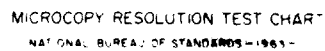


1/1

44

[illegible]



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-

AD-A190 363

ON PAGE

12. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Ada Compiler Validation Summary Report:
HEWLETT Packard. HP 9000 Series 200/300 Ada Compiler,
Rev.1.4.HP 9000 Series 200/300

5. TYPE OF REPORT & PERIOD COVERED

11 Dec 1986 to 11 Dec 1987

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Wright-Patterson AFB

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS

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

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE

11 December 1986

13. NUMBER OF PAGES

54

14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office)

Wright-Patterson

15. SECURITY CLASS (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

DTIC
ELECTE
JAN 06 1988
S D

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

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

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

See Attached

DD

FORM

1473

EDITION OF 1 NOV 65 IS OBSOLETE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)

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the HP 9000 Series 200/300 Ada Compiler (Ada/300), Revision 1.4, using Version 1.8 of the Ada[®] Compiler Validation Capability (ACVC). The Ada/300 was tested on the following five configurations:

- . HP 9000 Series 200 Model 220 under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 310, under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 320 under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 330 under HP-UX, Revision 5.19
- . HP 9000 Series 300 Model 350 under HP-UX, Revision 5.19

On-site testing was performed 8 December 1986 through 11 December 1986 at Hewlett Packard in Cupertino CA, under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2102 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 278 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2102 tests were processed, results for Class A, C, D, or E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 21 of the processed tests determined to be inapplicable. The remaining 2081 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	93	206	280	246	161	97	134	262	121	32	217	232	2081
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	23	119	140	1	0	0	5	0	9	0	1	1	299
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

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

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

AVF Control Number: AVF-VSR-50.0687
86-10-J1-HPC

Ada[®] COMPILER
VALIDATION SUMMARY REPORT:
Hewlett Packard
HP 9000 Series 200/300 Ada Compiler,
Revision 1.4
HP 9000 Series 200/300

Completion of On-Site Testing:
11 December 1986

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

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

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution	
Availability Codes	
Dist	Availability Codes
A-1	



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

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

Ada[®] Compiler Validation Summary Report:

Compiler Name: HP 9000 Series 200/300 Ada Compiler, Revision 1.4

Hosts and Targets:

- . HP 9000 Series 200 Model 220 under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 310 under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 320 under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 330 under HP-UX, Revision 5.19
- . HP 9000 Series 300 Model 350 under HP-UX, Revision 5.19

Testing Completed 11 December 1986 Using ACVC 1.8

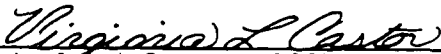
This report has been reviewed and is approved.



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



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



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

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

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the HP 9000 Series 200/300 Ada Compiler (Ada/300), Revision 1.4, using Version 1.8 of the Ada[®] Compiler Validation Capability (ACVC). The Ada/300 was tested on the following five configurations:

- . HP 9000 Series 200 Model 220 under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 310 under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 320 under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 330 under HP-UX, Revision 5.19
- . HP 9000 Series 300 Model 350 under HP-UX, Revision 5.19

On-site testing was performed 8 December 1986 through 11 December 1986 at Hewlett Packard in Cupertino CA, under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2102 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 278 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2102 tests were processed, results for Class A, C, D, or E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 21 of the processed tests determined to be inapplicable. The remaining 2081 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	93	206	280	246	161	97	134	262	121	32	217	232	2081
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	23	119	140	1	0	0	5	0	9	0	1	1	299
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

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

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

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	2-1
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	SPLIT TESTS	3-3
3.7	ADDITIONAL TESTING INFORMATION	3-4
3.7.1	Prevalidation	3-4
3.7.2	Test Method	3-4
3.7.3	Test Site	3-5
APPENDIX A	COMPLIANCE STATEMENT	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	

CHAPTER 1

INTRODUCTION

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

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

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

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

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

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

Testing of this compiler was conducted by SofTech, Inc., under the direction of the AVF according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was conducted from 8 December 1986 through 11 December 1986 at Hewlett Packard in Cupertino CA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

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

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1301 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, FEB 1983.
2. Ada Validation Organization: Procedures and Guidelines, Ada Joint Program Office, 1 JAN 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., DEC 1984.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. A set of programs that evaluates the conformity of a compiler to the Ada language specification, ANSI/MIL-STD-1815A.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established policies and procedures.
AVO	The Ada Validation Organization. In the context of this report, the AVO is responsible for setting procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	A test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

INTRODUCTION

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

Passed test A test for which a compiler generates the expected result.

Target The computer for which a compiler generates code.

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

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

1.5 ACVC TEST CLASSES

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

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective has been met. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

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

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

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers

permitted in a compilation on the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

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

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

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

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

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

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

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

Compiler: HP 9000 Series 200/300 Ada Compiler, Revision 1.4

ACVC Version: 1.8

Certificate Expiration Date: 10 March 1988

Host and Target Computers:

<u>Machine</u>	<u>Operating System</u>	<u>Memory Size</u>
HP 9000 Series 200 Model 220	HP-UX, Revision 5.141	7.5 megabytes
HP 9000 Series 300 Model 310	HP-UX, Revision 5.141	7.0 megabytes
HP 9000 Series 300 Model 320	HP-UX, Revision 5.141	7.0 megabytes
HP 9000 Series 300 Model 330	HP-UX, Revision 5.19	4.0 megabytes
HP 9000 Series 300 Model 350	HP-UX, Revision 5.19	4.0 megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

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

. Capacities.

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

. Universal integer calculations.

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

. Predefined types.

This implementation supports the additional predefined types `SHORT_INTEGER` and `LONG_INTEGER` in the package `STANDARD`. (See tests B86001C and B86001D.)

. Based literals.

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

. Array types.

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

A packed `BOOLEAN` array having a `'LENGTH` exceeding `INTEGER'LAST` does not raise an exception. (See test C52103X.)

A packed two-dimensional `BOOLEAN` array with more than `INTEGER'LAST` components raises `CONSTRAINT_ERROR` when the length of a dimension is calculated and exceeds `INTEGER'LAST`. (See test C52104Y.)

A null array with one dimension of length greater than `INTEGER'LAST` may raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation accepts the declaration.

(See test E52103Y.)

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

. Discriminated types.

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

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

. Aggregates.

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

In the evaluation of an aggregate containing subaggregates, all choices are not evaluated before being checked for identical bounds. (See test E43212B.)

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

. Functions.

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

. Representation clauses.

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

. Pragmas.

The pragma INLINE is not supported for procedures or functions. (See tests CA3004E and CA3004F.)

. Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants. The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. However, any call to OPEN or CREATE of such instances of DIRECT_IO with these types raises an exception. (See tests AE2101C, AE2101H, CE2201D, CE2201E, and CE2401D.)

