②

UNCLASSIFIED

SECU ... Entered

AD-A209 882

ON PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

	12. GOVT ACCESSION NO	3. RECIPIENT'S CATALOG NUMBER
1.		

4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
Ada Compiler Validation Summary Report: CONVEX Computer Corporation, CONVEX Ada, Version 1.1 C210, (Host and Target), 890508W1.10077	08 May 1989 to 08 May 1990
	6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
Wright-Patterson AFB Dayton, OH, USA	

9. PERFORMING ORGANIZATION AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Wright-Patterson AFB Dayton, OH, USA	

11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE
Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	
	13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)	15. SECURITY CLASS (of this report)
Wright-Patterson AFB Dayton, OH, USA	UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DTIC
ELECTE
JUN 3 0 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)

CONVEX Computer Corporation, CONVEX Ada, Version 1.1, C210 under CONVEX UNIX, Version 7.1 (Host and Target), ACVC 1.10, Wright-Patterson AFB.

89 C 165

Ada   Compiler Validation Summary Report:

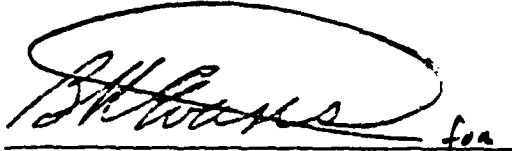Compiler Name: CONVEX Ada, Version 1.1

Certificate Number: 890508W1.10077

Host: C210 under
          CONVEX UNIX, Version 7.1

Target: C210 under
          CONVEX UNIX, Version 7.1
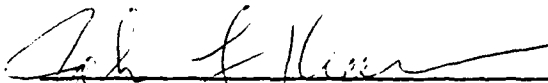

Testing Completed 8 May 1989 Using ACVC 1.10

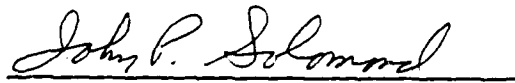This report has been reviewed and is approved.

_____ for

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

_____

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

_____

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

Ada   COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890508W1.10077
CONVEX Computer Corporation
CONVEX Ada, Version 1.1
C210

Completion of On-Site Testing:
8 May 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: CONVEX Ada, Version 1.1

Certificate Number: 890508W1.10077

Host: C210 under
         CONVEX UNIX, Version 7.1

Target: C210 under
         CONVEX UNIX, Version 7.1

Testing Completed 8 May 1989 Using ACVC 1.10

This report has been reviewed and is approved.

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

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

Ada Joint Program Office
Dr. John Solomond
Director, AJPO
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 8 May 1989 at Richardson, TX.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

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

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

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

or from:

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

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

> Ada Validation Organization
> Institute for Defense Analyses
> 1801 North Beauregard Street
> Alexandria VA 22311

1.3 REFERENCES

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

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

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

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

1.4 DEFINITION OF TERMS

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

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

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

Applicant   The agency requesting validation.

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

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

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

cross-compilers, translators, and interpreters.

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

Host   The computer on which the compiler resides.

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

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

Target   The computer for which a compiler generates code.

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

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

1.5 ACVC TEST CLASSES

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

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

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

illegal construct that it contains is detected by the compiler.

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

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

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

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

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

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

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

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

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

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

Compiler· CONVEX Ada, Version 1.1

ACVC Version: 1.10

Certificate Number: 890508W1.10077

Host Computer:

Machine:	C210
Operating System:	CONVEX UNIX Version 7.1
Memory Size:	64 Megabytes

Target Computer:

Machine:	C210
Operating System:	CONVEX UNIX Version 7.1
Memory Size:	64 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. 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 SHORT_FLOAT in package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

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

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

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

   (3) This implementation uses no extra bits for extra precision and

uses all extra bits for extra range. (See test C35903A.)

(4) 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 not gradual. (See tests C45524A..Z.)

d. Rounding.

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

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

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

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

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

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

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

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

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

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

f. Discriminated types.

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

g. Aggregates.

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

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

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

h. Pragmas.

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

i. Generics

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

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

j. Input and output

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

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

    (3) 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, CE2102R, CE2102T, and CE2102V.)

