UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	READ INSTRUCTIONS BEFORE COMPLETING FORM
4. TITLE (and Subtitle) **Ada Compiler Validation Summary Report:** Encore Computer Corporation, Encore Verdix Ada Development System Version 5.5, Encore Multimax 320 (Host & Target), 890727S1.10128	5. TYPE OF REPORT & PERIOD COVERED 27 July 1989 to 27 July 1990
	6. PERFOPMING ORG. REPORT NUMBER
7. AUTHOR(s) National Institute of Standards and Technology Gaithersburg, Maryland, USA	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS National Institute of Standards and Technology Gaithersburg, Maryland, USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS **Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081**	12. REPORT DATE
	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) National Institute of Standards and Technology Gaithersburg, Maryland, USA	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

UNCLASSIFIED

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)

Encore Computer Corporation, Encore Verdix Ada Development System, Version 5.5, Gaithersburg, MD, Encore Multimax 320 under Mach, Version 0.5 Beta (Host & Target), ACVC 1.10.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Compiler Validation Summary Report:

Compiler Name: Encore Verdix Ada Development System, Version 5.5

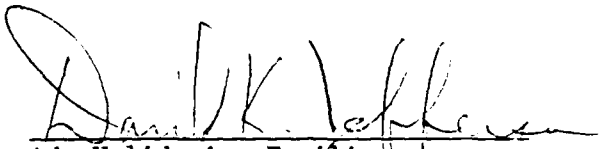
Certificate Number: 890727S1.10128

Host: Encore Multimax 320 under Mach, Version 0.5 Beta

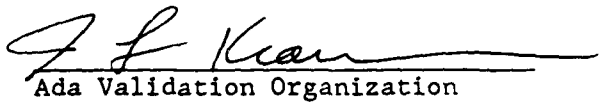
Target: Encore Multimax 320 under Mach, Version 0.5 Beta

Testing Completed July 27, 1989 Using ACVC 1.10

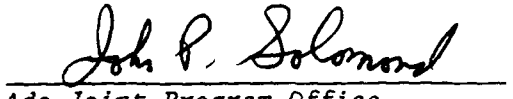
This report has been reviewed and is approved.



Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Computer Systems Laboratory (NCSL)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899



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



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

AVF Control Number: NIST89ENC560_2_1.10
DATE COMPLETE ON-SITE: 07-14-89
DATE REVISED: 08-11-89

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890727S1.10128
Encore Computer Corporation
Encore Verdix Ada Development System, Version 5.5
Encore Multimax 320 Host and Encore Multimax 320 Target

Completion of On-Site Testing:
July 27, 1989

Prepared By:
Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

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

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

On-site testing was completed July 27, 1989 at Marlborough, MA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

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

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

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

or from:

Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

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 which executes the code generated by the compiler.

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

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

1.5 ACVC TEST CLASSES

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

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

program executes to produce a PASSED message.

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

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

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

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

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

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some

of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

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

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

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

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

Compiler: Encore Verdix Ada Development System Version, 5.5

ACVC Version: 1.10

Certificate Number: 890727S1.10128

Host Computer:

Machine: Encore Multimax 320

Operating System: Mach, Version 0.5 Beta

Memory Size: 16MBytes

Target Computer:

Machine: Encore Multimax 320

Operating System: Mach, Version 0.5 Beta

Memory Size: 16MBytes

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. Universal integer calculations.

(1) An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64-bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

c. Predefined types.

(1) This implementation supports the additional predefined types SHORT_INTEGER, TINY_INTEGER, SHORT_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

d. 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) All 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 less precision than 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 for pre-defined integer comparison, pre-defined integer membership, large_int comparison, large_int membership, small_int comparison and no exception is raised for small_int membership. NUMERIC_ERROR/CONSTRAINT_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 by membership test "1.0E19 in LIKE_DURATION_M23" and "2.9E9 in MIDDLE_M3". (See test C45252A.)

(6) Underflow is not gradual. (See tests C45524A..Z (26 tests).)

e. Rounding.

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

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

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

(3) The method used for rounding to integer in static universal real expressions is round to even. (See test C4A014A.)

f. 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 an array type with INTEGER'LAST + 2 components. (See test C36202A.)

(3) NUMERIC_ERROR is raised when 'LENGTH is applied to an 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.)

g. Discriminated types.

