B86001X, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

g. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 47.

h. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package System. This test recompiles package System, making package TEXT_IO, and hence package REPORT, obsolete.

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

j. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

k. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

l. CD1009C, CD2A41A..B (2 tests), CD2A41E, and CD2A42A..J (10 tests) are not applicable because this implementation does not support size clauses for floating point types.

m. CD2A61I and CD2A61J are not applicable because this implementation does not support size clauses for array types, which imply compression, with component types of composite or floating point

types. This implementation requires an explicit size clause on the component type.

n. CD2A81A..F (6 tests), CD2A83A..C,E,F (5 tests), CD2A84B..I (8 tests), CD2A84K..L (2 tests), ED2A86A and CD2A87A are not applicable because this implementation does not support size clauses for access types.

o. CD2A91A..E (5 tests) are not applicable because this implementation does not support size clauses for tasks or task types.

p. The following 76 tests are not applicable because this implementation does not support addresses for variables and constants:

CD5003B..I (8)	CD5011A..I (9)	CD5011K..N (4)	CD5011Q..S (3)
CD5012A..J (10)	CD5012L..M (2)	CD5013A..I (9)	CD5013K..O (5)
CD5013R..S (2)	CD5014A..O (15)	CD5014R..Z (9)	

q. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

r. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

s. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

t. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

u. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

v. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

w. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

x. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

y. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

z. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

aa. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

ab. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

ac. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

ad. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

ae. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

af. CE2107A..F (6 tests), CE2107L, CE2110B, and CE2111D are not applicable because multiple internal files cannot be associated with the same external file for sequential files. The proper exception is raised when multiple access is attempted.

ag. CE2107G..H (2 tests), CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

ah. CE3102E is inapplicable because this implementation supports CREATE with IN_FILE mode for text files.

ai. CE3102F is inapplicable because this implementation supports RESET of external files.

aj. CE3102G is inapplicable because this implementation supports deletion of external files.

ak. CE3102I is inapplicable because this implementation supports CREATE with OUT_FILE mode for text files.

al. CE3102J is inapplicable because this implementation supports OPEN with IN_FILE mode for text files.

am. CE3102K is inapplicable because this implementation supports OPEN with OUT_FILE mode for text files.

an. CE3111A,B,D,E (4 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

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

Modifications were required for 12 tests.

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

B24009A	B25002A	B33301B	B36002A	B38003A	B38003B
B38009A	B38009B	B41202A	BC1303F	BC3005B	

The following test was graded using a modified evaluation criterion:

In test CE3804H, the string, "-3.525", is written to a text file, and a later attempt is made to read these characters as the value of a fixed-point variable. That variable is then compared to the real literal -3.525; this implementation finds the values not equal and reports failed. Since the real literal, -3.525, is not a model number, the AVO has ruled this test as passed for this implementation.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

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

3.7.2 Test Method

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

Host computer:	Harris H1000
Host operating system:	VOS, 8.1
Target computer:	Harris H1000
Target operating system:	VOS, 8.1
Compiler:	Harris Ada, Version 5.0

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 modifications during the prevalidation testing were included in their modified 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, linked, and all executable tests were run on the Harris H1000. Results were printed from the host computer.

The compiler was tested using command scripts provided by Harris Corporation, Computer Systems Division and reviewed by the validation team. The compiler was tested using all default option settings except for the following:

OPTION	EFFECT
-el	If warning or errors occur during compilation, generate a full source listing with the warning/ error messages included in the listing.
-w	Suppress compilation warning messages.
-L	Generate a full source listing even if no errors or warnings appeared during compilation.
-M unit_name	Create an executable image for main program unit_name.
-o exe_name	(with -M) Name the executable image exe_name.

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

Testing was conducted at Ft. Lauderdale, FL and was completed on June 27, 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Harris Corporation, Computer Systems Division has submitted the following Declaration of Conformance concerning the Harris Ada.

