UNCLASSIFIED

DTIC FILE COPY

(2)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A215 182

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Apollo Computer Inc., Domain/Ada, V3.0m, DN3500 (Host & Target), 890719W1.10125	5. TYPE OF REPORT & PERIOD COVERED 19 July 1989 to 19 July 1990 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson AFB Dayton, OH, USA	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson AFB Dayton, OH, USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson AFB Dayton, OH, USA	15. SECURITY CLASS (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

UNCLASSIFIED

DTIC ELECTE DEC 04 1989 S B D

18. SUPPLEMENTARY NOTES

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

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

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

Apollo Computer Inc., Domain/Ada, V3.0m, Wright-Patterson AFB, DN3500 under Domain/OS SR10.1.0.2 (Host & Target), ACVC 1.10.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

89 11 30 058

AVF Control Number: AVF-VSR-305.0889
89-03-20-ACI

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890719W1.10125
Apollo Computer Inc.
Domain/Ada, V3.0m
DN3500

Completion of On-Site Testing:
19 July 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: Domain/Ada, V3.0m

Certificate Number: #890719W1.10125

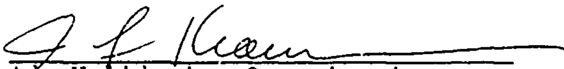
Host: DN3500 under
Domain/OS, SR10.1.0.2

Target: DN3500 under
Domain/OS, SR10.1.0.2

Testing Completed 19 July 1989 Using ACVC 1.10

This report has been reviewed and is approved.

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



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

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

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT 1-2
1.2 USE OF THIS VALIDATION SUMMARY REPORT 1-2
1.3 REFERENCES. 1-3
1.4 DEFINITION OF TERMS 1-3
1.5 ACVC TEST CLASSES 1-4

CHAPTER 2 CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED. 2-1
2.2 IMPLEMENTATION CHARACTERISTICS. 2-2

CHAPTER 3 TEST INFORMATION

3.1 TEST RESULTS. 3-1
3.2 SUMMARY OF TEST RESULTS BY CLASS. 3-1
3.3 SUMMARY OF TEST RESULTS BY CHAPTER. 3-2
3.4 WITHDRAWN TESTS 3-2
3.5 INAPPLICABLE TESTS. 3-2
3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS. . 3-5
3.7 ADDITIONAL TESTING INFORMATION. 3-6
3.7.1 Prevalidation 3-6
3.7.2 Test Method 3-6
3.7.3 Test Site 3-7

APPENDIX A DECLARATION OF CONFORMANCE

APPENDIX B APPENDIX F OF THE Ada STANDARD

APPENDIX C TEST PARAMETERS

APPENDIX D WITHDRAWN TESTS

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

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 19 July 1989 at Chelmsford 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:

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

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

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

1.3 REFERENCES

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

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

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

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

1.4 DEFINITION OF TERMS

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

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

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

Applicant The agency requesting validation.

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

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

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

cross-compilers, translators, and interpreters.

Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

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

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

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

illegal construct that it contains is detected by the compiler.

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

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

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

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

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

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

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

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

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

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

Compiler: Domain/Ada, V3.0m

ACVC Version: 1.10

Certificate Number: 890719W1.10125

Host Computer:

Machine:	DN3500
Operating System:	Domain/OS SR10.1.0.2
Memory Size:	8 Megabytes

Target Computer:

Machine:	DN3500
Operating System:	Domain/OS SR10.1.0.2
Memory Size:	8 Megabytes

Communications Network: Domain Ring

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 type TINY_INTEGER in package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

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

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

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

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

(4) Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

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

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

d. Rounding.

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

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

(2) The method used for rounding to longest integer is round to even. (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 array type with INTEGER'LAST + 2 components, with each component being a null array. (See test C36202A.)

(3) NUMERIC_ERROR is raised when 'LENGTH is applied to a array type with SYSTEM.MAX_INT + 2 components, with each component being a null array. (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.)