(1) 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.)

(2) 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.)

h. Aggregates.

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

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

(3) CONSTRAINT_ERROR is raised 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.)

i. Pragmas.

(1) The pragma INLINE is supported for functions or procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

j. Generics.

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

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

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

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

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

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

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

(8) Generic non-library package bodies as subunits can be

compiled in separate compilations. (See test CA2009C.)

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

k. Input and output.

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

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

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

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

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

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

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

(8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

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

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

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

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

(13) More than one internal file can be associated with each external file for sequential files when writing or reading.

(See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

(14) More than one internal file can be associated with each external file for direct files when writing or reading. (See tests CE2107F..H (3 tests), CE2110D and CE2111H.)

(15) More than one internal file can be associated with each external file for text files when writing or reading. (See tests CE3111A..B, (2 tests), CE3111D..E (2 tests) and CE3114B.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 331 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 10 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	1134	1988	17	28	46	3342
Inapplicable	0	4	327	0	0	0	331
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	577	545	245	172	99	163	331	137	36	252	288	299	3342
Inapplicable	14	72	135	3	0	0	3	1	0	0	0	81	22	331
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

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

A39005G	B97102E	C97116A	BC3009B	CD2A62D	CD2A63A
CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B	CD2A66C
CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D	CD2A76A
CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G	CD2A84M
CD2A84N	CD2B15C	CD2D11B	CD5007B	CD5011O	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B	E28005C	ED7004B	ED7005C	ED7005D
ED7006C	ED7006D				

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, 331 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 (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)

C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

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..N (2 tests) and C45532M..N (2 tests) use fine 48-bit fixed-point base types which are not supported by this compiler.

e. C45531O..P (2 tests) and C45532O..P (2 tests) use coarse 48-bit fixed-point base types which are not supported by this compiler.

f. C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

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

h. CD1009C, CD2A41A..B, CD2A41E, CD2A42A..J (14 tests) are inapplicable because size clause are not supported for floating point types.

i. CD2A61I..J (2 tests) are inapplicable because SIZE clauses applied to array types does not imply compression of the component type when the component type is a composite or floating point type; an explicit SIZE clause on the component type is required.

j. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are inapplicable because SIZE clauses are not supported for access types. Access types are represented by machine addresses which are 32 bits on this architecture.

k. CD2A91A..E (5 tests) are inapplicable because size clauses are not supported for tasks. A task value is implemented as an address and addresses on this architecture are 32 bits.

l. CD5003B..H (7 tests), CD5011A..H (8 tests), CD5011L..M (2 tests), CD5011Q..R (2 tests), CD5012A..I (9 tests), CD5012L, CD5013B, CD5013D, CD5013F, CD5013H, CD5013L, CD5013N, CD5013R, CD5014T..X (5 tests) (total of 41 tests) are inapplicable because an address

clause with a dynamic address is applied to a variable requiring initialization.

m. CD5011N is inapplicable because address clauses for constants of access type are not supported.

n. CD5012J, CD5013S, and CD5014S are inapplicable because address clauses are not supported for tasks.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ac. CE2102W is inapplicable because this implementation supports RESET

with OUT_FILE mode for DIRECT_IO.

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

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

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

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

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

ai. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is not supported by this implementation.

aj. CE3115A is not applicable because RESETting of external files for MODE OUT_FILE is not supported.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

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

Modifications were required for 10 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

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 Encore Verdix Ada Development System Version, 5.5 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 Encore Verdix Ada Development System Version, 5.5 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:	Encore Multimax 320
Host operating system:	Umax 4.2, Version R3.3
Target computer:	Encore Multimax 320
Target operating system:	Mach, Version 0.5 Beta
Linker:	a.ld

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precision was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized on-site.

Split tests as supplied by Encore were checked against those corresponding tests from the magnetic tape using a UNIX utility, diff. No differences were found except those expected. The split tests as supplied by Encore were used in the validation.

TEST INFORMATION

The contents of the magnetic tape were loaded onto an Encore Multimax 320 where a utility read the ASCII tape and converted the tape contents to UNIX directory format. The tests were unpacked using UNPACK.ADA provided by the AVF. The unpacked tests were partitioned into appropriate directories. These directories with their files were then transferred via Ethernet to each of the other host/target hardwares (Encore Multimax 320 running different operating systems).

