AD-A223 495

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1 Dec 89 to 1 Dec 90	Final

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: US Navy Ada/M, Version 2.0 (OPTIMIZE option), VAX 8550 and VAX 11/785 (Host) to AN/UYK-44 (Target), 891201S1.10214

5. FUNDING NUMBERS

6. AUTHOR(S)
National Institute of Standards and Technology
Gaithersburg, MD
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Institute of Standards and Technology
National Computer Systems Laboratory
Bldg. 255, Rm. A266
Gaithersburg, MD 20899
USA

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

U.S. Navy, Ada/M, Version 2.0 (/OPTIMIZE Option), Gaithersburg, MD, VAX 8550 and VAX 11/785 under VMS, Version 5.1 (Host) to AN/UYK-44 Bare machine (Target), ACVC 1.10.

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-01

AVF Control Number: NIST89USN555_6_1.10
DATE COMPLETED BEFORE ON-SITE: 08-11-89
DATE COMPLETED AFTER ON-SITE: 12-04-89

Ada Compiler Validation Summary Report:

Compiler Name: Ada/M, Version 2.0 (/OPTIMIZE Option)

Certificate Number: 891201S1.10214

Host: VAX 8550 and VAX 11/785 under VMS, Version 5.1

Target: AN/UYK-44 Bare machine

Testing Completed 12-01-89 Using ACVC 1.10

This report has been reviewed and is approved.

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

Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group
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: NIST89USN555_6_1.10
DATE VSR COMPLETED BEFORE ON-SITE: 08-11-89
DATE VSR COMPLETED AFTER ON-SITE: 12-04-89
DATE VSR MODIFIED PER AVO COMMENTS: 12-29-89
DATE VSR MODIFIED PER AVO COMMENTS: 04-27-90

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 891201S1.10214
U.S. NAVY
Ada/M, Version 2.0 (/OPTIMIZE Option)
VAX 8550 and VAX 11/785 Hosts and AN/UYK-44 Target

Completion of On-Site Testing:
12-01-89

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 12-01-89 at Syscon Corporation, Washington, D.C.

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 Commentary	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: Ada M, Version 2.0 (/OPTIMIZE Option)

ACVC Version: 1.10

Certificate Number: 891201S1.10214

Host Computer:

Machine: VAX 8550 and VAX 11/785

Operating System: VMS, Version 5.1

Memory Size: 48MBytes / 16MBytes

Target Computer:

Machine: AN/UYK-44

Operating System: Bare machine

Memory Size: 2MBytes

Communication Network: PORTAL/44

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 6 levels. (See test D64005E.)

b. Predefined types.

(1) This implementation supports the additional predefined types LONG_INTEGER in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

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

(1) 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 the same precision as the base type. (See test C35712B.)

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

(4) NUMERIC_ERROR is raised for pre-defined integer comparison and for pre-defined integer membership. NO EXCEPTION is raised for large_int comparison or for large_int membership. NUMERIC_ERROR is raised for small_int comparison and for small_int membership when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is gradual. (See tests C45524A..B (2 tests).)

d. Rounding.

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

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

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

(3) The method used for rounding to integer in static universal real expressions is round toward zero. (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 NUMERIC_ERROR. (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 STORAGE_ERROR when the array objects are declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises STORAGE_ERROR when the array objects are 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 no exception. (See test E52103Y.)

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

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

f. Discriminated types.

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

g. Aggregates.

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

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

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

E43211B.)

h. Pragmas.

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

i. Generics.

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

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

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

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

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

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

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

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

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

j. Input and output.

(1) The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

(2) The package DIRECT_IO cannot be instantiated with unconstrained array types or record types with

discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) USE_ERROR is raised when Mode IN_FILE is not supported for the operation of CREATE for SEQUENTIAL_IO. (See test CE2102D.)

(4) USE_ERROR is raised when Mode IN_FILE is not supported for the operation of CREATE for DIRECT_IO. (See test CE2102I.)

(5) Modes IN_FILE, OUT_FILE, and INOUT_FILE are not supported for DIRECT_IO. (See tests CE2102F, CE2102J, CE2102R, CE2102T, and CE2102V.)

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

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

(8) RESET and DELETE operations are not supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

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

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

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

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

(13) Only one internal file can be associated with each external file for sequential files when reading only. (See test CE2107A and CE2102L.)

(14) Only one internal file can be associated with each external file for sequential files when writing. (See tests CE2107B..E (4 tests), CE2110B, and CE2111D.)

(15) Only one internal file can be associated with each external file for direct files when reading. (See test CE2107F.)

(16) Only one internal file can be associated with each external file for direct files when writing. (See tests CE2107G..H (2 tests), CE2110D and CE2111H.)

(17) Only one internal file can be associated with each external

file for text files when reading only. (See CE3111A.)