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

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

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

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

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

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

CONFIGURATION INFORMATION

(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, 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 reading or writing. (See tests CE3111A..E, CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 43 tests had been withdrawn because of test errors. The AVF determined that 329 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation . Modifications to the code, processing, or grading for eleven 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	1993	17	28	46	3345
Inapplicable	0	6	323	0	0	0	329
Withdrawn	1	2	34	0	6	0	43
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	198	577	545	245	172	99	161	332	137	36	252	292	299	3345
Inappl	14	72	135	3	0	0	5	1	0	0	0	77	22	329
Wdrn	1	1	0	0	0	0	0	1	0	0	1	35	4	43
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 43 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

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

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

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 329 tests were inapplicable for the reasons indicated:

   a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y	C35705L..Y	C35706L..Y	C35707L..Y
C35708L..Y	C35802L..Z	C45241L..Y	C45321L..Y
C45421L..Y	C45521L..Z	C45524L..Z	C45621L..Z
C45641L..Y	C46012L..Z		

b. C35702B and B86001U are not applicable because this implementation supports no predefined type LONG_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. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 47.

e. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

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

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

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

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

j. CD2A61I and CD2A61J are not applicable because this implementation does not support size clauses for array types, which imply compression, with component types of composite or floating point types. This implementation requires an explicit size clause on the component type.

k. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because this implementation does not support size clauses for access types.

l. CD2A91A..E (5 tests), CD5012J, CD5013S, and CD5014S are not applicable because this implementation does not support size clauses for tasks or task types.

m. The following 42 tests are not applicable because this implementation does not support an address clause when a dynamic address is applied to a variable requiring initialization:

CD5003B..H	CD5011A..H	CD5011L..N	CD5011Q
CD5011R	CD5012A..I	CD5012L	CD5013B
CD5013D	CD5013F	CD5013H	CD5013L

CD5013N   CD5013R   CD5014T..X

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 this implementation supports CREATE
    with IN_FILE mode for text files.

ad. CE3102F is inapplicable because this implementation supports RESET for
    text files.

ae. CE3102G is inapplicable because this implementation supports deletion of an external file for text files.

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

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

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

ai. CE3115A is not applicable because resetting of an external file with OUT_FILE mode is not supported with multiple internal files associated with the same external file when they have different modes.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

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

Modifications were required for eleven tests.

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

B24009A	B33301B	B38003A	B38003B	B38009A	B38009B
B41202A	B91001H	BC1303F	BC3005B		

The following modifications were made to compensate for legitimate implementation behavior.

Since this implementation uses 4 storage units for Booleans, the following changes were made to CD1CO4E on the recommendation of the AVO: on line 51, "1..BOOLEAN'SIZE + 1" was changed to "0..BOOLEAN'SIZE -1" and lines 75, 99, and 107 were commented out. After these modifications, the test executed and reported passed.

TEST INFORMATION

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

3.7.2 Test Method

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

Host computer:	C210
Host operating system:	CONVEX UNIX, Version 7.1
Target computer:	C210
Target operating system:	CONVEX UNIX, Version 7.1
Compiler:	CONVEX Ada, Version 1.1

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

The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the C210. Results were transferred via magnetic tape to another system for printing.

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

OPTION	EFFECT
-nw	Suppress warning messages
-M	Upon successful compilation, perform link phase and build an executable.
-o	Specifies name of executable

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the

validation team were also archived.

3.7.3 Test Site

Testing was conducted at Richardson, TX and was completed on 8 May 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

CONVEX Computer Corporation has submitted the following
Declaration of Conformance concerning the CONVEX Ada.

DECLARATION OF CONFORMANCE

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

Base Configuration

Base Compiler Name: CONVEX Ada   Version: Version 1.1
Host Architecture ISA: C210   OS&VER #: CONVEX Unix, Version 7.1
Target Architecture ISA: C210   OS&VER #: CONVEX Unix, Version 7.1

Derived Compiler Registration

Derived Compiler Name: CONVEX Ada   Version: Version 1.1
Host Architecture ISA: C201, C202, C120,   OS&VER #: CONVEX Unix, Version 7.1
       C220, C230, C240,
       C210i, C220i, C230i
Target Architecture ISA: Same as host   OS&VER #: Same as host

Implementor's Declaration

I, the undersigned, representing CONVEX Computer Corporation, 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 CONVEX Computer Corporation 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 the declaration shall be made only in the owner's corporate name.

Frank J. Marshall   Date: _May 5, 1989_
CONVEX Computer Corporation
Frank J. Marshall, Vice President

Owner's Declaration

I, the undersigned, representing CONVEX Computer Corporation, 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.