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the Encore Multimax 320. Results were printed from the Encore Multimax 320 computer.

The compiler was tested using command scripts provided by Encore Computer Corporation and reviewed by the validation team. See Appendix E for a complete listing of the compiler options for this implementation. The following compiler options were invoked:

```
-M   -w   -el
```

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.

3.7.3 Test Site

Testing was conducted at Marlborough, MA and was completed on July 27, 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Encore Computer Corporation has submitted the following Declaration of Conformance concerning the Encore Verdix Ada Development System, Version 5.5.

DECLARATION OF CONFORMANCE

Compiler Implementor: Encore Computer Corporation
Ada Validation Facility: NIST, Software Standards Validation Group
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name: Encore Verdix Ada Development System Version: 5.5

Host Architecture ISA:	Encore Multimax 320	OS&VER #:	Mach, Version 0.5 Beta
Target Architecture ISA:	Encore Multimax 320	OS&VER #:	Mach, Version 0.5 Beta

Implementor's Declaration

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

Date: 6/15/89

Encore Computer Corporation
Juem Juergens
Software Engineer

Owner's Declaration

I, the undersigned, representing Encore Computer Corporation, take full responsibility for the implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that the Ada language compiler listed, and its host/target performance is in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Date: 6/13/89

Encore Computer Corporation
Pin-Yee Chen
Vice President, Parallel Products

APPENDIX B

APPENDIX I 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 Encore Verdix Ada Development System Version, 5.5, as described in this Appendix, are provided by Encore 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_INTECER is range -32768 .. 32767;
 type TINY_INTEGER is range  -128 .. 127;

 type FLOAT is digits 15 range
      -1.79769313486231E+308 .. 1.79769313486231E+308;

 type SHORT_FLOAT is digits 6 range
      -3.40282346638529E+38 .. 3.40282346638529E+38;

 type DURATION is delta 6.10351562500000E-05 range
             -131072.0..131071.99993;

...

end STANDARD;
```

ATTACHMENT II

APPENDIX F IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Pre-validation Materials/Encore VADS Version 5.5

UMAX 4.2

1. IMPLEMENTATION-DEPENDENT PRAGMAS

INLINE_ONLY

This pragma, when used in the same way as pragma INLINE, indicates to the compiler that the subprogram must *always* be inlined. This pragma also suppresses the generation of a callable version of the routine which saves code space.

BUILT_IN

This pragma is used in the implementation of some predefined Ada packages, but provides no user access. It is used only to implement code bodies for which no actual Ada body can be provided, for example the MACHINE_CODE package.

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 will try 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 will get a unique copy of the generated code. The extent to which code is shared between instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit.

The name pragma SHARE_BODY is also recognized by the implementation and has the same effect as SHARE_CODE. It is included for compatibility with earlier versions of Encore VADS Ada.

NO_IMAGE

This pragma suppresses the generation of the image array used for the IMAGE attribute of enumeration types. This eliminates the overhead required to store the array in the executable image.

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.

INTERFACE_OBJECT

This pragma takes the name of a variable defined in another language and allows it to be referenced directly in Ada. The pragma will replace 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 or an access type. The object *cannot* be any of the following:

a loop variable,
a constant,
an initialized variable,
an array, or
a record.

IMPLICIT_CODE

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

2. PREDEFINED PRAGMAS

CONTROLLED

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

ELABORATE

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

INLINE

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

INTERFACE

This pragma supports calls to C and FORTRAN functions. The Ada subprograms can be 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 Ada RM.

MEMORY_SIZE

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

OPTIMIZE

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

PACK

This pragma will cause the compiler to minimize gaps between components in the representation of composite types. For arrays, components will only be packed to bit sizes corresponding to powers of 2, if the field is smaller than STORAGE_UNIT bits. Objects larger than STORAGE_UNIT are packed to the nearest STORAGE_UNIT. level.

PAGE

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

PRIORITY

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

SHARED

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

STORAGE_UNIT

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

SUPPRESS

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

SYSTEM_NAME

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

3. IMPLEMENTATION-DEPENDENT ATTRIBUTES

The attribute REF has two forms: X'REF and SYSTEM.ADDRESS(N):

In X'REF, X must be a constant, variable, procedure, function, or label. The attribute returns a value of the type MACHINE_CODE.OPERAND and may only be used to designate an operand within a code statement.

In SYSTEM.ADDRESS(N), SYSTEM.ADDRESS must be of the type SYSTEM.ADDRESS. N must be an expression of type UNIVERSAL_INTEGER. The attribute returns a value of type SYSTEM.ADDRESS, which represents the address designated by N (this is similar to the effect of an unchecked conversion from integer to address except N must be static).

4. SPECIFICATION OF PACKAGE SYSTEM