(18) Only one internal file can be associated with each external file for text files when reading or writing. (See tests CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 638 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 327 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 38 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	127	1131	1694	15	22	46	3035
Inapplicable	2	7	621	2	6	0	638
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	189	527	472	245	170	99	160	332	137	36	252	181	235	3035
Inapplicable	23	122	208	3	2	0	6	0	0	0	0	188	86	638
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, 638 tests were inapplicable for the reasons indicated:

a. The following 327 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113C..Y (23 tests)	C35705C..Y (23 tests)
C35706C..Y (23 tests)	C35707C..Y (23 tests)
C35708C..Y (23 tests)	C35802C..Z (24 tests)

C45241C..Y (23 tests)	C45321C..Y (23 tests)
C45421C..Y (23 tests)	C45521C..Z (24 tests)
C45524C..Z (24 tests)	C45621C..Z (24 tests)
C45641C..Y (23 tests)	C46012C..Z (24 tests)

b. C35508I, C35508J, C35508M, and C35508N are not applicable because they include enumeration representation clauses for BOOLEAN types in which the representation values are other than (FALSE => 0, TRUE => 1). Under the terms of AI-00325, this implementation is not required to support such representation clauses.

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

d. C35702B and B86001U are not applicable because this implementation supports no predefined type LONG_FLOAT.

e. The following 16 tests are not applicable because this implementation does not support a predefined type SHORT_INTEGER:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	B86001V
CD7101E				

f. C45231D, B86001X, and CD7101G (3 tests) are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER or LONG_INTEGER.

g. G45531M..P (4 tests), C45532M..P (4 tests) are not applicable because this implementation does not support a 48 bit integer machine size.

h. D64005F and D64005G compile successfully but fail to build in the LINK because of excessive recursion.

i. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT.

j. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. This test recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

k. CD1009C, CD2A41A, CD2A41B, CD2A41E, CD2A42A, CD2A42B, CD2A42C, CD2A42D, CD2A42E, CD2A42F, CD2A42G, CD2A42H, CD2A42I, CD2A42J (14 tests) are not applicable because this implementation does not support 'SIZE representations for floating-point types.

l. CD1009N, CD1009X, CD1009Y, CD1009Z, CD1C03H, CD1C04E, CD4031A, CD4041A, CD4051A, CD4051B, CD4051C, CD4051D, CD7204C, ED1D04A are not applicable because record representation clauses are not

supported.

m. CD1C04C is not applicable because this implementation does not support 'SMALL specification clause for a derived fixed point type when it is inherited from the parent.

n. CD2A51A, CD2A51B, CD2A51C, CD2A51D, CD2A51E, CD2A52A, CD2A52B, CD2A52C, CD2A52D, CD2A52G, CD2A52H, CD2A52I, CD2A52J, CD2A53A, CD2A53B, CD2A53C, CD2A53D, CD2A53E, CD2A54A, CD2A54B, CD2A54C, CD2A54D, CD2A54G, CD2A54H, CD2A54I, CD2A54J, ED2A56A (27 tests) are not applicable because this implementation does not support 'SIZE representations for fixed-point types.

o. CD2A61A, CD2A61B, CD2A61C, CD2A61D, CD2A61E, CD2A61F, CD2A61G, CD2A61H, CD2A61I, CD2A61J, CD2A61K, CD2A61L, CD2A62A, CD2A62B, CD2A62C, CD2A64A, CD2A64B, CD2A64C, CD2A64D, CD2A65A, CD2A65B, CD2A65C, CD2A65D (23 tests) are not applicable because this implementation does not support size specifications for array types that imply compression of component storage.

p. CD2A71A, CD2A71B, CD2A71C, CD2A71D, CD2A72A, CD2A72B, CD2A72C, CD2A72D, CD2A74A, CD2A74B, CD2A74C, CD2A74D, CD2A75A, CD2A75B, CD2A75C, CD2A75D (16 tests) are not applicable because this implementation does not support the 'SIZE specification for record types implying compression of component storage.

q. CD2A84B, CD2A84C, CD2A84D, CD2A84E, CD2A84F, CD2A84G, CD2A84H, CD2A84I, CD2A84K, CD2A84L (10 tests) are not applicable because 'SIZE representation clauses for access types are not supported.

r. CD2A91A, CD2A91B, CD2A91C, CD2A91D, CD2A91E (5 tests) are not applicable because this implementation does not support the 'SIZE representation clauses for task types.

s. CD5003B, CD5003C, CD5003D, CD5003E, CD5003F, CD5003G, CD5003H, CD5003I, CD5011A, CD5011C, CD5011E, CD5011G, CD5011I, CD5011K, CD5011M, CD5011Q, CD5012A, CD5012B, CD5012E, CD5012F, CD5012I, CD5012J, CD5012M, CD5013A, CD5013C, CD5013E, CD5013G, CD5013I, CD5013K, CD5013M, CD5013O, CD5013S, CD5014A, CD5014C, CD5014E, CD5014G, CD5014I, CD5014K, CD5014M, CD5014O, CD5014S, CD5014T, CD5014V, CD5014X, CD5014Y, CD5014Z (46 tests) are not applicable because this implementation does not support 'ADDRESS clauses for variables.