Frank J. Marshall   Date: _May 8, 1989_
CONVEX Computer Corporation
Frank J. Marshall, Vice President

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 CONVEX Ada, Version 1.1, as described in this Appendix, are provided by CONVEX Computer Corporation. 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 -2147483648 .. 2147483647;
    type SHORT_INTEGER is range -32768 .. 32767;
    type TINY_INTEGER is range -128 .. 127;

    type FLOAT is digits 15 range -8.988465567431157E+307 .. 8.988465567431157E+307
    type SHORT_FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;

    type DURATION is delta 6.103515625E-5
                                        range -1.31072E+5 .. 1.31071999938965E+5

    ...

end STANDARD;
```

Implementation-Dependent Pragmas

CONVEX Ada supports the following implementation-dependent pragmas:

DYNAMIC_SELECT

This pragma causes the compiler to generate multiple versions of the loop that immediately follows, based on trip counts supplied by the user. Up to four versions of the loop can be generated: scalar, vector, parallel, and parallel-outer/vector-inner. The compiler also generates code to allow runtime selection of which version to execute.

The DYNAMIC_SELECT pragma accepts three parameters, which specify the trip (iteration) count at which the compiler is to select vector, parallel, or parallel-vector execution. Each parameter may be an integer or one of the keywords DEFAULT or NONE. The compiler selects a version of the loop to execute based on the following rules:

- If the actual trip count is less than the minimum of the trip counts specified in the pragma, the loop runs scalar.
- If the actual trip count is greater than the maximum of the trip counts specified in the pragma, the loop runs in the mode corresponding to the maximum of the specified trip counts.
- In all other cases, the loop runs in the mode corresponding to the greatest trip count specified that the actual trip count exceeds.

If you omit one or more of the trip counts by entering DEFAULT instead of an integer, the compiler selects a default trip value for the test. If you use the keyword NONE instead of an integer, the compiler does not generate code for the corresponding mode.

The DYNAMIC_SELECT pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

EXTERNAL_NAME

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

FORCE_PARALLEL

This pragma tells the compiler that the iterations of the following loop are independent and that the loop should be parallelized. If both this pragma and FORCE_VECTOR precede a loop, the loop is vectorized and the strip-mine loop is parallelized.

The FORCE_PARALLEL pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

FORCE_VECTOR

This pragma tells the compiler that the iterations of the following loop are independent but that the loop should be vectorized rather than parallelized. If both this pragma and FORCE_PARALLEL precede a loop, the loop is vectorized and the strip-mine loop is parallelized.

The FORCE_VECTOR pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

IMPLICIT_CODE

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

INTERFACE

This pragma takes the name of a variable defined in another language and allows it to be referenced directly in Ada. The pragma replaces all occurrences of the variable name with an external reference to the second (*link_argument*).

The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar, array, record, or access type. The object may not be any of the following:

- A loop variable
- A constant
- An initialized variable

MAX_TRIPS

This pragma tells the compiler that the expected number of iterations (trips) for the following loop is approximately n. The compiler uses this information for profitability analysis, dynamic selection, and variable strip mining. The pragma accepts a single integer constant argument.

The MAX_TRIPS pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

NO_PARALLEL

This pragma prevents parallelization of the following loop but does not prevent vectorization. If both this pragma and the NO_VECTOR pragma precede a loop, the effect is the same as if the SCALAR pragma is used.

The NO_PARALLEL pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

NO_RECURRENCE

This pragma instructs the compiler to vectorize a loop even if the compiler cannot prove that there are no recurrence vector dependencies in the loop. If the loop does, in fact, contain recurrences and the NO_RECURRENCE pragma is specified, incorrect code may be generated.

The NO_RECURRENCE pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

NO_SIDE_EFFECTS

This pragma tells the compiler that the subprogram being compiled does not change the value of any global objects. The optimizer portion of the compiler can then move operations on such variables across calls to the subprogram.

The NO_SIDE_EFFECTS pragma must appear in the declarative part of a function or procedure.

NO_VECTOR

This pragma prevents vectorization of the following loop but does not prevent parallelization. If both this pragma and the NO_PARALLEL pragma precede a loop, the effect is the same as if the SCALAR pragma is used.

The NO_VECTOR pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

PARALLEL_UNIT

This pragma causes the compiler to attempt to generate code that will execute in parallel for the compilation unit in which the pragma occurs. Vectorization may also take place.

Only one SCALAR_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragma may appear in a library unit. Any subsequent SCALAR_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragmas are flagged as errors and ignored.

PREFER_PARALLEL

This pragma tells the compiler that is there is a choice of loops in a nest to parallelize and it is valid to parallelize the loop following the pragma, then it should be chosen for parallelization. All dependencies are honored.

The PREFER_PARALLEL pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

PREFER_VECTOR

This pragma tells the compiler that is there is a choice of loops in a nest to vectorize and it is valid to vectorize the loop following the pragma, then it should be chosen for vectorization. All dependencies are honored.

The PREFER_VECTOR pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

PSTRIP

This pragma tells the compiler that the parallel loop which follows should be strip mined with length n. This pragma reduces the overhead required to synchronize CPUs working together on the loop. The pragma increases to n the number of iterations each CPU picks up as it gets its next unit of work. The pragma accepts a single integer constant argument.