```
package SYSTEM
is

    type NAME is ( UMAX_4_2 );

    SYSTEM_NAME      : constant NAME := UMAX_4_2;

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

      -- System-Dependent Named Numbers

    MIN_INT          : constant := -2_147_483_647 - 1;
    MAX_INT          : constant :=  2_147_483_647;
    MAX_DIGITS       : constant :=         15;
```

```
        MAX_MANTISSA    : constant := 31;
        FINE_DELTA      : constant := 2.0**(-31);
        TICK            : constant := 0.01;

          -- Other System-dependent Declarations

        subtype PRIORITY is INTEGER range 0 .. 99;

        MAX_REC_SIZE : integer := 64*1024;

          type ADDRESS is private;

          NO_ADDR: constant ADDRESS;

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

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

          pragma inline(PHYSICAL_ADDRESS);
          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);

    private

          type ADDRESS is new INTEGER;
          no_addr: constant address := 0;

    end SYSTEM
```

Attributes of the pre-defined type DURATION

```
first          -131072.00000
last            131071.99993
size                32
delta            6.10351562500000E-05
mantissa            31
small            6.10351562500000E-05
large            1.31071999938964E+05
fore                 7
aft                  5
safe_small       6.10351562500000E-05
safe_large       1.31071999938964E+05
machine_rounds  TRUE
machine_overflows TRUE
```

Attributes of type FLOAT

```
first          -1.79769313486231E+308
last            1.79769313486231E+308
size                64
digits              15
mantissa            51
epsilon          8.88178419700125E-16
emax               204
small            1.94469227433160E-62
large            2.57110087081438E+61
safe_emax           1021
safe_small       2.22507385850720E-308
safe_large       2.24711641857789E+307
machine_radix           2
machine_mantissa           53
machine_emax          1024
machine_emin          -1021
machine_rounds  TRUE
machine_overflows TRUE
```

Attributes of type SHORT_FLOAT

```
first          -3.40282346638529E+38
last            3.40282346638529E+38
size                32
digits               6
mantissa            21
```

```
epsilon          9.53674316406250E-07
emax                   84
small            2.58493941422821E-26
large            1.93428038904620E+25
safe_emax               125
safe_small       1.17549435082228E-38
safe_large       4.25352755827077E+37
machine_radix            2
machine_mantissa                24
machine_emax           128
machine_emin          -125
machine_rounds  TRUE
machine_overflows TRUE
```

Ranges of predefined integer types

```
TINY_INTEGER   -128 .. 127
SHORT_INTEGER          -32768 .. 32768
INTEGER           -2147483648 .. 2147483647
```

Default STORAGE_SIZE (collection size) for access types

100000

Priority range is 0 .. 99

Default STORAGE_SIZE for tasks is

10240

If tasks need larger stack sizes, the 'STORAGE_SIZE attribute may be used with the task type declaration.

Attributes and time-related numbers