DECLARATION OF CONFORMANCE

Compiler Implementor: Harris Corporation, Computer Systems Division
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name: Harris Ada Version: 5.0
Host Architecture ISA: Harris H1000 OS&VER #: VOS 8.1
Target Architecture ISA: Harris H1000 OS&VER #: VOS 8.1
Project Control Number #: 89-04-10-HAR

Derived Compiler Registration

Derived Compiler Name: Harris Ada Version: 5.0
Host Architecture ISA: Harris H60 or Harris H700 or Harris H800 or Harris H900 or Harris H1000 or Harris H1100 or Harris H1200 or Harris H1500 or Harris H1600.
Host OS&VER #: VOS 8.1
Target Architecture ISA: Harris H60 or Harris H700 or Harris H800 or Harris H900 or Harris H1000 or Harris H1100 or Harris H1200 or Harris H1500 or Harris H1600.
Target OS&VER #: VOS 8.1
Any host in combination with any target.

Implementor's Declaration

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

[signature] Date: 5-10-89

Harris Corporation, Computer Systems Division
Wendell Norton, Director of Contracts

Owner's Declaration

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

Wendell Norton Date: 5-5-89

Harris Corporation, Computer Systems Division
Wendell Norton, Director of Contracts

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Harris Ada, Version 5.0, as described in this Appendix, are provided by Harris Corporation, Computer Systems Division. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

   type INTEGER is range -8_388_608 .. 8_388_607;

   type FLOAT is digits 9 range -1.70141183E+038 .. 1.70141183E+038;

   type DURATION is delta 2.0**(-14) range -8388607.99995 .. 8388607.99978;

   ...