The PSTRIP pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

SCALAR

This pragma prevents vectorization of the loop that follows. The SCALAR pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

Loops nested within a loop that has the SCALAR pragma applied to it are eligible for vectorization.

SCALAR_UNIT

This pragma may appear any place in the declarative part of a library unit and tells the compiler that no vectorization is to be performed on the unit. The SCALAR_UNIT pragma overrides any optimization option specified on the compiler command line.

Only one SCALAR_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragma may appear in a library unit. Any subsequent SCALAR_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragmas are flagged as errors and ignored.

SHARE_CODE

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

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

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

The SHARE_CODE pragma may also be referenced as SHARE_BODY.

SPREAD_TASK

The SPREAD_TASK pragma may appear anyplace in the declarative part of a library unit and tells the compiler to use multiple CPUs (rather than one CPU) for tasking at runtime.

SYNCH_PARALLEL

This pragma tells the compiler, at optimization level -*O3*, that the loop that follows should be run in parallel even though it requires synchronization that may result in a significant loss of efficiency.

The SYNCH_PARALLEL pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

UNROLL

This pragma tells the compiler to attempt unrolling on the loop immediately following the pragma. Unrolling is performed only if the iteration count is less than 5.

The UNROLL pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

VECTOR_UNIT

This pragma may appear any place in the declarative part of a library unit and tells the compiler that library unit is a candidate for vectorization. Loops within the library unit are analyzed and those that have no recurrence vector dependencies are vectorized. The compiler can be forced to vectorize a loop, even if recurrence dependencies exist, with the NO_RECURRENCE pragma.

The VECTOR_UNIT pragma overrides any optimization option specified on the compiler command line. Only one SCALAR_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragma may appear in a library unit. Any subsequent SCALAR_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragmas are flagged as errors and ignored.

VSTRIP

This pragma tells the compiler that the vector loop immediately following the pragma should be strip mined with length *n*. This pragma allows the user to reduce strip-mine length, thus creating more iterations of the strip-mine loop so that it can be effectively parallelized. The pragma accepts a single integer constant argument.

The VSTRIP pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma is to apply to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

Implementation of Predefined Pragmas

CONVEX Ada implements the predefined pragmas as follows.

CONTROLLED

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

ELABORATE

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language*.

INLINE

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language*.

INTERFACE

This pragma supports calls to C and FORTRAN functions. Ada subprograms are either functions or procedures.

The types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS in SYSTEM. An optional third argument overrides the default link name. All parameters must have mode IN. Record and array objects can be passed by reference using the ADDRESS attribute.

LIST

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language*.

MEMORY_SIZE

This pragma is recognized by the compiler but has no effect. The implementation does not allow SYSTEM to be modified by pragmas; the SYSTEM package must be recompiled.

OPTIMIZE

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

PACK

This pragma causes the compiler to choose a nonaligned representation for composite types but does not cause objects to be packed at the bit level.

PAGE

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language*.

PRIORITY

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language*.

SHARED

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

STORAGE_UNIT

This pragma is recognized by the compiler but has no effect. The implementation does not allow SYSTEM to be modified by pragmas; the SYSTEM package must be recompiled.

SUPPRESS

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language* except that RANGE_CHECK and DIVISION_CHECK cannot be suppressed.

SYSTEM_NAME

This pragma is recognized by the compiler but has no effect. The implementation does not allow SYSTEM to be modified by pragmas; the SYSTEM package must be recompiled.

Implementation-Dependent Attributes

CONVEX Ada provides the implementation-dependent attribute P'REF, where P can represent an object, a program unit, a label, or an entry.

This attribute denotes the effective address of the first of the storage units allocated to P. For a subprogram, package, task unit, or label, it refers to the address of the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, it refers to the corresponding hardware interrupt.

This attribute is of type OPERAND as defined in the package MACHINE_CODE and is only allowed within a machine-code procedure. This attribute is not supported for a package, task unit, or entry.

Specification of the Package SYSTEM

The specification of the package SYSTEM is shown below. This specification is available online in the file *system.a* in the *standard* library.

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

     SYSTEM_NAME      : constant NAME := convex_unix;

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

     -- System-Dependent Named Numbers

     MIN_INT          : constant := -2_147_483_648;
     MAX_INT          : constant := 2_147_483_647;
     MAX_DIGITS       : constant := 15;
     MAX_MANTISSA     : constant := 31;
     FINE_DELTA       : constant := 2.0**(-31);
     TICK             : constant := 0.0001;

     -- Other System-dependent Declarations

     subtype PRIORITY is INTEGER range 0 .. 99;
     MAX_REC_SIZE : integer := 64*1024;

     type ADDRESS is private;

     NO_ADDR : constant ADDRESS;

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

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

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

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

private

type ADDRESS is new integer;
NO_ADDR : constant ADDRESS := 0;

end SYSTEM;
```