(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, 44 tests had been withdrawn because of test errors. The AVF determined that 327 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 8 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	1133	1993	17	28	46	3346
Inapplicable	0	5	322	0	0	0	327
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	162	331	137	36	252	293	299	3346
Inappl	14	72	135	3	0	0	4	1	0	0	0	76	22	327
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

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

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

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

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 327 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 is not applicable because this implementation does not support a 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, when the specified size implies compression of composite of floating point components. 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) are not applicable because this implementation does not support size clauses for task types.

m. The following 41 tests are not applicable because an address clause with a dynamic address is applied to a variable requiring initialization:

CD5003B..H (7 tests)	CD5011A..H (8 tests)	CD5011L..M (2 tests)
CD5011Q..R (2 tests)	CD5012A..I (9 tests)	CD5014T..X (5 tests)
CD5012L	CD5013B	CD5013D
CD5013F	CD5013H	CD5013L
CD5013N	CD5013R	

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

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 this implementation supports CREATE with IN_FILE mode for text files.

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

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

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

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

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

aj. 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 8 tests.

The following 8 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
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 Domain/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 Domain/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:	DN3500
Host operating system:	Domain/OS, SR10.1.0.2
Target computer:	DN3500
Target operating system:	Domain/OS, SR10.1.0.2
Compiler:	Domain/Ada, V3.0m

The host and target computers were linked via Domain Ring.

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 onto a DN590 and made available to the host computers via Domain Ring.

The full set of tests was compiled, linked, and all executable tests were run on 4 DN3500 computers. Results were printed from a DN4000 via Domain Ring.

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

OPTION	EFFECT
-el	Intersperse compiler error messages among source lines and display on standard output.

Tests were compiled, linked, and executed (as appropriate) using 4 computers. 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 Chelmsford MA and was completed on 19 July 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Apollo Computer, Inc. has submitted the following Declaration of Conformance concerning the Domain/Ada.

DECLARATION OF CONFORMANCE

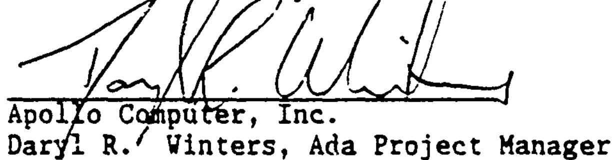
Compiler Implementor: Apollo Computer, Inc.
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name: Domain/Ada, V3.0m
Host Architecture ISA: DN3500
Host OS and Version: Domain/OS, SR10.1.0.2
Target Architecture ISA: DN3500
Target OS and Version: Domain/OS, SR10.1.0.2

Implementor's Declaration

I, the undersigned, representing Apollo Computer, Inc., 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 Apollo Computer, Inc. is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

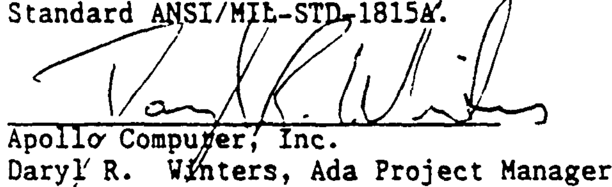


Date: 7/19/89

Apollo Computer, Inc.
Daryl R. Winters, Ada Project Manager

Owner's Declaration

I, the undersigned, representing Apollo Computer, Inc., 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.



Date: 7/19/89

Apollo Computer, Inc.
Daryl R. Winters, Ada Project Manager

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 Domain/Ada, V3.0m, as described in this Appendix, are provided by Apollo Computer, Inc. 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 -1.79769313486231E+308
                                 .. 1.79769313486231E+308;
   type SHORT_FLOAT is digits 6 range -3.40282E+38 .. 3.40282E+38;

   type DURATION is delta 1.0E-3range -2147483.648 .. 2147483.647;

   ...