end STANDARD;
```

APPENDIX F

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

F.1 PROGRAM STRUCTURE AND COMPILATION

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). A "main" program cannot be a generic subprogram or an instantiation of a generic subprogram.

F.2 PRAGMAS

F.2.1 Implementation-Dependent Pragmas

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

Pragma INLINE is implemented as described in section 6.3.2 and Appendix B of the Ada RM. This implementation expands recursive subprograms marked with the pragma up to a maximum nesting depth of 4. Warnings are produced for nesting depths greater than this or for bodies that are not available for inline expansion.

Pragma INTERFACE is recognized by the implementation and support calls to C and FORTRAN language functions. 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 defined in the package SYSTEM. Record and array objects can be passed by reference using the ADDRESS attribute. The default link name is the symbolic representation of the simple name converted to lowercase. The link name of interface routines can be changed via the implementation-defined pragma **external_name.**

For FORTRAN, all parameters are passed by reference. The parameter types must have the type ADDRESS defined in the package SYSTEM. The result type for a FORTRAN function must be a scalar type. Care should be taken when using tasking and FORTRAN functions. Since FORTRAN is not reentrant, it is recommended that an Ada controller task be used to access FORTRAN functions. The default link name is the symbolic representation of the simple name converted to lowercase, with a leading and trailing underscore ("_") character. The link name of interface routines can be changed via the implementation-defined pragma **external_name.**

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

Pragma MEMORY_SIZE is recognized by the implementation but has no visible effect. The implementation restricts the argument to the predefined value in the package system.

Pragma OPTIMIZE is recognized by the implementation but has no effect in this release. See the -O option for **ada** for code optimization options, or the implementation defined pragma. OPT_LEVEL.

Pragma PACK causes the compiler to choose a non-aligned representation for elements of composite types. Application of the pragma will cause objects to be packed to the bit level.

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. Priorities on the H-series range from 0 to 23, with 23 being the most urgent.

Pragma SHARED is recognized by the implementation but has no effect.

Pragma STORAGE_UNIT is recognized by the implementation but has no visible effect. The implementation restricts the argument to the predefined value in the package system.

Pragma SUPPRESS in the single parameter form is supported and applies from the point of occurrence to the end of the innermost enclosing block. 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 visible effect. The implementation provides only one enumeration value for SYSTEM_NAME in the package SYSTEM.

F.2.2 Implementation-Defined Pragmas

Pragma EXTERNAL_NAME provides a method for specifying an alternative ***link name*** for variables, functions and procedures. The required parameters are the simple name of the object and a string constant representing the link name. An underscore is automatically prepended to the specified name, unless the first character of the name is an underscore. Note that this pragma is useful for referencing functions and procedures that have had pragma INTERFACE applied to them, in such cases where the functions or procedures have link names that do not conform to Ada identifiers. The pragma must occur after any such applications of pragma INTERFACE and within the same declarative part or package specification that contains the object.

Pragma INTERFACE_OBJECT provides an interface to objects defined externally from the Ada compilation model, or an object defined in a foreign language. For example, a variable defined in the run-time system may be accessed via the pragma. This pragma has two required parameters, the first being the simple name of an Ada variable to be associated with the foreign object. The second parameter is a string constant that defines the link name of the object. The variable declaration must occur before the pragma and both must occur within the same declarative part or package specification.

Pragma INTERFACE_COMMON_OBJECT provides an interface to objects defined in foreign languages as common blocks. Its semantics and syntax are identical to those of pragma **interface_object** except that the second parameter is required. The second parameter must be a string constant representing the link name of the common block as defined by the foreign language.

Pragma INTERFACE_MCOM_OBJECT provides an interface to Monitor Common objects as defined externally—through a foreign language or some other means. Its semantics and syntax are identical to those of pragma **interface_common_object** except for an additional third parameter. The required third parameter must be a string constant representing the name of the Monitor Common disc area. This can be a fully rooted or relative pathname or a VOS areaname. (Note: The VOS convention of automatically inserting the characters "M " before the final name is retained, therefore, the user should not specify these characters.) Specification of this pragma does not cause the Monitor Common disc area to be created.

Pragma SHARED_PACKAGE provides for the sharing and communication of library level packages. All variables declared in a package marked pragma SHARED_PACKAGE (henceforth referred to as

a shared package) are allocated in a VOS Monitor Common area that is created and maintained by the implementation. The pragma can only be applied to library level package specifications. Each package specification nested in a shared package will also be shared and all objects declared in the nested packages reside in the same VOS Monitor Common area as the outer package.

The implementation restricts the kinds of objects that can be declared in a shared package. No unconstrained or dynamically sized objects can be declared in a shared package. No access type objects can be declared in a shared package. No explicit initialization of objects can occur in a shared package. If any of these restrictions are violated, a warning message is issued and the package is not shared. These restrictions apply to nested packages as well. Note that if a nested package violates one of the above restrictions, it prevents the sharing of all enclosing packages as well.

Task objects are allowed within shared packages, however, the tasks as well as the data defined within those tasks are not shared.

