AD-A210 112

(When Data Entered)

[REPORT DOCUM]ENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER / 2. GOVT ACCESSION NO	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Harris Corporation, Computer Systems Division, Harris Ada, Version 5.0, Harris HCX-9 (Host) to Harris NH-3800 (Target), 890118W1.10017	5. TYPE OF REPORT & PERIOD COVERED 17 Jan 1989 to 17 Jan 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.

DTIC ELECTE JUN 30 1989 S E D

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)

Harris Corporation, Computer Systems Division, Harris Ada, Version 5.0, Wright-Patterson AFB, Harris HCX-9 under CX/UX, Version 4.0 (Host) to Harris NH-3800 under CX/UX, Version 4.0 (Target), ACVC 1.10

89 6 29 152

Ada Compiler Validation Summary Report:

Compiler Name: Harris Ada, Version 5.0

Certificate Number: 890118W1.10017

Host: Harris HCX-9 under CX/UX, Version 4.0

Target: Harris NH-3800 under CX/UX, Version 4.0

Testing Completed 17 January 1989 Using ACVC 1.10

This report has been reviewed and is approved.

for

Ada Validation Facility
Steven 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

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

DTIC COPY INSPECTED 6

AVF Control Number: AVF-VSR-216.0589
88-08-25-HAR

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890118W1.10017
Harris Corporation, Computer Systems Division
Harris Ada, Version 5.0
Harris HCX-9 Host and Harris NH-3800 Target

Completion of On-Site Testing:
17 January 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: 890118W1.10017

Host: Harris HCX-9 under CX/UX, Version 4.0

Target: Harris NH-3800 under CX/UX, Version 4.0

Testing Completed 17 January 1989 Using ACVC 1.10

This report has been reviewed and is approved.

Ada Validation Facility
Steven 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

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

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

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

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

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

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

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by 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 17 January 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

Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

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

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

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

1.4 DEFINITION OF TERMS

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

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

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

Applicant — The agency requesting validation.

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

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

Compiler — A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

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

Host — The computer on which the compiler resides.

Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer which executes the code generated by the compiler.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn 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 errors because of the way in which a program library is used at link time.

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

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

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

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

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

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

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

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

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

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

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

Compiler: Harris Ada, Version 5.0

ACVC Version: 1.10

Certificate Number: 890118W1.10017

Host Computer:

Machine:	Harris HCX-9
Operating System:	CX/UX, Version 4.0
Memory Size:	32 Megabytes

Target Computer:

Machine:	Harris NH-3800
Operating System:	CX/UX, Version 4.0
Memory Size:	8 Megabytes

Communications Network: ETHERNET

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) This implementation supports the additional predefined types SHORT_INTEGER, TINY_INTEGER, and LONG_FLOAT in the 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) 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 comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

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 is round to even. (See tests C46012A..Z (26 tests).)

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

(3) The method used for rounding to integer in static universal real expressions is round to even. (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 an array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

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

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises 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 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, the test results indicate that 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 before 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 or procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

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.)

(3) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

(4) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

(5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

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

(7) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

(8) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

(9) 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) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

(4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

(5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

(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 (2 tests), 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 can be associated with each external file for sequential files when writing or reading. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

(14) More than one internal file can 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 can be associated with each external file for text files when writing or reading. (See tests CE3111A..E (5 tests), 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, 36 tests had been withdrawn because of test errors. The AVF determined that 465 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 14 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	1132	1857	17	33	46	3216
Inapplicable	0	6	460	0	1	0	465
Withdrawn	1	2	33	0	0	0	36
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	193	547	489	245	172	99	161	332	137	36	252	254	297	3216
N/A	20	102	191	3	0	0	5	1	0	0	0	121	24	465
Wdrn	0	1	0	0	0	0	0	1	0	0	1	29	4	36
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 36 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

A39005G	B97102E	BC3009B	CD2A62D	CD2A63A..D	CD2A66A..D
CD2A73A..D	CD2A76A..D	CD2A81G	CD2A83G	CD2A84M..N	CD50110
CD2B15C	CD7205C	CD5007B	CD7105A	CD7203B	CD7204B
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, 465 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 (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)

C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

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

c. 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				

d. C45531I..P (8 tests) and C45532I..P (8 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 32.

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

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

g. 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.