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

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

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

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

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

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

. Generics.

Body and subunits of a generic unit must be in the same compilation as the specification if instantiations precede them. (See tests CA2009C, CA2009F, and BC3205D.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.8 of the ACVC contains 2399 tests. When validation testing of the HP 9000 Series 200/300 Ada Compiler (Ada/300), Revision 1.4, was performed, 19 tests had been withdrawn. The remaining 2380 tests were potentially applicable to this validation. The AVF determined that 299 tests were inapplicable to this implementation, and that the 2081 applicable tests were passed by the implementation.

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

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	69	864	1076	17	11	44	2081
Failed	0	0	0	0	0	0	0
Inapplicable	0	3	292	0	2	2	299
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

TEST INFORMATION

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14		
Passed	93	206	280	246	161	97	134	262	121	32	217	232	2081	
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	
Inapplicable	23	119	140	1	0	0	5	0	9	0	1	1	299	
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19	
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399	

3.4 WITHDRAWN TESTS

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

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

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

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. For this validation attempt, 299 tests were inapplicable for the reasons indicated:

- . C34001F and C35702A use SHORT_FLOAT which is not supported by this compiler.
- . C34001G and C35702B use LONG_FLOAT which is not supported by this compiler.
- . C55B16A makes use of an enumeration representation clause containing noncontiguous values which is not supported by this compiler.

- . B8b001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.
 - . C8b001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.
 - . C87B62A..C (3 tests) use length clauses which are not supported by this compiler. The length clause is rejected during compilation.
 - . BA2001E requires that duplicate names of subunits with a common ancestor be detected at compile time. This implementation detects the error at link time, and the AVO ruled that this behavior is acceptable.
 - . CA2009C, CA2009F, and BC3205D compile the body and subunits of a generic unit in separate compilation files. For this implementation, the body and subunits of a generic unit must be in the same compilation as the specification if instantiations precede them.
 - . CA3004E, EA3004C, and LA3004A use INLINE pragma for procedures which is not supported by this compiler.
 - . CA3004F, EA3004D, and LA3004B use INLINE pragma for functions which is not supported by this compiler.
 - . CE2401D uses an instantiation of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. In this implementation, the exception USE_ERROR is raised upon a call to OPEN or CREATE of such instances.
 - . The following 278 tests require a floating-point accuracy that exceeds the maximum of 6 supported by the implementation:
- C24113C..Y (23 tests) C35708C..Y (23 tests) C45421C..Y (23 tests)
 C35705C..Y (23 tests) C35802C..Y (23 tests) C45424C..Y (23 tests)
 C35706C..Y (23 tests) C45241C..Y (23 tests) C45521C..Z (24 tests)
 C35707C..Y (23 tests) C45321C..Y (23 tests) C45621C..Z (24 tests)

3.6 SPLIT TESTS

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

TEST INFORMATION

because of its size is split into a set of smaller subtests that can be processed.

Splits were required for 15 Class B tests.

B32202A	B43201D	B91004A
B32202B	B61012A	B95069A
B32202C	B62001B	B95069B
B33001A	B74401F	BA1101B
B37004A	B74401R	BC3205C

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.8 produced by the Ada/300 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 Ada/300 using ACVC Version 1.8 was conducted on-site by a validation team from the AVF. The configuration consisted of an HP 9000 Series 300 Model 320 operating under HP-UX, Version 5.141. Two identical computers were used in testing the compiler. The following four configurations were also tested using a subset of the ACVC:

- . HP 9000 Series 200 Model 220 under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 310 under HP-UX, Revision 5.141
- . HP 9000 Series 300 Model 330 under HP-UX, Revision 5.19
- . HP 9000 Series 300 Model 350 under HP-UX, Revision 5.19

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring splits during the prevalidation testing were included in their split form on the magnetic tape.

The contents of the magnetic tape were loaded onto an HP 9000 Series 200. The tests were accessed from the Series 200 Model 220 computer via the local area network. The full set of tests was compiled, linked, and executed as appropriate on the two Series 300 Model 320 computers. Results were transferred over the network to either a Series 300 Model 350 or a Series 500 Model 530 and printed.

A subset of the ACVC, Version 1.8, was run on a Series 200 Model 220 and a Series 300 Model 310, Model 330, and Model 350. The subset of sixty tests consisted of five tests selected at random from all classes of tests within each chapter. The tests were compiled, linked, and executed as appropriate on each host/target computer. The test results were the same as those reviewed for the Series 300 Model 320 on which full testing was performed.

The compiler was tested using command scripts provided by Hewlett Packard and reviewed by the validation team. The following options were in effect for testing:

<u>Option</u>	<u>Effect</u>
-L	Generate a listing
-P 24	List page length (24 lines)
-e 999	Terminate compilation if the error count exceeds 999
-c	Suppress automatic linking of program
-W c, -STUBS	Places the code and symbol table information for instantiated units into the Ada library as distinct and separate units in the library (one unit in the library per instantiated generic). This option is used to decrease the possibility of exceeding the symbol table capacity when instantiating generic units.

Test output, compilation listings, job logs, and the compiler and its environment were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

The validation team arrived at Hewlett Packard in Cupertino CA on 8 December 1986, and departed after testing was completed on 11 December 1986.

Source files were read across a local area network which was not dedicated to only tested systems, so some delay may have been introduced by local area network traffic.

APPENDIX A
COMPLIANCE STATEMENT

Hewlett Packard has submitted the following compliance statement concerning the HP 9000 Series 200/300 Ada Compiler (Ada/300), Revision 1.4.

COMPLIANCE STATEMENT

Compliance Statement

Configurations:

Compiler: HP 9000 Series 200/300 Ada¹ Compiler, Revision 1.4

Test Suite: Ada Compiler Validation Capability, Version 1.8

Host and Target Computers:

Machine:	HP 9000 Series 200 Model 220
Operating System:	HP-UX, Revision 5.141

Machine:	HP 9000 Series 300 Model 310
Operating System:	HP-UX, Revision 5.141

Machine:	HP 9000 Series 300 Model 320
Operating System:	HP-UX, Revision 5.141

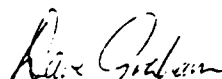
Machine:	HP 9000 Series 300 Model 330
Operating System:	HP-UX, Revision 5.19

Machine:	HP 9000 Series 300 Model 350
Operating System:	HP-UX, Revision 5.19

Hewlett Packard has made no deliberate extensions to the Ada language standard.

Hewlett Packard agrees to the public disclosure of this report.

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



Hewlett Packard
Dave Graham
Ada Project Manager

Date: 12/8/86

¹Ada is a registered trademark of the United States Government (Ada Joint Program Office).