Pragma SHARED_PACKAGE accepts as an optional argument that, if specified, must be a string constant containing a blank separated list of VOS disc area control options, as defined by the following:

- **N=name** specifies the name of the Monitor Common disc area to be created. If this parameter is not specified, the implementation chooses a VUE pathname and creates the file under the .mcom HAPSE subdirectory on the host system.

- **P=n** where n is the pack number used when the Monitor Common disc area is created.

- **G=n** where n is the granule size in sectors used when the Monitor Common disc area is created.

- NR specifies that the Monitor Common area is to be Non-Resident, which is the default.

- RS specifies that the Monitor Common area is to be Resident.

- **O=usernum** specifies the owner of the Monitor Common disc area.

- **PR,PW,PD,OW,AR,AW,AD,OD** specifies the access permissions of the Monitor Common disc area.

- **L=address** specifies an address where the Monitor Common disc area is to be bound into physical memory. This is useful for sharing packages across systems configured with shared memory. Note that the RS control option must be specified if L=address is used.

With the valid application of pragma **SHARED_PACKAGE** to a library level package, the following assumptions can be made about the objects declared in the package:

- The lifetime of such objects is greater than the lifetime defined by the complete execution of a single program.

- The state of such objects is not changed between invocations of programs that reference objects, except as defined by the recreation of such programs.

- A program that causes the creation of such an object during the elaboration of a shared package retains the state of the object, if it previously existed, except as defined by the recreation of such a program.

Programs that attempt to reference the contents of objects declared in shared packages that have not been implicitly or explicitly initialized are technically erroneous as defined by the RM (3.2.1(18)). This implementation, however, does not prevent such references and, in fact expects them.

Since packages that contain objects that are initialized are not candidates for pragma **SHARED_PACKAGE,** the implementation suggests that programs be created for the sole purpose of initializing objects in the shared package.

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

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.

Pragma OPT_LEVEL controls the level of optimization performed by the compiler. This pragma takes one of the following as an argument: NONE, MINIMAL, GLOBAL, or MAXIMAL. The default is MINIMAL. NONE produces inefficient code but allows for faster compilation time. MINIMAL produces more efficient code with the compilation time slightly degraded. GLOBAL produces highly optimized code but the compilation time is significantly impacted. MAXIMAL is an extension of GLOBAL that can produce even better code but may change the meaning of the program. MAXIMAL attempts strength reduction optimizations that may raise OVERFLOW exceptions when dealing with values that approach the limits of the architecture of the machine. The pragma is allowed within any declarative part. The specified optimization level will apply to all code generated for the specifications and bodies associated with the immediately enclosing declarative part.

In general, programs should be developed and debugged using OPT_LEVEL (MINIMAL), reserving GLOBAL and MAXIMAL for a thoroughly tested product.

The following optimizations are performed at the various levels.

OPT_LEVEL NONE:
- Short circuit boolean tests
- Use of machine idioms
- Literal pooling

OPT_LEVEL MINIMAL: (in addition to those done with NONE)
- Binding of intermediate results to registers
- Determination of optimal execution order
- Simplification of algebraic expressions
- Re-association of expressions to collect constants
- Detection of unreachable instructions
- Elimination of jumps to adjacent labels

Elimination of jumps over jumps
Replacement of a series of simple adjacent instructions by a single faster complex instruction
Constant folding

OPT_LEVEL GLOBAL: (in addition to those done with MINIMAL)
Elimination of unreachable code
Insertion of zero trip tests
Elimination of dead code
Constant propagation
Variable propagation
Constraint propagation
Folding of control flow constructs with constant tests
Elimination of local and global common sub-expressions
Move loop invariant code out of loops
Reordering of blocks to minimize branching
Binding variables to registers
Detection of uninitialized uses of variables

OPT_LEVEL MAXIMAL: (in addition to those done with GLOBAL)
Strength reduction
Test replacement
Induction variable elimination
Elimination of dead regions

F.3 IMPLEMENTATION-DEPENDENT ATTRIBUTES

HAPSE has defined the following attributes for use in conjunction with the implementation-defined pragma SHARED_PACKAGE.

P'LOCK
P'UNLOCK

Where the prefix P denotes a package marked with pragma SHARED_PACKAGE. The 'LOCK attribute defines a function that alters the "state" of the package to the LOCK state. The function has two optional parameters and returns a BOOLEAN result that has the value TRUE if a successful LOCK operation occurred or FALSE if the package was already LOCKed. The 'UNLOCK attribute defines a function that alters the "state" of the package to the UNLOCK state. It has no parameters and returns a BOOLEAN result that has the value TRUE if a successful UNLOCK operation occurred or FALSE if the package was already UNLOCKed.

The "state" of the package is only meaningful to the 'LOCK and 'UNLOCK attribute functions that set and query the state. A LOCK state does not prevent concurrent access to objects in the shared package. These attributes only provide indivisible operations for the set and test of implicit semaphores that could be used to control access.

The first parameter of the 'LOCK attribute function must be of the base type BOOLEAN and specifies whether to put the program into a sleep state until such time as the package becomes UNLOCKed, before executing the LOCK operation. This parameter is declared with a default value of FALSE, such that no sleep will occur unless explicitly specified by the user. The sleep state is induced through the VOS $WAIT service. Note that the sleep state will not be pre-empted by the implementations time-slice for tasks. Note that even if sleep is requested, this does not guarantee that the LOCK operation will be successful when it finally is attempted.

The second parameter must be of the base type INTEGER and represents the timeout period in clock ticks, should the function be requested to sleep. This parameter defaults to zero, which represents no timeout.

F.4 SPECIFICATION OF PACKAGE SYSTEM