```
Duration'small              9.76562500000000E-04
System.tick          1.00000000000000E-02
```

6. RESTRICTIONS ON REPRESENTATION CLAUSES

Pragma PACK

See section (2) above.

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.

7. RECORD REPRESENTATION CLAUSES

Component clauses must be aligned on STORAGE_UNIT boundaries.

Address Clauses

Address clauses are supported for objects and entries.

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
Entries

8. 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. It provides the necessary record type(s) 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*});

where 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. The format is as follows:

CODE_N'(*opcode*, (*operand* {, *operand*}));

For those opcodes that require no operands, named notation must be used (cf. RM 4.3(4)).

CODE_0'(op => *opcode*);

The *opcode* must be an enumeration literal (i.e. it cannot be an object, attribute, or a rename).

An *operand* can only be an entity defined in MACHINE_CODE or the 'REF attribute.

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

Inline expansion of machine code procedures is supported.

9. CONVENTIONS FOR IMPLEMENTATION-GENERATED NAMES

There are no implementation-generated names.

10. 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.

11. RESTRICTIONS ON UNCHECKED CONVERSIONS

None.

12. RESTRICTIONS ON UNCHECKED DEALLOCATIONS

None.

13. 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_RECORD_SIZE is defined in SYSTEM and can be changed by a program before instantiating DIRECT_IO to provide an upper limit on the record size. In any case, the maximum size supported is 64 * 1024 bytes. DIRECT_IO will raise USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size

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

14. IMPLEMENTATION LIMITS

The following limits are actually enforced by the implementation. It is not intended to imply that resources up to or even near these limits are available to every program.

Line Length

The implementation supports a maximum line length of 499 characters not including the end of line character.

Record and Array Sizes

The maximum size of a statically sized array type is 4,000,000 x STORAGE_UNITS. The maximum size of a statically sized record type is 4,000,000 x STORAGE_UNITS. A record type or array type declaration that exceeds these limits will generate a warning message.

Default Stack Size for Tasks

In the absence of an explicit STORAGE_SIZE length specification every task except the main program is allocated a fixed size stack of 10,240 STORAGE_UNITS. This is the value returned by T'STORAGE_SIZE for a task type T.

Default Collection Size

In the absence of an explicit STORAGE_SIZE length attribute the default collection size for an access type is 100,000 STORAGE_UNITS. This is the value returned by T'STORAGE_SIZE for an access type T.

Limit on Declared Objects

There is an absolute limit of 6,000,000 x STORAGE_UNITS for objects declared statically within a compilation unit. If this value is exceeded the compiler will terminate the compilation of the unit with a FATAL error message.

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.

$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	1..498 => 'A', 499 => '1'
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	1..498 => 'A', 499 => '2'
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	1..127 => 'A', 128 => '3', 129..499 => 'A'
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	1..127 => 'A', 128 => '4', 129..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..252 => '0', 253..499 => '298'
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the	1..250 => '0', 251..499 => '690.0'

maximum line length.

$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	1..195 => "A"
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	196..498 => "127, 499 => "1"
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	1..235 => ' '
$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
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	UMAX_V
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2.0**(-31)
$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	NO_SUCH_FLOATING_TYPE

FLOAT, SHORT_FLOAT, or LONG_FLOAT.

$GREATER_THAN_DURATION 2147484.0
A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.

$GREATER_THAN_DURATION_BASE_LAST 2147484.0
A universal real literal that is greater than DURATION'BASE'LAST.

$HIGH_PRIORITY 99
An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.

$ILLEGAL_EXTERNAL_FILE_NAME1 1..511 => a, 512 => 1
An external file name which contains invalid characters.

$ILLEGAL_EXTERNAL_FILE_NAME2 1..511 => b, 512 => 2
An external file name which is too long.

$INTEGER_FIRST -2147483648
A universal integer literal whose value is INTEGER'FIRST.

$INTEGER_LAST 2147483647
A universal integer literal whose value is INTEGER'LAST.

$INTEGER_LAST_PLUS_1 2147483648
A universal integer literal whose value is INTEGER'LAST + 1.

$LESS_THAN_DURATION -2147484.0
A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.

$LESS_THAN_DURATION_BASE_FIRST -2147484.0
A universal real literal that is less than DURATION'BASE'FIRST.

$LOW_PRIORITY 0
An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.

$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..250 => '0', 251..499 => '11:'
$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..2 => '16:', 3..248 => '0' 249..499 => '16: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.	UMAX_V
$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.	16_777_216
$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.	UMAX512_V
$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 inout parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

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

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

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

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

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).

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).

CD2A63A..D, CD2A66A..D, CD2A73A..D, 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.