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

i. This implementation does not suppport address clauses. Therefore, the following 76 tests are not applicable:

CD5003B	CD5003C	CD5003D	CD5003E	CD5003F
CD5003G	CD5003H	CD5003I	CD5011A	CD5011B
CD5011C	CD5011D	CD5001E	CD5011F	CD5011G
CD5011H	CD5011I	CD5011K	CD5011L	CD5011M
CD5011N	CD5011Q	CD5011R	CD5011S	CD5012A
CD5012B	CD5012C	CD5012D	CD5012E	CD5012F
CD5012G	CD5012H	CD5012I	CD5012J	CD5012L
CD5012M	CD5013A	CD5013B	CD5013C	CD5013D
CD5013E	CD5013F	CD5013G	CD5013H	CD5013I
CD5013K	CD5013L	CD5013M	CD5013N	CD5013O
CD5013R	CD5013S	CD5014A	CD5014B	CD5014C
CD5014D	CD5014E	CD5014F	CD5014G	CD5014H
CD5014I	CD5014J	CD5014K	CD5014L	CD5014M
CD5014N	CD5014O	CD5014R	CD5014S	CD5014T
CD5014U	CD5014V	CD5014W	CD5014X	CD5014Y

CD5014Z

j. This implementation does not support intermediate 'SIZE clauses for floating point types. Therefore, the following 14 tests are not applicable:

CD1009C   CD2A41A..B   CD2A41E   CD2A42A..J

k. This implementation does not support 'SIZE clauses for access types. Therefore, the following 23 tests are not applicable:

CD2A81A..F   CD2A83A..C   CD2A83E..F   CD2A84B..I
CD2A84K..L   CD2A87A   ED2A86A

l. This implementation does not support 'SIZE clauses for task types. Therefore, the following 5 tests are not applicable:

CD2A91A..E

m. CD2A61I and CD2A61J are not applicable because they specify length clauses that require the implementation to compress the storage allocated to the components of an aggregate object. This compression is not required by the Ada Reference Manual (13.2(5)).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ac. CE3102E is inapplicable because text file CREATE with IN_FILE mode is supported by this implementation.

ad. CE3102F is inapplicable because text file RESET is supported by this implementation.

ae. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.

af. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.

ag. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

ah. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is supported by this implementation.

ai. CE3115A is inapplicable because resetting an external file with OUT_FILE mode is not supported when the internal file is associated with more than one external file. USE_ERROR is raised for such an attempt.

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 was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 14 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 modification was made to compensate for legitimate implementation behavior:

a. At the recommendation of the AVO, the line containing INTEGER'IMAGE was commented out in test ED7006C since SYSTEM.MEMORY_SIZE is outside the range of INTEGER for this implementation and there is no predefined integer type whose range includes SYSTEM.MEMORY_SIZE.

The following tests were graded using a modified evaluation criteria:

a. In test CE3804G, 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 floating 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.

b. 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, Version 5.0 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Harris Ada, Version 5.0 compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	Harris HCX-9
Host operating system:	CX/UX, Version 4.0
Target computer:	Harris NH-3800
Target operating system:	CX/UX, Version 4.0
Compiler:	Harris Ada, Version 5.0

The host and target computers were linked via ETHERNET.

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 and linked on the Harris HCX-9, then all executable images were transferred to the Harris NH-3800 via ETHERNET and run. 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 occurred 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 host computer and a single target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Ft. Lauderdale FL and was completed on 17 January 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Harris Corporation, Computer Systems Division has submitted the following Declaration of Conformance concerning the Harris Ada, Version 5.0 compiler.

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 HCX-9	OS&VER #: CX/UX, 4.0
Target Architecture ISA: Harris NH-3800	OS&VER #: CX/UX, 4.0

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.

Wendell Norton Date: 11-7-88

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: 11-7-88

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 compiler, 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 -2_147_483_648 .. 2_147_483_647;
     type SHORT_INTEGER is range -32_768 .. 32_767;
     type TINY_INTEGER is range -128 .. 127;

     type FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;
     type LONG_FLOAT is digits 9 range -1.70141183E+38 .. 1.70141183E+38;

     type DURATION is delta 2.0**(-13) range -131_072.0 .. 131_072.0;

   ...

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 recognised 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 recognised 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**.