end STANDARD;
```

Appendix F

Implementation–Dependent Characteristics

This appendix summarizes the Domain/Ada MC680*X*0 implementation–dependent characteristics as required by Appendix F of the *Ada Reference Manual* (RM). In particular, this appendix

- Lists the Domain/Ada pragmas and attributes
- Gives the specification for the **package** SYSTEM
- Lists the restrictions on representation clauses and unchecked type conversions
- Gives the naming conventions for denoting implementation–dependent components in record representation clauses
- Gives the interpretations of expressions in address clauses
- Presents the implementation–dependent characteristics of I/O packages
- Presents any additional implementation–dependent features

The following sections summarize each of these topics. Throughout these sections, we use the notation MC680*X*0 to mean the MC68020 and the MC68030 microprocessors only.

F.1 Implementation–Dependent Pragmas and Attributes

This section details the Domain/Ada MC680*X*0 implementation–dependent pragmas and attributes.

F.1.1 Implementation-Dependent Pragmas

Domain/Ada provides the following pragmas in its MC680X0 implementation. Some of the entries in this list refer you to the chapter or section in this document where you can find additional information about the pragma.

- **pragma** EXTERNAL_NAME allows you to specify a *link_name* for an Ada variable or subprogram so that you can reference the object from programs written in other languages. (Refer to Section 10.1.3 for more information.)

- **pragma** IMPLICIT_CODE specifies that implicit code generated by the compiler is allowed (ON) or disallowed (OFF). You can use this pragma only within the declarative part of a machine code procedure. (Refer to Section 9.9.3 for more information.)

- **pragma** INLINE_ONLY, when used in the same way as **pragma** INLINE, indicates to the compiler that the subprogram must *always* be inlined. (This is very important for some code procedures.) This pragma also suppresses the generation of a callable version of the routine, which saves code space.

- **pragma** INTERFACE_NAME allows you to reference variables and subprograms defined in another language from your Ada programs, replacing all occurrences of a *ada_subprogram_or_object_name* with an external reference to a *link_name* in the object file. The **pragma** INTERFACE_NAME, used in conjunction with **pragma** INTERFACE, allows you to specify the exact name of the subprogram being called by providing an optional linker name for the subprogram. This optional linker name enables you to call a subprogram defined in another language whose name contains characters that are not allowed in an Ada identifier (Refer to Section 10.3 for more information.)

- **pragma** NO_IMAGE 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.

- **pragma** NOT_ELABORATED, which is allowed only within a package specification, suppresses elaboration checks for all entities defined within a package, including the package specification itself. In addition, this pragma suppresses the generation of elaboration code. When using **pragma** NOT_ELABORATED, you must ensure that there are no entities defined in your program that require elaboration.

- **pragma** SHARE_CODE provides for the sharing of object code between multiple instantiations of the same generic procedure or package body. A "parent" instantiation is created and subsequent instantiations of the same types can share the parent's object code, reducing program size and compilation times. You can use the name **pragma** SHARE_BODY instead of SHARE_CODE with the same effect. (Refer to Section 13.2 for more information.)

In addition to the pragmas mentioned in the previous list, the Domain/Ada MC680X0 implementation expands upon the functionality of the following predefined language pragmas:

- **pragma** INLINE is implemented as described in Appendix B of the RM with the addition that you can expand recursive calls up to the maximum depth of 8. The compiler produces warnings for nestings that are too deep or for bodies that are not available for inline expansion.

- **pragma** PACK causes the compiler to minimize gaps between components in the representation of composite types. For arrays, the compiler packs components to bit sizes corresponding to powers of 2 (if the field is smaller than STORAGE_UNIT bits). The compiler packs objects larger than a single STORAGE_UNIT to the nearest STORAGE_UNIT.

- **pragma** SUPPRESS is supported in the single parameter form. The pragma applies from the point of occurrence to the end of the innermost enclosing block. You cannot suppress DIVISION_CHECK. The double parameter form of the pragma with a name of an object, type, or subtype is recognized, but has no effect in the current release. (Refer to Section 9.9.2 for more information.) You can use this pragma to suppress elaboration checks on any compilation unit except a package specification.

The Domain/Ada MC680X0 implementation recognizes the following pragmas, but they have no effect in the current release:

- **pragma** CONTROLLED

- **pragma** MEMORY_SIZE

- **pragma** OPTIMIZE (Refer to the **ada** -O option for code optimization in Chapter 4.)

- **pragma** SHARED

- **pragma** STORAGE_UNIT (This implementation does not allow you to modify **package** SYSTEM by means of pragmas. However, you can achieve the same effect by recompiling **package** SYSTEM with altered values.)

- **pragma** SYSTEM_NAME (This implementation does not allow you to modify **package** SYSTEM by means of pragmas. However, you can copy the file **system.a** from the STANDARD library to a local Domain/Ada library and recompile the file there with the new values.)

The following pragmas are implemented as described in Appendix B of the RM:

- **pragma** ELABORATE

- **pragma** INTERFACE (supports C and FORTRAN only)

- **pragma** LIST
- **pragma** PAGE
- **pragma** PRIORITY

F.1.2 Implementation-Defined Attribute: 'REF

The Domain/Ada MC680*X*0 implementation provides one implementation-defined attribute, **'REF**. You can use this attribute in one of two ways: *X*'**REF** and SYSTEM.ADDRESS'**REF**(*N*). You can use *X*'**REF** only in machine code procedures while you can use **SYSTEM.ADDRESS'REF**(*N*) anywhere that you want to convert an integer expression to an address.

F.1.2.1 *X*'REF

The *X*'**REF** attribute generates a reference to the entity to which it is applied.

In *X*'**REF**, *X* must be either a constant, variable, procedure, function, or label. The attribute returns a value of the **type** MACHINE_CODE.OPERAND, which you can use only to designate an operand within a machine code-statement.

You can precede the instruction generated by the code-statement in which the attribute occurs by additional instructions needed to facilitate the reference (for example, loading a base register). If the declarative section of the procedure contains **pragma** IMPLICIT_CODE (OFF), the compiler will generate a warning if additional code is required.

References can also cause the generation of run-time checks. You can use pragma SUPPRESS to eliminate these checks.