```
package SYSTEM is
   type ADDRESS is new INTEGER;
   type NAME is ( harris_vue);

   SYSTEM_NAME            : constant NAME := harris_vue;

   -- System-Dependent Constraints

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

   -- System-Dependent Named Numbers

   MIN_INT                : constant := -8_388_608;
   MAX_INT                : constant := 8_388_607;
   MAX_DIGITS             : constant := 9;
   MAX_MANTISSA           : constant := 37;
   FINE_DELTA             : constant := 2.0**(-37);
   TICK                   : constant := 0.01;

   -- Other System-dependent Declarations

   subtype PRIORITY is INTEGER range 0 .. 23;

   MAX_REC_SIZE           : INTEGER := 1_044_480;  -- 340*1024*3 storage_units

end SYSTEM;
```

F.5 RESTRICTIONS ON REPRESENTATION CLAUSES

F.5.1 Pragma PACK

Pragma PACK is fully supported. Objects and components are packed to the nearest and smallest bit boundary when pragma **PACK** is applied.

F.5.2 Length Clauses

The specification T'SIZE is fully supported for all scalar and composite types, except for floating point. The specification T'SIZE is not supported for access and task types.

T'SIZE applied to a composite type will cause compression of scalar component types and the gaps between the components. T'SIZE applied to a composite type whose components are composite types does not **imply** compression of the inner composite objects. To achieve such compression, the implementation requires explicit application of T'SIZE or pragma **PACK** to the inner composite type.

Composite types which contain components that have had T'SIZE applied to them, will adhere to the specified component size, even if it causes alignment of components on non STORAGE_UNIT boundaries.

The size of a non-component object of a type whose size has been adjusted, via T'SIZE or pragma **PACK**, will be exactly the specified size; however, the implementation will choose an alignment for such objects that provides optimal performance.

F.5.3 Record Representation Clauses

The simple expression following the keywords "**at mod**" in an alignment clause specifies the STORAGE_UNIT alignment restrictions for the record, and must be one of the following values: 1 or 3.

The simple expression following the keyword "**at**" in a component clause specifies the STORAGE_UNIT (relative to the beginning of the record) at which the following **range** is applicable. The static range following the keyword **range** specifies the bit range of the component. Components may overlap word boundaries (3 STORAGE_UNITs). Components that are themselves composite types must be aligned on a STORAGE_UNIT boundary.

A component clause applied to a component that is a composite type does not **imply** compression of that component. For such component types, the implementation requires that T'SIZE or pragma **PACK** be applied, if compression beyond the default size is desired.

F.5.4 Address Clauses

Address clauses are not supported.

F.5.5 Interrupts

Interrupts clauses are not supported.

F.6 OTHER REPRESENTATION IMPLEMENTATION-DEPENDENCIES

The ADDRESS attribute is not supported for the following entities: static constants, packages, tasks, labels, and entries. Application of the attribute to these entities generated a compile time warning and a value of 0 at runtime.

F.7 CONVENTIONS FOR IMPLEMENTATION-GENERATED NAMES

There are no implementation generated names.

F.8 UNCHECKED CONVERSIONS

F.8.1 Restrictions

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.

F.9 IMPLEMENTATION CHARACTERISTICS OF I/O PACKAGES

F.9.1 Implementation-Dependent Characteristics Of DIRECT I/O

Instantiations of DIRECT_IO use the value MAX_ELEM_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_ELEM_SIZE is used instead. MAX_ELEM_SIZE is defined in package DIRECT_IO as ((64 * 1024 * 3) -1) or 196_607 storage units.

F.9.2 Implementation-Dependent Characteristics Of SEQUENTIAL I/O

Instantiations of SEQUENTIAL_IO use the value MAX_ELEM_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_ELEM_SIZE is used instead. MAX_ELEM_SIZE is defined in package SEQUENTIAL_IO as ((64 * 1024 * 3) -1) or 196_607 storage units.

F.9.3 Interpretation of Strings as Applied to External Files

Strings that contain names of external files are interpreted in the following manner for each of the respective external file environments.

VUE external files: filenames can be composed of up to 512 characters of the ASCII character set except for "/", ascii.nul, and non-printable characters. Further, the first character of a file must be alphanumeric, "." or "_". If the "/" character is encountered in a string, it is interpreted as a separator between filenames that specify VUE directories.

VOS external files: filenames are composed of a one to eight character qualifier plus a one to eight character areaname. The first character of the areaname must be alphabetic. The remaining characters comprising the areaname may be drawn from the following set of characters: A-Z, 0-9, :, #, -, /, . and " ". The qualifier portion of a filename is optional. If specified, it must be comprised of an account portion, a name portion, and an asterisk. The account portion can be null, or one to four characters from the following set: 0-9. The name portion can be null, or one to four characters from the following set: A-Z, 0-9. The name portion cannot be null if the account portion is not null. If lowercase letters are encountered in the string they are converted to uppercase.

F.9.4 Interpretation of Strings as Applied to Form Parameters

The OPEN and CREATE I/O procedures accept FORM parameters, in order to specify implementation dependent attributes of files. The HAPSE implementation supports the attributes described below. These attributes may be specified in any order. Blanks may be inserted between attributes, however none are required. No attribute can be specified more than once. All attributes must be specified in uppercase. These attributes are only applicable to CREATE calls. A form string passed to OPEN is ignored.