CD2A81G, CD2A83G, CD2A84M & N, & CD50110
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, 86 & 96, and 58, resp.).

CD2B15C & 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.

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.

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

ED7004B, ED7005C & D, 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.

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).

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

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.

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)

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

CE3301A
This test contains several calls to END_OF_LINE & 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, & 136).

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.

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 that must appear at the top of the page.

APPENDIX E

COMPILER OPTIONS AS SUPPLIED BY

Encore Computer Corporation

Compiler: Encore Verdix Ada Development System Version, 5.5

ACVC Version: 1.10

NAME

ada - Ada compiler

SYNTAX

ada [options] [ada_source.a]... [linker_options] [object_file.o]...

DESCRIPTION

The command ada executes the Ada compiler and compiles the named Ada source file, ending with the .a suffix. The file must reside in a VADS library directory. The ada.lib file in this directory is modified after each Ada unit is compiled.

The object for each compiled Ada unit is left in a file with with the same name as that of the source with .01, .02, etc. substituted for .a. The -o option can be used to produce an executable with a name other than a.out, the default. For cross compilers, the default name is a.vox.

By default, ada produces only object and net files. If the -M option is used, the compiler automatically invokes a.ld and builds a complete program wih the named library unit as the main program.

Non-Ada object files (.o files produced by a compiler for another language) may be given as arguments to ada. These files will be passed on to the linker and will be linked with the specified Ada object files.

Command line options may be specified in any order, but the order of compilation and the order of the files to be passed to the linker can be significant.

Several VADS compilers may be simultaneously available on a single system. Because the ada command in any VADS_location/bin on a system will execute the correct compiler components based upon visible library directives, the option -sh is provided to print the name of the components actually executed.

Program listings with a disassembly of machine instructions are generated by a.db or a.das.

OPTIONS

-a file_name	(archive) treat file_name as an ar file. Since archive files end with .a, -a is used to distinguish archive files from Ada source files.
-d	(dependencies) analyze for dependencies only. Do not do semantic analysis or code generation. Update the library, marking any defined units as uncompiled. The -d option is used by a.make to establish dependencies among new files.
-e	(error) process compilation error messages using a.error and direct it to stdout.-only the source lines containing errors are listed. Only one -e or -E option should be used.
-E	
-E file	
-E directory	(error output) without a file or directory argument, ada processes error messages using a.error and directs the output to stdout; the raw error messages are left in ada_source.err. If a file pathname is given, the raw error messages are placed in that file. If a directory argument is supplied, the raw error output is placed in dir/source.err. Only one -e or -E option should be used.
-el	(error listing) intersperse error messages among source lines and direct to stdout.
-El	
-El file	

-El directory (error listing) same as the -E option, except that source listing with errors is produced.

-ev (error vi) process syntax error messages using a.error, embed them in the source file, and call the environment editor ERROR_EDITOR. (If ERROR_EDITOR is defined, the environment variable ERROR_PATTERN should also be defined. ERROR_PATTERN is an editor search command that locates the first occurrence of '###' in the error file.) If no editor is specifed, call vi.

-lfile_abbreviation
(link) Link this library file. (Do not space between the -l and the file abbreviation.) See also Operating system documentation, ld(1).

-M unit_name (main) produce an executable program using the named unit as the main program. The unit must be either a parameterless procedure or a parameterless function returning an integer. The executable program will be left in the file a.out unless overridden with the -o option.

-M ada_source.a (main) like *-M unit_name, except that the unit name is assumed to be the root name of the* .a file (for foo.a the unit is foo). Only one .a file may be preceded by -M.

-o executable_file
(output) this option is to be used in conjunction with the -M option. executable_file is the name of the executable rather than the default a.out.

-O[0-9] (optimize) invoke the code optimizer (no space before the digit). An optional digit limits the number of passes by the optimizer; without the -O option, one pass is made; -O0 prevents optimization; -O with no digit optimizes as far as possible.

-R VADS_library
(recompile instantiation) force analysis of all generic instantiations, causing reinstantiation of any that are out of date.

-S (suppress) apply pragma SUPPRESS to the entire compilation for all suppressible checks.

-T (timing) print timing information for the compilation.

-v (verbose) print compiler version number, date and time of compilation, name of file compiled, command input line, total compilation time, and error summary line.

-w (warnings) suppress warning diagnostics.

SEE ALSO

[VADS Reference] a.db, a.error, a.ld, a.mklib, a.das and Operating system documentation, ld(1)

DIAGNOSTICS

The diagnostics produced by the VADS compiler are intended to be self-explanatory. Most refer to the RM. Each RM reference includes a section number and optionally, a paragraph number enclosed in parentheses.