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation classes. The implementation-dependent characteristics of the HP 9000 Series 200/300 Ada Compiler (Ada/300), Revision 1.4, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A). The specification of the package STANDARD is also included in this appendix.

Appendix F

Implementation-Dependent Characteristics for the HP Ada Compilation System

F.1 Form and Effect of Implementation-Dependent Pragmas

Pragmas *INTERFACE* and *INTERFACE_NAME*

Ada programs can interface with subprograms written in C and Assembler. This interface is possible through use of the predefined pragmas *INTERFACE* and *INTERFACE_NAME*.

The pragma *INTERFACE* informs the compiler that a non-Ada object is supplied when the Ada program is linked. The pragma specifies the name of the interfaced subprogram, the other programming language being interfaced to, and implicitly, the parameter calling conventions corresponding to that language.

The implementation-defined pragma *INTERFACE_NAME* associates an alternative name with a non-Ada subprogram that has been specified to the Ada program via the pragma *INTERFACE*.

The two pragmas take the form:

```
pragma INTERFACE (language_name, subprogram_name);  
pragma INTERFACE_NAME (subprogram_name, string);
```

where:

<i>language_name</i>	is either C or ASSEMBLER.
<i>subprogram_name</i>	is the name used within the Ada program when referring to the interfaced subprogram.
<i>string</i>	is the external name of the subprogram. This name is literal and case is significant.

The pragma `INTERFACE_NAME` is not required. If omitted, the name of the subprogram specified in pragma `INTERFACE` is used, with all alphabetic characters shifted to lower case. Pragma `INTERFACE_NAME` must be used when the interfaced subprogram name contains characters not acceptable within Ada names, or when the interfaced program name contains an uppercase letter.

If the interfaced subprogram language is C, a modification to the name of the interface subprogram is made by the compiler. This modification precedes the truncated name with an underscore character. This modification conforms to the naming conventions used by the linker for subprograms written in C.

Pragma `INTERFACE_NAME` is allowed at the same places of an Ada program as the pragma `INTERFACE` (see *Reference Manual for the Ada Programming Language*, Appendix B). The pragma `INTERFACE_NAME` must follow the corresponding `INTERFACE` pragma declaration.

EXAMPLE

```
package SAMPLE_LIB is
  function SAMPLE_DEVICE (X:INTEGER) return INTEGER;
  function PROCESS_SAMPLE (X:INTEGER) return INTEGER;
private
  pragma INTERFACE (ASSEMBLER, SAMPLE_DEVICE);
  pragma INTERFACE (C, PROCESS_SAMPLE);
  pragma INTERFACE_NAME (SAMPLE_DEVICE, "DEV10");
  pragma INTERFACE_NAME (PROCESS_SAMPLE, "psample");
end SAMPLE_LIB;
```

To avoid conflicts with the Ada run-time system, the names of interfaced subprograms should not begin with the letters *ALSY*, in any combination of upper and lower case.

F.2 Name and Type of Implementation-Dependent Attributes

There are no implementation dependent attributes available to the user.

```

generic
  type ELEMENT_TYPE is private;
  procedure STORE (INTO : ADDRESS; OBJECT : ELEMENT_TYPE);

```

```

end SYSTEM;

```

F.4 Restrictions on Representation Clauses

Representation clauses are not supported. Any program containing representation clauses is illegal and rejected at compile-time. Use of the pragma *PACK* results in a warning that the pragma is not supported.

F.5 Conventions for Implementation-Generated Names

There are no implementation-generated names available to the user.

F.6 Interpretation of Expressions Appearing in Address Clauses

Address clauses are not supported.

F.7 Restrictions on Unchecked Type Conversions

Unchecked conversions are allowed between any types implemented in the same physical size.

F.8 Implementation-Dependent Characteristics of the I/O Packages

The *Reference Manual for the Ada Programming Language* defines the predefined input-output packages listed below. Use of the facilities provided by these packages is described. The package *IO_EXCEPTIONS* specifies exceptions used by the other packages. *LOW_LEVEL_IO* is not implemented. There is no direct correspondence between HP-UX and Ada low-level I/O.

SEQUENTIAL_IO	(<i>Reference Manual for the Ada Programming Language</i> , 14.2.3)
DIRECT_IO	(<i>Reference Manual for the Ada Programming Language</i> , 14.2.5)
TEXT_IO	(<i>Reference Manual for the Ada Programming Language</i> , 14.3.10)
IO_EXCEPTIONS	(<i>Reference Manual for the Ada Programming Language</i> , 14.5)

Correspondence between External Files and HP-UX Files

Ada I/O is considered in terms of external files. Data is read from and written to external files. Each external file is implemented as a standard HP-UX file. Before an external file can be used by an Ada program, it must be associated with a file object belonging to that program. This association is achieved by supplying the name of the file object and the name of the external file to the procedures *CREATE* or *OPEN* of the predefined I/O packages. Once this association has been made, the external file can be read from or written to with the file object.

The name of the external file can be either of the following:

- Null string (*CREATE* only)
- HP-UX pathname

If the name is a null string, the associated external file is a temporary file, created using the HP-UX facility *tmpnam(3)*. This external file will cease to exist upon completion of the program.

If the external file is an HP-UX pathname, the pathname is extended in conformance with HP-UX rules (see *intro(2)* in the HP-UX Reference Manual). The exception *USE_ERROR* is raised by the procedure *CREATE* if the specified external file is a device. *USE_ERROR* is also raised for either *OPEN* or *CREATE* if the user has insufficient access rights to the file.

If an existing external file is specified to the *CREATE* procedure, the contents of that file will be deleted. The recreated file is left open, as for a newly created file.

NOTE

The execution of the procedures and functions of the predefined I/O packages involves the use of the Ada run-time system and the possible execution of a number of HP-UX I/O primitives. When such primitives are executed, an HP-UX I/O signal could be raised (such signals are raised, for example, if errors are detected during the processing of an I/O primitive).

If certain HP-UX I/O signals are raised, they are caught and processed by the Ada run-time system. This processing causes the appropriate Ada exception to be raised. Such Ada exceptions can then be handled by the Ada program.

Standard Implementation of External Files

External files have a number of implementation-dependent characteristics, such as their physical organization and file access rights. In the future, it will be possible to customize these characteristics through the *FORM* parameter of the *CREATE* and *OPEN* procedures. At this writing, only the default of *FORM* is supported, that is the null string. In addition, the functions *FORM* (of *TEXT_IO* and any instantiation of *DIRECT_IO* or *SEQUENTIAL_IO*) return a null string. The exception *USE_ERROR* is raised should a non-null *FORM* string be given to the *CREATE* or *OPEN* procedures. The following section describes the standard or default implementation of external files. It covers three types of external file: sequential, direct and text files. Default protection for external files is also described.