Restrictions on Representation Clauses

This section describes the restrictions on representation clauses in CONVEX Ada.

Size Specification

The size specification T'SMALL is not supported except when the representation specification is the same as the value 'SMALL for the base type.

Record Representation Clauses

Component clauses must be aligned on STORAGE_UNIT boundaries if the component exceeds 4 storage units.

Address Clauses

Address clauses are supported for variables and constants. An object cannot be initialized at the point of declaration if a subsequent address clause is applied to the object.

Interrupts

Interrupt entries are supported for UNIX signals. The Ada *for* clause gives the UNIX signal number.

Representation Attributes

The ADDRESS attribute is not supported for the following entities:

- Packages
- Tasks
- Labels
- Entries

Machine-Code Insertions

Machine-code insertions are supported. The general definition of the package MACHINE_CODE provides an assembly-language interface for the target machine. This package provides the necessary record types needed in the code statement, an enumeration type of all the opcode mnemonics, a set of register definitions, and a set of addressing-mode functions.

The general syntax of a machine-code statement is as follows:

```
CODE_n' (opcode, operand [,operand] );
```

The parameter n indicates the number of operands in the aggregate. A special case arises for a variable number of operands. The operands are listed within a subaggregate in the following format:

```
CODE_n' (opcode, (operand [,operand] ));
```

For those opcodes that require no operands, named notation must be used. The format is as follows:

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

The *opcode* must be an enumeration literal; it cannot be an object, attribute, or rename. An *operand* can only be an entity defined in the package MACHINE_CODE or with the 'REF attribute.

The arguments to any of the functions define in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE. The attribute 'REF may not be used as an argument in any of these functions. Inline expansion of machine-code procedures is supported.

Conventions for Implementation-Generated Names

There are no implementation-generated names.

Interpretation of Expressions in Address Clauses

Address clauses are supported for constants and variables. Interrupt entries are specified with the number of the UNIX signal.

Restrictions on Unchecked Conversions

There are no restrictions on unchecked conversions.

Restrictions on Unchecked Deallocations

There are no restrictions on unchecked deallocations.

Implementation Characteristics of I/O Packages

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITS) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as string, where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead.

MAX_REC_SIZE is defined in SYSTEM and can be changed by a program before DIRECT_IO is instantiated to provide an upper limit on the record size. In any case, the maximum size supported is 1024 x 1024 x STORAGE_UNIT bits. DIRECT_IO raises USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

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

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')

TEST PARAMETERS

Name and Meaning	Value

$BIG_ID4
    An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.
	(1..249 => 'A', 250 => '4',
	      251..499 => 'A')

$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..496 => '0', 497..499 => "298")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..494 => '0', 495..499 => "690.0")
$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.	2_147_483_647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	16_777_216
$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.	CONVEX_UNIX
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	(1..27 => "0_000_000_000_465_661_287_3", 28..43 => "07_739_257_812_5")
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2_147_483_647
$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.	99
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	(1..27 => "/no/such/directory/ILLEGAL_", 28..46 => "EXTERNAL_FILE_NAME1")
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	(1..27 => "THIS_FILE_NAME_IS_TOO_LONG_", 28..73 => "FOR_MY_SYSTEM", 74..288 => 'd')
$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.
	31

$MAX_DIGITS
    Maximum digits supported for
    floating-point types.
	15

$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.
	2_147_483_648

$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.	CONVEX_UNIX
$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.	16777216

TEST PARAMETERS

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.	CONVEX_UNIX
$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.	0.0001

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 43 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.

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

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

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

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

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

6. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

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

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

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

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

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

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

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

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

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

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

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

132, and 136).

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