File Type Attributes

BL Blocked file
UB Unblocked file
RA Random file
BS Byte stream file

These attributes specify the VOS file type of a file to be created. UB is the default for all files. In general, the defaults should not be overridden for direct and sequential I/O.

Double Buffered Blocking

DB Defines a BL type file as permanently double buffered

This attribute can only be specified if the file type is BL.

Directory Type

CD The VOS directory entry for this file is to be kept resident
DD The VOS directory entry is kept on disc

Access Parameters

PR PUBLIC READ
PW PUBLIC WRITE
PD PUBLIC DELETE
AR ACCOUNT READ

AW	ACCOUNT WRITE
AD	ACCOUNT DELETE
OW	OWNER WRITE
OD	OWNER DELETE

These attributes determine the access permissions associated with a file. The default access is OW OD. Note that if any access attributes are specified, then only the specified accesses will be granted (i.e., OW OD is not assumed).

File Definition Attributes

A=n	Access level, n = 0-15, VOS access required to access file
B=n	Blocking factor, where n is 1-7 sectors
C=n	Current size, where n is the number of sectors requested
E=n	Eliminate date, where n is the number of days before purging
G=n	Granule size, where n is the number of sectors per granule
M=n	Maximum size, n = number of sectors to which file may expand
P=n	Pack number, n = pack number of pack on which to create file
T=n	Type number, n = 0-7, provided for user file classification

No spaces are allowed between the attribute letter, the equal sign, and the integer value.

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
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	24
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	(1..498 => 'A', 499 => '1')
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	(1..498 => 'A', 499 => '2')
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	(1..249 => 'A', 250 => '3', 251..499 => 'A')

TEST PARAMETERS

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	(1..249 => 'A', 250 => '4', 251..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")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..493 => '0', 494..499 => "69.0e1")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1..249 => 'A')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1..249 => 'A', 250 => '1')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..479 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	8388607
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	6291456
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	HARRIS_VUE
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2.0**(-37)
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	8388607
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	(NO_SUCH_FIXED_TYPE)
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	(NO_SUCH_FLOAT_TYPE)
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10_000_000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	23
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	/no/such/directory/illegal_name1
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/no/such/directory/illegal_name2
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-8388608

TEST PARAMETERS

Name and Meaning	Value
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	8388607
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	8388608
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10_000_000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	37
$MAX_DIGITS Maximum digits supported for floating-point types.	9
$MAX_IN_LEN Maximum input line length permitted by the implementation.	499
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	8388607
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	8388608
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 => "2:",3..496 => '0', 497..499 => "11:")

Name and Meaning	Value
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 => "16:",4..495 => '0', 496..499 => "F.E:")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '"', 2..498 => 'A', 499 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-8388608
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	24
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_INTEGER_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	HARRIS_VUE
$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#FFFFFF#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	6291456

Name and Meaning	Value
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	HARRIS_VUE
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	24
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

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

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

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

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

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

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

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

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These

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

h. CD2A81G, CD2A83G, CD2A84M..N, and CD50110 (5 tests): These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86, 96, and 58, respectively).

i. CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B: This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C..D, and ED7006C..D (5 tests): These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).

q. CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, and 136).

s. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.