```
CODE_1'(JSR, PROC'REF);
CODE_2'(MOVE_L, X.ALL(Z)'REF, DO);
```

For more information on machine code insertions, refer to Chapter 9.

F.1.2.2 SYSTEM.ADDRESS'REF(*N*)

The effect of **SYSTEM.ADDRESS'REF**(*N*) is similar to the effect of an unchecked conversion from integer to address. However, you should use this attribute instead of an unchecked conversion in the following circumstances (in these circumstances, *N* must be static):

- Within any of the run-time configuration packages:
  Use of unchecked conversion within an address clause would require the generation of elaboration code, but the configuration packages are not elaborated.

- In any instance where N is greater than INTEGER'LAST:
  Such values are required in address clauses that reference the upper portion of memory. To use unchecked conversion in these instances would require that the expression be given as a negative integer.

- To place an object at an address, use the '**REF** attribute:
  The *integer_value,* in the following example, is converted to an address for use in the address clause representation specification. The form avoids UNCHECKED_CONVERSION and is also useful for 32-bit unsigned addresses.

```
--place an object at an address
for object use at ADDRESS'REF (integer_value)

--to use unsigned addresses
for VECTOR use at SYSTEM.ADDRESS'REF(16#808000d0#);
TOP_OF_MEMORY: SYSTEM.ADDRESS:= SYSTEM.ADDRESS'REF(16#FFFFFFFF#);
```

In **SYSTEM.ADDRESS**'REF(*N*), **SYSTEM.ADDRESS** must be 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*.

F.2 Specification of the Package SYSTEM

```
package SYSTEM is
        type NAME is ( apollo_4_3_unix );

        SYSTEM_NAME         : constant NAME := apollo_4_3_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 := 0.01;

        -- 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;
                      NCR: INTEGER) return ADDRESS renames INCR_ADDR;
        function "-"(A: ADDRESS;
                      DECR: INTEGER) return ADDRESS renames DECR_ADDR;