Sequential Files

For sequential access, the file is viewed as a sequence of values that are transferred in the order of their appearance (as produced by the program or by the run-time environment). Each file component object in a sequential file has exactly the same binary representation as the Ada object in the executable program.

The information placed in a sequential file is dependent upon whether the type, used for the instantiation of the sequential I/O packages, is constrained or unconstrained. If it is constrained, the objects are put consecutively into the file, without holes or separators. If the type is unconstrained, the length of the object (in bytes) is added to the front of the object, as a 32-bit integer value. There is no buffer between the physical external file and the Ada program.

Direct Files

For direct access, the file is viewed as a set of elements occupying consecutive positions in a linear order. The position of an element in a direct file is specified by its index, which is an integer in the range 1 to $(2^{31})-1$ of subtype *POSITIVE_COUNT*. Only objects of a constrained type can be written to or read from a direct access file. Such objects have exactly the same binary representation as the Ada object in the executable program. Although instantiation of *DIRECT_IO* is accepted for unconstrained types, the exception *USE_ERROR* is raised on any call to *CREATE* or *OPEN* where the object is of an unconstrained type.

All elements within the file have the same length. The number of bytes occupied by each element is determined by the size of the object stored in the file. There is no buffer between the physical external file and the Ada program.

Text Files

Text files are used for the input and output of information in human-readable form. Each text file is read or written sequentially, as a sequence of characters grouped into lines, and as a sequence of lines grouped into pages. All text file column numbers, line numbers, and page numbers are in the range 1 to $(2^{31})-1$ of subtype *POSITIVE_COUNT*. The line terminator (end-of-line) is physically represented by the ASCII character, *ASCII.LF*. The page terminator (end-of-page) is physically represented by a succession of the two characters, *ASCII.LF* and *ASCII.FF*, in that order. The file terminator (end-of-file) is physically represented by the character *ASCII.LF*, followed by the *HP-UX* end-of-file.

The user who controls line, page, and file structure by calling predefined subprograms (*Reference Manual for the Ada Programming Language*, 14.3.4) need not be concerned with the above terminator-implementation details. If structural control is effected by explicitly inputting or outputting these control characters (via the *PUT* function, for example), it is the user's responsibility to ensure the external file is maintained in a coherent state. The standard implementation of text files does not buffer text file I/O. However, buffering is a device-dependent characteristic that can be modified at the system level (for example, to buffer lines of text entered from a terminal). If a terminal is used for text input, the functions *END_OF_PAGE* and *END_OF_FILE* always return *FALSE*.

Access Protection of External Files

HP-UX provides protection of a file by means of access rights. These rights are used within Ada programs to protect external files. There are three levels of protection:

- User (the owner of the file).
- Group (users belonging to the owner's group).
- Others (users belonging to other groups).

For each of these levels, access to the file can be limited to one or several of the following rights: read, write, or execute. The standard external file access rights are specified by the *UMASK* command (see *umask(1)* and *umask(2)* in the *HP-UX Reference Manual*). Access rights apply equally to sequential, direct, and text files.

The Sharing of External Files and Tasking Issues

Several file objects can be associated with the same external file. These file objects can have identical or differing I/O modes; each file object has independent access to the external file. The effects of sharing an external file depend on the nature of the file. The user must keep in mind the nature of the device attached to the file object and the sequence of I/O operations on the device. Multiple I/O operations on an external file shared by several file objects are processed in the order they are received.

In the case of shared files on random access devices such as disks, the data is shared. Reading from one file object does not affect the file positioning of another file object, nor the data available to it. Simultaneous reading and writing of separate file objects to the same random access external file should be avoided however; due to buffering, the effects are unpredictable.

In the case of shared files as sequential or interactive devices such as mag tapes or keyboards, the data is no longer shared. In other words, a mag tape record or keyboard input buffer read by one I/O operation is no longer available the next operation, whether it is performed on the same file object or not. This is simply due to the sequential nature of the device. By default, the Ada file objects represented by `STANDARD_IN` and `STANDARD_OUT` are preconnected to the HP-UX streams *stdin* and *stdout* (see *stdio(5)*), and thus are of this sequential variety of file.

Each file operation is completed before a subsequent file operation commences. In a tasking program, this means no explicit synchronization needs be performed unless the order of I/O operations is critical. An Ada tasking program is erroneous if it depends on the order of I/O operations without explicitly synchronizing them. Remember that the files associated with `STANDARD_IN` and `STANDARD_OUT` are shared by all tasks, and that care must be taken when Ada file objects use buffered I/O.

I/O Involving Access Types

When an object of an access type is specified as the source or destination of an I/O operation (write or read), the 32-bit binary access value is read or written unchanged. If an access value is read from a file, care should be taken to ensure that the access value so read, designates a sensible object. This is only likely to be the case if the access value read was previously written by the same program that is reading it, and the object which it designated at the time it was written still exists (that is, the scope in which it was allocated has not been exited, nor has an `UNCHECKED_DEALLOCATION` been performed on it). A program may execute erroneously if an access type read from a file does not designate a sensible object.

I/O Involving Local Area Networks

Ada I/O can be used reliably across local area networks only if the network special files representing remote file systems are contained in the directory */net*, as is customary.

Implementation-Defined I/O Packages

`UNIX_ENV` is the only implementation-defined I/O package. An Ada program has no implicit access to its execution environment. In particular, there is no direct equivalent to the `argc` and `argv` parameters of a C main program. In this implementation, parameters cannot be passed to a procedure used as an Ada main program. The binder will not bind a program with parameters in its main program. The implementation predefined package `UNIX_ENV` enables an Ada program to retrieve information from the HP-UX environment. The specification of `UNIX_ENV` is listed below and in Chapter 4 of the *Ada Users Guide*.

```
package UNIX_ENV is
  -- This package enables an Ada program to retrieve information from
  -- its HP-UX environment.
  --
  -- WARNING: The function ENT_VALUE will not behave correctly if the
  --          program is linked with an interface C routine named
  --          GETENV.

  function ARG_COUNT return POSITIVE;
  -- Returns the number of arguments in the command line of the program
  -- (same as argc parameter of a C program); this number is always
  -- positive (the first argument being the name by which the program
  -- has been invoked).

  function ARG_VALUE (INDEX : in NATURAL) return STRING;
  -- Returns the INDEXth argument in the command line of the program
  -- (same as argv[INDEX] parameter of a C program); if INDEX
  -- is  $\geq$  ARG_COUNT, the function will raise ILLEGAL_ARG_INDEX
  -- exception.

  ILLEGAL_ARG_INDEX : exception;

  function ENV_VALUE (NAME : in STRING) return STRING;
  -- Returns the value of environment variable NAME (same as
  -- getenv(NAME) C function); if there is no environment function named
  -- NAME, the function will raise UNKNOWN_ENV_NAME exception.

  UNKNOWN_ENV_NAME : exception;

  type EXIT_STATUS is range 0 .. 255;
  -- Declared so that a variable of this type is constrained to 8
  -- significant bits (as it may be tested under HP-UX) but implemented
  -- as a 16-bit integer (for the sake of C interface).

  procedure SET_EXIT_STATUS (STATUS : in EXIT_STATUS);
  -- Sets the HP-UX exit status of the program. This status is 0 by
  -- default. Any call to this procedure overrides the previous value
  -- of the status.

  function EXEC_SHELL_COMMAND (COMMAND : in STRING) return EXIT_STATUS;
  -- Passes the given string as a command to the SH HP-UX shell, waits
  -- until shell completion and returns the exit status of the shell.

end UNIX_ENV;
```