For FORTRAN, the implementation also detects usage of this pragma at link time see (a.ld) and includes a call to the system supplied FORTRAN initialisation routine as part of the elaboration of the Ada program. Additionally, the default system FORTRAN libraries are included in the linking of the Ada program.

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

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

Pragma OPTIMIZE is recognised by the implementation but has no effect in this release. See the -O option for **ada** for code optimisation 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 GCX range from 0 to 9, with 9 being the most urgent.

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

Pragma STORAGE_UNIT is recognised 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. DIVISION_CHECK and OVERFLOW_CHECK for floating point types will reduce the amount of overhead associated with checking, but is not fully repressible. The double parameter form of the pragma, with a name of an object, type, or subtype is recognised, but has no effect.

Pragma SYSTEM_NAME is recognised 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_SHARED_OBJECT provides an interface to objects defined in foreign languages which exist in CX/UX shared memory segments. Specifically, this allows for the sharing of data between Ada objects and FORTRAN or C objects defined within the same process or in a separate process.

Pragma INTERFACE_SHARED_OBJECT associates an Ada variable with a CX/UX shared memory segment. It has two required parameters. The first parameter is the simple name of the Ada variable to be associated with the foreign object. The second parameter is a string constant that defines the external link name of the object as defined in the foreign language. The variable declaration must occur before the pragma and both must occur within the same declarative part or package specification.

Variables marked with the pragma must have a static size. It is recommended that an explicit length clause be specified for composite objects to ensure conformance with the size as defined by the foreign language. Additionally, record representation clauses may be used to define the layout of records to match the foreign language definitions.

The association of the shared memory segment to the Ada variable is effected at program startup time, by the HAPSE run-time system. However, specific control over the configuration of the shared memory is defined externally from the Ada compilation model and requires user intervention. The CX/UX *shmdefine* utility has been provided to aid the user in defining the configuration of shared memory segments. The utility produces a link-ready file and a loader command file which must be included in the link of any Ada program using pragma **INTERFACE_SHARED_OBJECT.** To include these files in the link process, the user should invoke the HAPSE prelinker, *a.ld* , adding the names of these files to the end of the command line. See section 11.2.2 for an example application of the pragma. Refer to the *CX/UX User's Reference Manual* for details on the *shmdefine* utility.

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 shared memory 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 shared memory 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, "**params**", that, if specified, must be a string constant containing a comma separated list of CX/UX shared segment configuration parameters, as defined by the following:

- key= *name,* which identifies the CX/UX shared segment key to be used in subsequent *shmget* system calls, which are done automatically by the implementation in configuring the shared segment. *name* is considered to be a CX/UX filename which will be translated to a shared segment key using the CX/UX *ftok(3C)* service. By default, HAPSE applies "key= *{absolute HAPSE library path}/.shmem/package_name* to the shared package. Note that relative path names may be specified and would cause key translation to be dependent on the user's current working directory when program execution is initiated. If *name* is a decimal integer literal, HAPSE interprets this as the actual CX/UX key, and does not translate it using the *ftok* service.

- ipc= (IPC_CREAT, IPC_EXCL, IPC_PRIVATE), which allows the user to specify details about the initialization of the shared segment. By default, HAPSE applies ipc= (IPC_CREAT) to the shared package, thereby creating the shared segment if it did not previously exist. If any ipc parameters are given, they entirely replace the default ipc specification.

- **SHM_RDONLY,** which specifies that the segment is only available for READ operations. HAPSE defaults shared package segments to READ/WRITE.

- **mode** = *n*, where *n* is assumed to be a 3 digit octal number defining the access to the shared segment. By default, HAPSE applies **mode=644** to the shared package, (owner read/write, group read, other read).

A detailed explanation of the IPC and SHM flags, and access modes may be found in the *CX/UX Programmer's Reference Manual, Chapter 2.*

The pragma must appear within the specification of the library level package. The pragma may also be repeated in the package body to allow the user to override the shared memory configuration parameters that were associated with the pragma in the specification. Additionally, these configuration parameters, as defined above, may also be specified at link time to **a.ld**, via the -shmem "**params**" option, where "**params**" is defined as above with the addition that the first item in the list must be the name of a shared package. If this option is used, then it replaces all previous information that may have been provided with all pragmas for that package.