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

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

end SYSTEM;
```

F.3 Restrictions on Representation Clauses and Unchecked Type Conversions

This section summarizes the restrictions on representation clauses and unchecked type conversions for the MC680*X*0 implementation of Domain/Ada.

We describe the representation clauses that Domain/Ada supports in Chapter 13.

F.3.1 Representation Clauses

The Domain/Ada MC680*X*0 implementation supports bit level, length, enumeration, size, and record representation clauses. Size clauses are not supported for tasks, floating-point types, access types, or array types. This implementation supports address clauses for objects except for task objects and for initialized objects given dynamic addresses. Address clauses for task entries are supported; the specified value is a UNIX signal value.

The only restrictions on record representation clauses are the following:

- If a component does not start and end on a storage unit boundary, it must be possible to get the component into a register with one move instruction. On a MC680*X*0 machine, where longwords start on even bytes, the component must fit into 4 bytes starting on a word boundary.
- A component that is itself a record must occupy a power of 2 bits. Components that are of a discrete type or packed array can occupy an arbitrary number of bits subject to the previously mentioned restrictions.

F.3.2 Unchecked Type Conversions

This implementation of Domain/Ada supports the generic function UNCHECKED_CONVERSION with the following restriction:

- You cannot instantiate the predefined generic function UNCHECKED_CONVERSION with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

F.4 Naming Conventions for Denoting Implementation-Dependent Components in Record Representation Clauses

Record representation clauses are based on the target machine's word, byte, and bit order numbering so that Domain/Ada is consistent with various machine architecture manuals. Bits within a STORAGE_UNIT are also numbered according to the target machine manu-

als. This implementation of Domain/Ada does not support the allocation of implementation-dependent components in records.

F.5 Interpretations of Expressions in Address Clauses

This implementation of Domain/Ada supports the **SYSTEM.ADDRESS'REF(*N*)** summarized in Section F.1.2.2.

F.6 Implementation-Dependent Characteristics of I/O Packages

The Ada I/O system is implemented using Domain/OS I/O. Both formatted I/O and binary I/O are available. There are no restrictions on the types with which you can instantiate DIRECT_IO and SEQUENTIAL_IO except that the element size must be less than a maximum given by the variable SYSTEM.MAX_REC_SIZE. You can set this variable to any value prior to the generic instantiation; thus, you can use any element size. You can instantiate DIRECT_IO with unconstrained types, but each element will be padded out to the maximum possible for that type or to SYSTEM.MAX_REC_SIZE, whichever is smaller. No checking—other than normal static Ada type checking—is done to ensure that values from files are read into correctly sized and typed objects.

Domain/Ada file and terminal input/output are identical in most respects and differ only in the frequency of buffer flushing. Output is buffered (buffer size is 1024 bytes) and the buffer is flushed after each write request if the destination is a terminal.

The procedure FILE_SUPPORT.ALWAYS_FLUSH (*file_ptr*) will cause flushing of the buffer associated with *file_ptr* after all subsequent output requests. Refer to the source code for **file_spprt_b.a** in the **standard** library for more information.

F.6.1 Instantiations of DIRECT_IO

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as a string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. You can change MAX_REC_SIZE (defined in package SYSTEM) before instantiating DIRECT_IO to provide an upper limit on the record size. The maximum size supported is 1024 * 1024 * STORAGE_UNIT bits. DIRECT_IO will raise USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

F.6.2 Instantiations of SEQUENTIAL_IO

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. You can change MAX_REC_SIZE (defined in **package** SYSTEM) before instantiating INTEGER_IO to provide an upper limit on the record size. SEQUENTIAL_IO imposes no limit on MAX_REC_SIZE.

F.7 Additional Implementation–Dependent Features

This section details any other features that are specific to the Domain/Ada MC680*X*0 implementation.

F.7.1 Restrictions on 'Main' Programs

Domain/Ada requires that a 'main' program must be a non–generic subprogram that is either a procedure or a function returning an Ada STANDARD.INTEGER (the predefined type). In addition, a 'main' program cannot be an instantiation of a generic subprogram

F.7.2 Generic Declarations

Domain/Ada does not require that a generic declaration and the corresponding body be part of the same compilation, and they are not required to exist in the same Domain/Ada library. The compiler generates an error if a single compilation contains two versions of the same unit.

F.7.3 Implementation–Dependent Portions of Predefined Ada Packages

Domain/Ada supplies the following predefined Ada packages given by the Ada RM C(22) in the **standard** library:

- **package** STANDARD
- **package** CALENDAR
- **package** SYSTEM
- **generic procedure** UNCHECKED_DEALLOCATION
- **generic function** UNCHECKED_CONVERSION
- **generic package** SEQUENTIAL_IO
- **generic package** DIRECT_IO
- **package** TEXT_IO

- **package** IO_EXCEPTIONS
- **package** LOW_LEVEL_IO
- **package** MACHINE_CODE

The implementation-dependent portions of the predefined Ada packages define the following types and objects:

<in **package** STANDARD>

```
type BOOLEAN is          <8-bit, byte>;
type TINY_INTEGER is     <8-bit, byte integer>;
type SHORT_INTEGER is    <16-bit, word integer>;
type INTEGER is          <32-bit, longword integer>;
type SHORT_FLOAT is      <6-digit, 32-bit, float>;
type FLOAT is            <15-digit, 64-bit, float>;
type DURATION is     delta 1.00000000000000E-03 range -2147483.648 .. 2147483 647,
```

<in **package** DIRECT_IO>

```
type COUNT is            range 0 .. 2_147_483_647;
```

<in **package** TEXT_IO>