The use of a number of these functions is demonstrated by the following example command line

```
prog1 info1 info2
```

If applied to this example, the function *ARG_COUNT* returns the value 3, and the function calls *ARG_VALUE(0)*, *ARG_VALUE(1)*, and *ARG_VALUE(2)* return the strings "*prog1*", "*info1*" and "*info2*".

F.9 Interfaced Subprograms

Parameter and Result Passing Conventions

The following Ada types are not supported when passing parameters to, nor returning results from, interfaced subprograms (as neither C nor ASSEMBLER recognize these types):

- Task types
- Private types
- Fixed-point types

The result type of an interfaced subprogram is further restricted to be of a scalar or access type.

Subprogram parameters are passed on the stack in the reverse order of their declaration. Since this is the C parameter-passing convention, interfaced subprograms written in C will automatically interface properly. For interfaced subprograms written in ASSEMBLER, it is the programmer's responsibility to follow the same convention.

Three parameter passing modes are supported for interfaced subprograms:

- In
- In Out
- Out

When passing scalar types (integer, real, character, or enumeration) or access types, to formal parameters of mode *in*, each scalar or access type is passed as a copy of its value.

All supported types passed with modes *in out* or *out*, and array or records types passed with mode *in*, are passed by reference (address). In particular, an access type passed as an *in out* or *out* parameter is passed as the address of the access value (which is in turn the address of the object it designates); the designated object is therefore doubly indirect within the interfaced subprogram.

NOTE

No consistency checking is performed between the subprogram parameters as declared in Ada and the corresponding parameters in the interfaced subprogram.

If the ADDRESS of any object is passed to an interface subprogram, that object is not protected from being updated by the interfaced subprogram. Such objects, as well as parameters passed with mode *in out* or *out* (and *in* for arrays and records), will have no run-time checks performed on

their contents upon return from the interfaced subprogram. Erroneous program behavior may result if the parameter values are altered such that they violate Ada constraints for the type of the actual parameter.

When calling an ASSEMBLER subprogram, the Ada compiler expects the contents of registers D2 through D7 and A2 through A7 to be left unchanged. The ASSEMBLER subprogram must save and restore these registers if it, or any routine it calls, might change them.

Scalar Parameters

Integer Types

The integer types `LONG_INTEGER`, `INTEGER`, and `SHORT_INTEGER` may be passed to interfaced subprograms.

When passed by value to an interfaced subprogram written in C, all integer values are sign extended to 32-bits and the 32-bit value is passed to C. This conforms with the C parameter-passing conventions. Values of type `LONG_INTEGER` are pushed onto the stack unchanged. Values of type `INTEGER` and `SHORT_INTEGER` are sign extended on the left to 32 bits and the 32-bit value is pushed onto the stack.

When passed by reference to an interfaced subprogram written in C, no integer values are sign extended or modified in any way. The 32-bit address of the value is pushed onto the stack.

NOTE

The term *short integer* has different meanings to Ada and C. A `SHORT_INTEGER` in Ada occupies 8 bits, while one in C (a *short int* or *int*) occupies 16 bits. An ordinary `INTEGER` passed by reference from Ada to C, must be treated as a *short integer* (using *pointer-to short int*) in C. An Ada `SHORT_INTEGER` passed by reference from Ada to C, cannot be meaningfully accessed as any type of integer in C at all. An Ada `SHORT_INTEGER` passed by reference from Ada to C can, however, be accessed as a character in C (using *pointer-to char*). Since C permits "integer like" operations on objects of type *char*, an Ada `SHORT_INTEGER` passed by reference to C, can be successfully used and modified by C (as an object of type *char*). This difficulty does not apply to integers passed by value, as they are all sign extended and passed in 32-bit constructs, as described earlier.

When passed by value to an interfaced subprogram written in ASSEMBLER, values of type `LONG_INTEGER` and `INTEGER` are pushed onto the stack unchanged (as 32-bit and 16-bit values respectively). Values of type `SHORT_INTEGER` are pushed onto the stack in the *high-order* (leftmost) 8 bits of a 16-bit field (the low order 8 bits of the 16-bit field are meaningless and should not be accessed).

When passed by reference to an interfaced subprogram written in ASSEMBLER, no integer values are sign extended or modified in any way. The 32-bit address of the value is pushed onto the stack.

Character Types

All Ada character types (including the predefined type `CHARACTER`) correspond exactly with the type `char` in C. Both the Ada and C types occupy 8 bits and have the same internal representation.

When passed by value to an interfaced subprogram written in C, they are extended to 32 bits and the 32-bit value is pushed onto the stack, with the actual value in the *low-order* (rightmost) 8 bits.

When passed by reference to an interfaced subprogram written in C, the value is unchanged and the 32-bit address of the character is pushed onto the stack.

Character values, whether passed by value or reference to C, can be used directly by the interfaced C subprogram as values of type `char` (using pointer-to `char` when passed by reference).

When passed by value to an interfaced subprogram written in ASSEMBLER, character values are pushed onto the stack in the *high-order* (leftmost) 8 bits of a 16-bit field (the low order 8 bits of the 16-bit field are meaningless and should not be accessed).

When passed by reference to an interfaced subprogram written in ASSEMBLER, the value is unchanged and the 32-bit address of the character is pushed onto the stack.

Enumeration Types

Values of an Ada enumeration type are represented internally as positive integers representing their position in the list of enumeration literals defining the type. The first literal in the list corresponds to an integer value of zero. Enumeration types with 128 elements or less are represented as short (8-bit) integers, those with more than 128 elements as ordinary (16-bit) integers. The maximum number of values an enumeration type can include ($2^{15} - 1$).

When an enumeration value is passed to an interfaced subprogram, the underlying integer value is passed as described above for integer types:

- When passed by value to C, they are extended to 32 bits and the 32-bit value is pushed onto the stack.
- When passed by value to ASSEMBLER, a 16-bit value is passed unchanged, while an 8-bit value is passed in the *high-order* (leftmost) 8 bits of a 16-bit value pushed onto the stack.
- When passed by reference, the value is unchanged and the 32-bit address of the value is pushed onto the stack.