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 lifetime of such objects is guaranteed to extend from the elaboration of the shared package by the first *concurrent program* until the termination of execution of the last *concurrent program.*

In the assumptions above, a *concurrent program* is defined to be any Ada program which elaborates the body of a shared package, whose span of execution, from elaboration of such a package to termination, overlaps that of another such program.

In actuality, the shared memory segments created by these programs remain even after the last *concurrent program* has exited. The values of objects within these segments remains valid until the segment is destroyed, or until the system is rebooted. Segments may be explicitly removed through the shared memory service *shmctl,* to which an interface is provided in the HAPSE package *shared_memory_support.* Alternatively, the user may obtain information about active shared memory segments through the CX/UX utility *ipcs(3).* These segments may be removed via the CX/UX utility *iprm(1).*

Programs that attempt to reference the contents of objects declared in shared packages that have not been implicitly or explicitly initialised 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.

The above discussion describes the intent that several Ada programs may begin, continue, and complete their execution simultaneously, with the contents of the variables in the shared packages consistent with the execution of those programs.

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

The association of a CX/UX shared memory segment with the shared package is effected during the elaboration of the package body. If this association should fail due to system shared memory constraints, access, or improper use of shared memory configuration parameters, one of several predefined exceptions will be raised. The exceptions are of the form:

shared_package_error.{name of package}.{service}.{code}

where .{code} is a CX/UX error code mnemonic.

For example, shared_package_error.package.shmat.EMFILE would be raised to indicate that the shared package attachment failed because it would exceed the system imposed limit on active shared segments. These exceptions are not available to the user since exceptions generated from the elaboration of library level package bodies have no enclosing scope from which to supply a handler. Refer to the *CX/UX Programmer's Reference Manual* for a detailed list of the error conditions for *shmget(2)* and *shmop(2).*

So that programs can define critical sections to reference and update variables within the shared packages, HAPSE has provided semaphore operations. See the description of the implementation-defined attributes P'LOCK and P'UNLOCK.

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 sise.

Pragma OPT_LEVEL controls the level of optimisation 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 optimised 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 optimisations 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 optimisation level will apply to all code generated for the specifications and bodies associated with the immediately enclosing clarative part.

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

The following optimisations 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 sero 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 minimise branching
- Binding variables to registers
- Detection of uninitialised 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'KEY
P'LOCK
P'UNLOCK

Where the prefix P denotes a package marked with pragma **SHARED_PACKAGE.**

The 'KEY attribute is an overloaded parameterless function which returns the key used to identify the CX/UX shared segment associated with the package. One specification of the function returns the predefined type *string,* and returns a value specifying the filename used in the key translation *(ftok(3C)).* If an integer literal key was specified in the pragma **shared_package** parameters, this function returns a null string. The other specification of the function returns the predefined type *universal_integer,* and returns a value specifying the translated integer key. The latter form of the function will raise the predefined exception *PROGRAM_ERROR* if the shared package body has not yet been elaborated.

The 'LOCK and 'UNLOCK attributes are parameterless procedures which manipulate the "state" of a shared package. HAPSE defines all shared packages to have two states: LOCKed and UNLOCKed. Upon return from the 'LOCK procedure, the state of the package will be LOCKed. If upon invocation, 'LOCK finds the state already LOCKed, it will wait until it becomes UNLOCKed before altering the state and returning. 'UNLOCK sets the state of the package to UNLOCKed and then returns. At the point of unlocking the package, if another process waiting in the 'LOCK procedure has a more favorable CX/UX priority, the system will immediately schedule its execution.

Note that if 'LOCK is waiting, it may be interrupted by the HAPSE run-time system's time slice for tasks which may cause another task within the process to become active. Eventually, HAPSE will again transfer control to the 'LOCK procedure in the original task, and it will continue waiting or return to the task.

The state of the package is only meaningful to the 'LOCK and 'UNLOCK attribute procedures 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 setting and testing of implicit semaphores that could be used to control access to shared package objects.

HAPSE provides the package, **shared_memory_support.** This package contains Ada type, subprogram definitions, and interfaces to aid the user in manually interfacing to the CX/UX shared memory services.