```
type COUNT is            range 0 .. 2_147_483_647;
subtype FIELD is         INTEGER range 0 .. INTEGER'last;
```

F.7.4 Values of Integer Attributes

The MC680*X*0 implementation of Domain/Ada provides three integer types in addition to *universal_integer*: INTEGER, SHORT_INTEGER, and TINY_INTEGER. Table F-1 lists the ranges for these integer types.

Table F-1. Domain/Ada Integer Types

Name of Attribute	Attribute Value of INTEGER	Attribute Value of SHORT_INTEGER	Attribute Value of TINY_INTEGER
FIRST	-2_147_483_648	-32_768	-128
LAST	2_147_483_647	32_767	127

F.7.5 Values of Floating-Point Attributes

Table F-2 lists the attributes of floating-point types.

Table F-2. Domain/Ada Floating-Point Types

Name of Attribute	Attribute Value of FLOAT	Attribute Value of SHORT_FLOAT
SIZE	64	32
FIRST	-1.79769313486231E+308	-3.40282E+38
LAST	1.79769313486231E+308	3.40282E+38
DIGITS	15	6
MANTISSA	51	21
EPSILON	8.88178419700125E-16	9.53674316406250E-07
EMAX	204	84
SMALL	1.94469227433160E-62	2.58493941422821E-26
LARGE	2.57110087081438E+61	1.93428038904620E+25
SAFE_EMAX	1022	126
SAFE_SMALL	1.11253692925360E-308	5.87747175411143E-39
SAFE_LARGE	4.49423283715578E+307	8.50755116541 54E+37
MACHINE_RADIX	2	2
MACHINE_MANTISSA	53	24
MACHINE_EMAX	1024	128
MACHINE_EMIN	-1022	-126
MACHINE_ROUNDS	TRUE	TRUE
MACHINE_OVERFLOWS	TRUE	TRUE

F.7.6 Attributes of Type DURATION

Table F-3 lists the attributes for the fixed-point type DURATION.

Table F-3. Attributes for the Fixed-Point Type DURATION

Name of Attribute	Attribute Value for DURATION
SIZE	32
FIRST LAST	-2147483.648 2147483.647
DELTA	1.00000000000000E-03
MANTISSA	31
SMALL LARGE	9.76562500000000E-04 4.19430399902343E_06
FORE AFT	8 3
SAFE_SMALL SAFE_LARGE	9.76562500000000E-04 4.19430399902343E+06
MACHINE_ROUNDS MACHINE_OVERFLOWS	TRUE TRUE

F.7.7 Implementation Limits

Character Set: Domain/Ada provides the full *graphic_character* textual representation for programs. The character set for source files and internal character representations is ASCII.

Lexical Elements, Separators, and Delimiters: Domain/Ada uses normal Domain/OS I/O text files as input. Each line is terminated by a newline character (ASCII.LF).

Source File Limits: Domain/Ada imposes the following limitations on source files:

- 499 characters per source line
- 1296 Ada units per source file
- 32767 lines per source file

Compiler/Tool Limits: Domain/Ada imposes the following limits on the use of the Domain/Ada compiler:

- 499 characters in identifiers and literals
- 4,000,000 STORAGE UNITS in a statically sized record or array
- 32,768 bytes as the STORAGE_SIZE default for a task
- No limit on the number of declared objects (except virtual space)
- 800 characters in a rooted name (full pathname of an object)
- 8 recursive inlines
- 8 nested inlines
- 400 nested constructs
- 2048 characters in ADAPATH (library search list)
- 2048 characters in a WITH or INFO directive
- 16M of memory use per compilation (other Domain/OS limits may apply)
- 50 lexical errors before the front end exits
- 100 syntax errors before the front end exits
- 10 attempts to lock **GVAS_table**
- 10 attempts to lock **ada.lib**
- 20 attempts to lock **gnrx.lib**
- 64 debugger breakpoints
- 32 debugger array dimensions in a **p** command
- 9 debugger 'call parameters'
- 256 debugger 'run parameters'

——— 88 ———

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..493 => '0', 494..499 => "69.0E1")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1 => '"', 2..200 => 'A', 201 => '"')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1 => '"', 2..300 => 'A', 301 => '1', 302 => '"')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..479=> ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	16777216
$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.	APOLLO_4_3_UNIX
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	0.0000000004656612873077392578125
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_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.	100000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10000000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	99
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	/ILLEGAL/FILE_NAME/2}]$%2102C.DAT
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/ILLEGAL/FILE_NAME/CE2102C*.DAT
$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.	-100000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10000000.0
$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.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 => "2:", 3..496 => '0', 497..499 => "11:")

Name and Meaning	Value
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 => "16:", 4..495 => '0', 496..499 => "F.E:")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '"', 2..498 => 'A', 499 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	TINY_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	APOLLO_4_3_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.	APOLLO_4_3_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.	0.01

APPENDIX D

WITHDRAWN TESTS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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