Enumeration values in C are represented in the same general way as in Ada, but are always expressed as 16-bit integers, regardless of the number of elements. When C passes an enumeration value by value, it is extended to 32-bits with the high-order 16 bits disregarded. Notice that the automatic 32-bit extension performed when Ada passes an enumeration value to C by value, ensures that it will be represented in the proper C (by value) form.

When Ada passes an enumeration value by reference, however, the value is not extended; consequently, 8-bit enumeration values from Ada cannot be used directly by C when passed by reference (as C would

assume the reference to be to a 16-bit object). An Ada 8-bit enumeration passed by reference to C can, however, be accessed as a character in C (using pointer-to *char*). Since C permits "integer like" operations on objects of type *char*, an Ada 8-bit enumeration value passed by reference to C, can be successfully used and modified by C as an object of type *char*. This difficulty does not apply to enumeration values passed by value, as they are all sign extended and passed in 32-bits, as noted above.

Floating Point Types

Ada has one floating-point type, the type *FLOAT*, which is implemented in 32-bits with the IEEE standard representation *A Proposed Standard for Binary Floating-Point Arithmetic*, IEE P754, draft 10.0) and which is passed as a parameter with that representation and length. C has both the 32-bit IEEE floating-point type (*float*) which corresponds to the Ada *FLOAT* type, and a 64-bit IEEE floating-point type (*double*), which Ada does not currently support.

However, C extends all 32-bit (*float*) values to 64 bits (*double*) when they are passed as parameters or returned as results. Thus the Ada *FLOAT* type cannot be passed by value directly to C, nor returned directly from C. An Ada *FLOAT* can be passed directly to C by reference, as C does not expect such parameters to be extended.

It is however possible to pass Ada *FLOAT* values to C by value and to return Ada *FLOAT* values from C if the C routine uses a union containing a *float* field and an *long int* field.

A sample C function to square an Ada floating point value follows. The parameter in the C function is declared to be the union type (the union type has a size of 32-bits, which matches the Ada *FLOAT* value being passed). The function uses the *float* field (f) of the union to access the Ada floating point value and stores its result in a union of the same type. It returns the *long int* field (i) of the union which overlays (occupies the same storage as) the *float* field and which is returned in the manner in which Ada expects a *FLOAT* function result to be returned. Note that neither the *float* field of the union nor the entire union should be returned (the former would be returned as a *double* and the latter is a C structured type and would returned in a manner that is not consistent with where Ada expects a *FLOAT* result to be returned).

```
typedef union {float f; long int i} ada_float;

int squareit (parm)
ada_float parm;
{
    ada_float result;

    result.f = parm.f * parm.f;

    return result.i;
}
```

A sample Ada procedure to call the sample C function follows:

```

with TEXT_IO;
procedure SQUARE_PI is

pi  : constant FLOAT := 3.1415926;
pi2 : FLOAT;

package FLOATER is new TEXT_IO.FLOAT_IO (FLOAT);

function SQUARE (f: FLOAT) return FLOAT;
pragma INTERFACE (C, SQUARE);
pragma INTERFACE_NAME (SQUARE, "squareit");

begin -- SQUARE_PI
    pi2 := SQUARE (pi);
    TEXT_IO.PUT ("Pi squared = ");
    FLOATER.PUT (pi2);
    TEXT_IO.NEW_LINE;
end SQUARE_PI;

```

Fixed-Point Types

Fixed-point types are not supported as parameters or results of interfaced subprograms.

Access Types

Values of an access type are represented internally by the 32-bit address of the underlying designated object. When an access value is passed by value, this 32-bit address itself is pushed onto the stack. When passed by reference, a double-indirect address (the address of the address of the designated object) is pushed instead.

Composite Types

When a composite type (an array or record type) is passed as parameter to an interfaced subprogram, it is always passed by reference (regardless of whether the mode is *in*, *in out*, or *out*). A component of a composite type is passed according to its type classification (scalar, access, or composite). The following section discusses the passing of composite type parameters.

Array Types

Ada arrays are always passed by reference; the value passed on the stack is the 32-bit address of the first element of the array. When an array is passed as a parameter to an interfaced subprogram, the usual consistency checking between the array bounds declared in the calling program and the subprogram are not enforced. It is the programmer's responsibility to ensure that the subprogram does not violate the bounds of the array.

A description of array element allocation and alignment can be found in section F.12 of this appendix.

Note that Ada arrays with *SHORT_INTEGER* and 8-bit enumeration type components do not correspond directly with any C array type; however they can be accessed in C as an array of *char*. Since the contents of such arrays are not characters, the C routine must access and modify them as is appropriate for the actual component type.

String Types

Values of the predefined type *STRING* are a special case of arrays, and are passed on the stack in the same way; the 32-bit address of the first character in the string is pushed onto the stack.

Note that Ada strings do not (normally) end with an ASCII null character, as required by the C string representation convention. It is the programmer's responsibility to append a null character (ASCII NUL) to the end of the Ada string before passing it to a C subprogram, and if necessary to delete the null character on return.

Record Types

Like arrays, Ada records are always passed by reference; the value passed is the 32-bit address of the first component of the record. However, unlike arrays, the individual components of a record may be reordered internally by the Ada compiler. Moreover, if a record contains discriminants or composite components having a dynamic size, implicit components may have been added to the record.

The exact internal structure of a record is, therefore, not known directly at coding time. However, the location of components of a record may be determined with the `POSITION` attribute, which evaluates to the offset of a record component with respect to the starting address of the record. By passing such record component offsets, in addition to the record, Ada record components can be accessed and modified by an interfaced subprogram (although considerable pointer arithmetic, or type conversion and casting, or both, be necessary in the interfaced subprogram).

Result Returned by an Interfaced Function Call

Only objects of a scalar type or an access type can be returned as the result of an interfaced function call. All scalar types except `FLOAT` are returned correctly in a straightforward manner. For information on returning a value of type `FLOAT`, refer to "Floating-Point Types," earlier in this appendix.

Potential Problems when Using Interfaced Subprogram Units

The Ada run-time system generates and catches the following HP-UX signals:

- SIGALRM
- SIGILL
- SIGIOT
- SIGSEGV
- SIGBUS

Any attempt to trap or ignore these reserved signals by an interface subprogram that is not part of the Ada run-time system may have unpredictable results (except as described later in this appendix).

Problems can arise if an interfaced subprogram initiates a "slow" system function that can be interrupted by signal (for example, a read on a terminal or a wait for a child process to complete) or if an interfaced subprogram can be called by more than one task and is not reentrant. If an Ada reserved signal occurs during such an operation or nonreentrant region, the program may function erroneously.