This includes:

- System *defines* and records layouts as defined by the CX/UX C Programming Language include files, <*sys/shm.h*> and <*sys/ipc.h*>.
- Interface specifications to shared memory system calls: *shmbind, shmget shmat, shmctl, shmdt.*
- Interface specifications to the CX/UX binary semaphore operators: *binsemget, lockbinsem, unlockbinsem.*

F.4 SPECIFICATION OF PACKAGE SYSTEM

```
package SYSTEM is
   type ADDRESS is private;
   type NAME is ( Harris_GCX);

   SYSTEM_NAME          : constant NAME := Harris_GCX;

   -- System-Dependent Constraints

   STORAGE_UNIT         : constant := 8;
   MEMORY_SIZE          : constant := 3_221_225_469;

   -- System-Dependent Named Numbers

   MIN_INT              : constant := -2_147_483_648;
   MAX_INT              : constant := 2_147_483_647;
   MAX_DIGITS           : constant := <9 for HCX/GCX, 15 for GCX/GCX>;
   MAX_MANTISSA         : constant := 30;
   FINE_DELTA           : constant := 2.0**(-30);
   TICK                 : constant := 0.01;

   -- Other System-dependent Declarations

   subtype PRIORITY is INTEGER range 0 .. 9;

   MAX_REC_SIZE         : INTEGER := 4_000_000;

   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;
```

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: 1,2 or 4.

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 (4 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 only supported for the task entries.

The function PHYSICAL_ADDRESS is defined in the package SYSTEM to provide conversion from INTEGER values to ADDRESS values.

F.5.5 Interrupts

Interrupt entries (UNIX signals) are supported. This feature allows Ada programs to bind a UNIX signal to an interrupt entry by using a for clause with a signal number. There is no protection against two tasks binding the same signal. The result is undefined. Interrupt entries should have no parameters and can be called explicitly by the program. See SIGVEC(2).

The HAPSE runtime uses SIGALRM (14) to perform time slicing and delays. The result of establishing a signal handler for SIGALRM is undefined.

The following example program uses an interrupt entry that prints a message when the process receives SIGINT.

```
with TEXT_IO, SYSTEM;
use TEXT_IO;
procedure INTR is
-- This program waits for the user to generate SIGINT (<CONTROL>C)

SIGINT_NUMBER : constant := 2;

task SIGINT_HANDLER is
  entry SIGINT;
  for SIGINT use at SYSTEM.PHYSICAL_ADDRESS(SIGINT_NUMBER);
end SIGINT_HANDLER;
task body SIGINT_HANDLER is
  begin
    accept SIGINT;
    PUT_LINE("Control-C received");
  end SIGINT_HANDLER;

begin
  null;
end INTR;
```

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 record type with discriminants.

F.8.2 Implementation

The following describes the transfer of data between the source and target operands when performing unchecked conversion. When possible, the implementation may optimise the conversion operation such that no transfer of data actually occurs.

F.8.2.1 Simple Types

For all access, task and scalar types, unchecked conversion is implemented using the most efficient MOVE instruction to move a 1, 2, 4 or 8 byte object to its destination.

If the sizes of the source and target differ, then the smallest size is used.

If the target has a larger size than the source, the source is moved to the low order bits of the target with no change in bit pattern. The high order bits of the target are zero filled if the source had an unsigned representation, else the high bit of the source is signed extended through the high bits of the target.

If the target has a smaller size than the source, the low order bits of the source are copied to the target.

F.8.2.2 Composite Types

All conversions logically occur by moving bits from the source to the target, starting at the highest order bit of the source and target. The size and shape of the target object is not changed, even if the size/shape of the source is different. When performing the move the smaller of the source and target sizes is used as the amount of data to move.

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_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 by a program before instantiating DIRECT_IO to provide an upper limit on the record size. In any case, the maximum size supported is 268_435_455 storage units. DIRECT_IO raises USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

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

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 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 by a program before instantiating SEQUENTIAL_IO to provide an upper limit on the record size. In any case, the maximum size supported is 268_435_455 storage units. SEQUENTIAL_IO raises USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

F.10 MACHINE CODE INSERTIONS

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 mnemonics, a set of register definitions, and a set of addressing mode functions. Also supplied (fo· use only in units that WITH MACHINE_CODE) is implementation-defined 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});

In the example shown below, CODE_2 is a record 'format' whose first argument is an enumeration value of type OPCODE followed by two operands of type OPERAND.

CODE_2'(MOVE_L, a'ref, b'ref);

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

For an object, 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.