For example, an Ada program that using tasking will cause the generation of `SIGALRM`. If an interfaced subprogram needs to perform a potentially interruptible system call, or if it might be called from more than one task and is not reentrant, it can be protected by blocking `SIGALRM` around the system call or nonreentrant region. Here is an example of a protected `read(2)` in C:

```

#include <signal.h>
void interface_rout()
{
    long mask;

    ...

    /*
    Add SIGALRM to the list of currently blocked signals (see sigblock(2)).
    */

    mask = sigblock (1L << (SIGALRM-1));

    ... read (...) ;    /* or nonreentrant region */

    sigsetmask (mask); /* return to previous mask */

    ...
}

```

If any Ada reserved signal other than *SIGALRM* is to be similarly blocked, *SIGALRM* must be blocked at the same time or already be blocked. When any Ada reserved signal other than *SIGALRM* is unblocked, *SIGALRM* must be unblocked at the same time or be unblocked later.

Any Ada reserved signal blocked in interfaced code, should be unblocked as soon as is possible, to avoid unnecessarily stalling the Ada run-time system.

F.10 Predefined Language Attributes

The predefined language attributes are implemented as defined in the *Reference Manual for the Ada Programming Language*, Appendix A, with the following exception.

P'STORAGE_SIZE

When applied to an access type or subtype of an access type the compiler produces one of two responses

- If there is no length clause, the compiler issues the warning:

The prefix of the attribute STORAGE_SIZE is an access type which has no length clause. The value returned by STORAGE_SIZE is always 0.

- If a length clause is specified, the compiler issues the error:

Length clauses are not currently implemented.

F.11 Predefined Language Pragmas

The predefined language pragmas are implemented as defined in the *Reference Manual for the Ada Programming Language*, Appendix B, with the following exceptions (see also "Form and Effect of Implementation-Dependent Pragmas," later in this appendix).

Use of the predefined language pragmas

CONTROLLED
MEMORY_SIZE
OPTIMIZE
PACK
SHARED,
STORAGE_UNIT
SYSTEM_NAME

are either ignored without comment (the pragma has no effect) or elicits a warning from the compiler:

This pragma is not currently supported by the implementation.

F.12 Data Type Ranges, Alignment, and Layout

Table F-1. Ranges

Types	Ranges
SHORT INTEGER	-128.. 127
INTEGER	-32768.. 32767
LONG INTEGER	-2_147_483_648.. 2_147_483_647
ENUMERATION	32768 elements maximum
BOOLEAN	FALSE.. TRUE
CHARACTER	0..127 (ASCII.NUL.. ASCII.DEL)
FLOATING POINT	-2#1.111_1111_1111_1111_1111_1111#E+127 .. 2#1.111_1111_1111_1111_1111_1111#E+127 (that is, -3.402823E+38 .. +3.402823E+38)

Table F-2. Alignment

Types	Alignment
SHORT INTEGER	allocated on a byte boundary
INTEGER	allocated on a word boundary
LONG INTEGER	allocated on a word boundary
ENUMERATION (≤ 128 elements)	allocated on a byte boundary
ENUMERATION (> 128 elements)	allocated on a word boundary
BOOLEAN	allocated on a byte boundary
CHARACTER	allocated on a byte boundary
FIXED POINT	mapped into one of the three integer types
FLOATING POINT	allocated on a word boundary
COMPOSITE	may be allocated on a byte or word boundary depending on the alignment constraints of the components or elements
ACCESS	allocated on a word boundary

Layout

1. Integer Types

`SHORT_INTEGER`, `INTEGER` and `LONG_INTEGER` are respectively allocated 8 bits, 16 bits and 32 bits. For a user-defined integer type, the representation chosen is the smallest of the predefined integer types whose range includes the range of the declared type.

2. Enumeration Types Other Than Boolean

A value of an enumeration type is represented by the integer value corresponding to its position in the type definition (positions being numbered from 0). The values of enumeration types are signed. Thus, only enumeration types defining up to 128 elements are represented in a byte.

3. The Type Character

The type character is represented as a user defined enumeration type of less than 129 elements would be (integer range 0..127, corresponding to `ASCII.NUL`..`ASCII.DEL`).

4. The Type Boolean

The values `FALSE` and `TRUE` are represented as the unsigned integer values 0 and 255 in a byte. Although the internal representation of `TRUE` is 255, the value of `BOOLEAN'POS (TRUE)` is 1.

5. Fixed Point Types

A value of a fixed point type is managed by the compiler as the value of *signed_mantissa* * *small*. *Signed_mantissa* is a signed integer. *Small* is the largest power of two that is not greater than the delta of the fixed accuracy definition. Thus fixed point types are mapped into one of the three predefined integer types.

6. Floating Point Types

A floating point value is represented in normalized 32-bit IEEE format:

bit 0: sign (most significant bit)

bits 1 to 8: exponent

bits 9 to 31: mantissa

7. Array Types

- General case: Elements are allocated contiguously in memory and stored by columns. The size of an array is the product of the number of its elements and its element size.
- Special case: Elements with an odd number of bytes are padded so as to start at a word boundary. An example of this is an array, with an element type that is a record. The record contains two components, of type `INTEGER` and `BOOLEAN`, respectively. Instead of allocating these elements contiguously, all elements are extended with a null byte to make them even sized. Therefore, the

array size (in bytes) is the product of the number of elements and the number represented by a value greater than the element size (in bytes).

8. Record Types

Record components are allocated contiguously in memory with holes inserted between components to satisfy their alignment constraints. The alignment constraint of a record is the strongest alignment constraint of any of its components.

Implicit components are components created by the compiler within a record. They can be used to hold the record size, array or record descriptors or variant indexes. The implicit component for the record size is in general laid out at the beginning of the record.

Record components can be rearranged by the compiler to take into account alignment constraints.

For dynamic components, a *dope* (pointer) is created where the component would normally have been laid out, and the component itself is allocated at the end of the record. The *dope* contains the offset from the start of the record to the actual component.

For a record type with variant parts, the common part and each variant part are laid out in the same way as described above, with the exception that dynamic components are laid out at the end of the entire record, not at the end of a specific variant part. Variant parts are overlaid when possible and holes of previously laid out variant parts are used whenever possible.

9. Parameters

When passing parameters by value within Ada, byte aligned objects are considered as word aligned and right justified and odd-byte-sized parameters are passed in even-byte-sized containers. This ensures that the stack pointer remains even. (For information on passing parameter to interfaced subprograms, refer to section F.9 of this appendix.)

F.13 Compiler Limitations

NOTE

It is impossible to give exact numbers for most of the limits given here. The various language features may interact in complex ways to lower the limits given below. The numbers represent "hard" limits in simple program fragments devoid of other Ada features.

Limit	Description
255	Maximum number of characters in a source line (**)
255	Maximum identifier length (**)
15300	Maximum number of characters in a string literal.
1023	Maximum number of compilation units in a library.
3000	Maximum number of identifiers in a unit. An identifier includes enumerated type identifiers, record field definitions, and (generic) unit parameter definitions.
150	Maximum "structure" depth. Structure includes the following: nested blocks, compound statements, aggregate associations, parameter associations, subexpressions.
15	Maximum array dimensions. Set to Maximum structure depth/10 (*)
144	Maximum number of discriminants in a record constraint (*)
32767	Maximum number of enumeration elements in a single enumeration type (this limit is further constrained by the maximum number of identifiers in a unit). (*)
82	Maximum number of associations in a record aggregate (*)
125	Maximum number of parameters in a subprogram definition (*)
190	Maximum expression depth (*)
30	Maximum number of "created" units in a single compilation.
74	Maximum number of nested frames. Library level unit counting as a frame.
250	Maximum number of overloads per compilation unit.
50	Maximum number of overloads per identifier
-	Maximum number of tasks is only limited by heap size.

2**31-1 Maximum number of bits in any size computation

* There is a limit on the size of tables used in overloading resolution which can potentially lower this figure. This value is set at 5 and reflects the number of possible interpretations of names in any single construct under analysis by the compiler (procedure call, assignment statement etc)

** Limit not affected by context.

F.14 Specification of the Package STANDARD

Package STANDARD is

...

type INTEGER is range -32768 .. 32767;

type SHORT_INTEGER is range -128 .. 127;

type LONG_INTEGER is range -2147483648 .. 2147483647;

type FLOAT is digits 6 range -2#1.11111111111111111111111111111111#E-127

.. 2#1.11111111111111111111111111111111#E+127;

type DURATION is delta 0.02 range -86400.0 .. 86400.0;

...

end STANDARD;

APPENDIX C TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
\$BIG ID1 Identifier the size of the maximum input line length with varying last character.	(1..5 => "X2345", 6..15 16..25 26..35 36..45 46..55 => "6789012345", 56..254 => 'A', 255 => '1')
\$BIG ID2 Identifier the size of the maximum input line length with varying last character.	(1..5 => "X2345", 6..15 16..25 26..35 36..45 46..55 => "6789012345", 56..254 => 'A', 255 => '2')
\$BIG ID3 Identifier the size of the maximum input line length with varying middle character.	(1..5 => "X2345", 6..15 16..25 26..35 36..45 46..55 => "6789012345", 56..126 128..255 => 'A', 127 => '3')
\$BIG ID4 Identifier the size of the maximum input line length with varying middle character.	(1..5 => "X2345", 6..15 16..25 26..35 36..45 46..55 => "6789012345", 56..126 128..255 => 'A', 127 => '4')
\$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..252 => '0', 253..255 => "298")

TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
<u>\$BIG_REAL_LITERAL</u> A real literal that can be either of floating- or fixed-point type, has value 690.0, and has enough leading zeroes to be the size of the maximum line length.	(1..249 => '0', 250..255 => "09.0E1")
<u>\$BLANKS</u> A sequence of blanks twenty characters fewer than the size of the maximum line length.	(1..235 => ' ')
<u>\$COUNT_LAST</u> A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
<u>\$EXTENDED_ASCII_CHARS</u> A string literal containing all the ASCII characters with printable graphics that are not in the basic 55 Ada character set.	"abcdefghijklmnopqrstuvwxyz" & "!\$%?@[\\]^`{}~"
<u>\$FIELD_LAST</u> A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
<u>\$FILE_NAME_WITH_BAD_CHARS</u> An illegal external file name that either contains invalid characters, or is too long if no invalid characters exist.	FILE_NAME_LONGER_THAN_14_CHARS
<u>\$FILE_NAME_WITH_WILD_CARD_CHAR</u> An external file name that either contains a wild card character, or is too long if no wild card character exists.	not_there/*/*/*not_there
<u>\$GREATER_THAN_DURATION</u> A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST if any, otherwise any value in the range of DURATION.	100_000.0
<u>\$GREATER_THAN_DURATION_BASE_LAST</u> The universal real value that is greater than DURATION'BASE'LAST, if such a value exists.	100_000_000.0

<u>Name and Meaning</u>	<u>Value</u>
\$ILLEGAL_EXTERNAL_FILE_NAME1 An illegal external file name.	not_there//not_there/*
\$ILLEGAL_EXTERNAL_FILE_NAME2 An illegal external file name that is different from \$ILLEGAL_EXTERNAL_FILE_NAME1.	"not_there/not_there/not_there" & "/././not_there///"
\$INTEGER_FIRST The universal integer literal expression whose value is INTEGER'FIRST.	-32768
\$INTEGER_LAST The universal integer literal expression whose value is INTEGER'LAST.	32767
\$LESS_THAN_DURATION A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST if any, otherwise any value in the range of DURATION.	-100_000.0
\$LESS_THAN_DURATION_BASE_FIRST The universal real value that is less than DURATION'BASE'FIRST, if such a value exists.	-100_000_000.0
\$MAX_DIGITS The universal integer literal whose value is the maximum digits supported for floating-point types.	6
\$MAX_IN_LEN The universal integer literal whose value is the maximum input line length permitted by the implementation.	255
\$MAX_INT The universal integer literal whose value is SYSTEM.MAX_INT.	2_147_483_647

TEST PARAMETERS

Name and Meaning	Value
\$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER if one exists, otherwise any undefined name.	NO_SUCH_TYPE
\$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#FF_FF_FF_FD#
\$NON_ASCII_CHAR_TYPE An enumerated type definition for a character type whose literals are the identifier NON_NULL and all non-ASCII characters with printable graphics.	(NON_NULL.)

APPENDIX D
WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 19 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

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

- . B4900bA: Object declarations at lines 41 and 50 are terminated incorrectly with colons, and end case; is missing from line 42.
- . B4A010C: The object declaration in line 10 follows a subprogram body of the same declarative part.
- . B74101B: The begin at line 9 causes a declarative part to be treated as a sequence of statements.
- . C87B50A: The call of "/=" at line 31 requires a use clause for package A.
- . C92005A: The "/=" for type PACK.BIG_INT at line 40 is not visible without a use clause for the package PACK.
- . C940ACA: The assumption that allocated task TT1 will run prior to the main program, and thus assign SPYNUMB the value checked for by the main program, is erroneous.
- . CA3005A..D (4 tests): No valid elaboration order exists for these tests.
- . BC3204C: The body of BC3204C0 is missing.

END

DATE

FILMED

DTIC

4/88