The 'REF attribute denotes the effective address of the first of the storage units allocated to the object. For a label, it refers to the address of the machine code associated with the corresponding body or statement. The attribute is of type OPERAND defined in package MACHINE_CODE and is allowed only within a machine code procedure. 'REF is only supported for simple objects and labels..

Registers - The supported register operands are D0, D1, ..., D7, A0, A1, ..., A7, SP (which is A7) and FP (which is A6).

The general syntax for a machine data statement is:

DATA_n'(size, operand {, operand });

In the following example, DATA_1 is a record 'format' where the first argument is the size of the data (byte, word or long) followed by the data.

Data_1'(long, a 'ref);

The size is an enumeration literal. An operand can only be an entity defined in MACHINE_CODE or the 'REF attribute.

Addressing Modes - All of the GCX's addressing modes are supported by the compiler, except for some PC relative modes. They are accessed through the following functions provided in MACHINE_CODE.

Address Mode	Assembler Notation	Ada Function Call
Data Register Direct	dn (n=0,1,...,7)	dn (n in 0...7)
Address Register Direct	an (n=0,1,...,7)	an (n in 0...7)
Address Register Indirect	(An)	indr(<addr_reg>)
with Postincrement	(An)+	incr(<addr_reg>)
with Predecrement	-(An)+	decr(<addr_reg>)
with Displacement	(d,An)	disp(<reg>, <disp>)
with Index (8-bit displacement)	(d,An,Xn.L*SCALE)	index(<base_reg>, <disp>, <index_reg>, <scale_factor>)
Memory Indirect Preindexed	([bd,An,Xn.L*SCALE],od)	index_pre(<base_reg>, <disp>, <index_reg>, <scale_factor>, <outer_disp>)
Memory Indirect Postindexed	([bd,An],Xn.L*SCALE,od)	index_post(<base_reg>, <disp>, <index_reg>, <scale_factor>, <outer_disp>)
Absolute Short Address	(xxx).W	Absol(<disp>)
Absolute Long Address	(xxx).L	Absol(<disp>)
Immediate Data	#xxx #xxx #xxx	immed(<integer>) immed(<float>) immed(<character>)
External Name	$<name> +#xxx +#xxx -#xxx -#xxx <operand>/<operand>	ext(<name>) +(<integer>) +(<float>) -(<integer>) -(<float>) <operand>/<operand>

The following example uses machine code to move a block of data.

```
with machine_code;
with system;

  procedure move(src,dest : in system.address; length : in positive) is
  --
     use machine_code;
  --
  begin
  --
  code_2' (movea_l, src'ref, a0);
  code_2' (movea_l, dest'ref, a1);

  code_2' (move_l, length'ref, d0);
  code_2' (subq_l, +1, d0);

  <<start>>
        code_2' (move_b, incr(a0), incr(a1));
        code_2' (dbf, d0, start'ref);
  --
  end move;
```

NOTES

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.	32
$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')

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 => '"', 2..250 => 'A', 251 => '"')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1 => '"', 2..250 => 'A', 251 => '1', 252 => '"')
$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.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	3221225469
$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_GCX
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2.0**(-30)
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
$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.	9
$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.	-2147483648

TEST PARAMETERS

Name and Meaning	Value
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-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.	30
$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.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(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.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	TINY_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	HARRIS_GCX
$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#
$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.	3221225469

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_GCX
$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.	32
$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 36 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. A39005G has been withdrawn because it unreasonably expects a component clause to pack an array component into a minimum size (line 30).

b. B97102E has been withdrawn because it contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

c. BC3009B has been withdrawn because it 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).

d. CD2A62D has been withdrawn because it 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).

e. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests] have been withdrawn because they 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 sub-program (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.

f. CD2A81G, CD2A83G, CD2A84M and N, and CD50110 have been withdrawn because they 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 and 96, 86 and 96, and 58, respectively).

g. CD2B15C and CD7205C have been withdrawn because they 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.

h. CD5007B has been withdrawn because it wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

i. CD7105A has been withdrawn because it 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).

j. CD7203B and CD7204B have been withdrawn because they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

k. CD7205D has been withdrawn because it 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.

l. CE2107I has been withdrawn because it 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)

m. CE3111C has been withdrawn because it requires certain behavior when two files are associated with the same external file; however, this is not required by the Ada standard.

n. CE3301A has been withdrawn because it 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).

o. CE3411B has been withdrawn because it 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.