t. CD5011B, CD5011D, CD5011F, CD5011H, CD5011L, CD5011N, CD5011R, CD5011S, CD5012C, CD5012D, CD5012G, CD5012H, CD5012L, CD5013B, CD5013D, CD5013F, CD5013H, CD5013L, CD5013N, CD5013R, CD5014B, CD5014D, CD5014F, CD5014H, CD5014J, CD5014L, CD5014N, CD5014R, CD5014U, CD5014W (30 tests) are not applicable because this implementation does not support 'ADDRESS clauses for constants.

u. AE2101C, EE2201D, and EE2201E use instantiations of package

SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

v. AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

w. The following 49 tests are not applicable because direct access files are not supported:

CE2102B	CE2102H
CE2102J..K (2 tests)	CE2102R..W (6 tests)
CE2102Y	CE2103D
CE2104C..D (2 tests)	CE2105B
CE2106B	CE2108C..D (2 tests)
CE2108G..H (2 tests)	CE2109B
CE2110C	CE2111B
CE2111E	CE2111G
CE2115A	CE2401A..C (3 tests)
CE2401E..F (2 tests)	CE2401H..L (5 tests)
CE2404A..B (2 tests)	CE2405B
CE2406A	CE2407A..B (2 tests)
CE2408A..B (2 tests)	CE2409A..B (2 tests)
CE2410A..B (2 tests)	CE2411A

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

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

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

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

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

ac. CE2105A is inapplicable because CREATE with IN_FILE mode is not supported by this implementation for SEQUENTIAL_IO.

ad. CE2107A..E (5 tests), CE2107L, CE2110B CE2111D are not applicable because multiple internal files cannot be associated with the same external file when one or more files is reading or writing for sequential files. The proper exception is raised when multiple access is attempted.

ae. CE2107F..H (3 tests), CE2110D, and CE2111H are not applicable

because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

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

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

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

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

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

ak. CE3109A is inapplicable because text file CREATE with IN_FILE mode is not supported by this implementation.

al. CE3111A..B (2 tests), CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is reading or writing for text files. The proper exception is raised when multiple access is attempted.

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 38 tests.

C35A06Q consists of a single procedure which instantiates several generic subprograms. In order to test the functionality that this test requires, the test was split into two tests. These two tests together encompass the same functionality of the original single test. If the test were not split, the size of the code generated for this test would be too large for the maximum size of a phase in the Ada/M system which is 64KBytes. With the test being split, this test reports PASS.

CC3126A was modified by inserting the initializing expression ":= (others => 'H')" into line numbered 117. With this modification, this test reports PASS.

For this implementation CD2C11A and CD2C11B were modified by inserting the initialization ":= 5.0" into variable W's declaration (note that W is declared along with one or two other variables in a single object declaration; the initialization is not needed for them, but does not affect their use). With this modification, these tests report PASS.

The following 34 tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B28003A	B28003C	B2A003A	B33201C	B33202C	B33203C	B33301B
B37106A	B37201A	B37301I	B38003A	B38003B	B38009A	B38009B
B44001A	B44004A	B51001A	B54A01L	B91001H	B95063A	BB1006B
BC1002A	BC1102A	BC1109A	BC1109B	BC1109C	BC1109D	BC1201F
BC1201G	BC1201H	BC1201I	BC1201J	BC1201L	BC3013A	

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 Ada/M, Version 2.0 (/OPTIMIZE Option) compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Ada/M, Version 2.0 (/OPTIMIZE Option) compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	VAX 8550 and VAX 11/785
Host operating system:	VMS, Version 5.1
Target computer:	AN/UYK-44
Target operating system:	Bare machine
Compiler:	Ada/M, Version 2.0 (/OPTIMIZE Option)
Linker:	LNK_M
Importer:	IMP_M
Exporter:	EXP_M
Loader/Downloader:	PORTAL/44
Runtime System:	RTEXEC Version 2.0/ RTLIB Version 2.0

The host and target computers were linked via PORTAL/44.

A magnetic tape containing all tests except for withdrawn tests 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 modified on-site.

TEST INFORMATION

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

The ACVC Version 1.10 was compiled and linked on the host VAX 8550. All executable tests were transferred to the AN/UYK-44 using PORTAL/44 and were run on the AN/UYK-44. Results were uploaded from the target system to the VAX 8550 stored on disk and printed.

The ACVC Version 1.10 was compiled and linked on the host VAX 11/785. All executable tests were transferred to the AN/UYK-44 using PORTAL/44 and were run on the AN/UYK-44. Results were uploaded from the target system to the VAX 11/785 stored on disk and printed.

The compiler was tested using command scripts provided by U.S. NAVY and reviewed by the validation team. See Appendix E for a complete listing of the compiler options for this implementation. The compiler options invoked during this test were:

For A, C, D, L Tests:

/SUMMARY /OPTIMIZE

For B, E Tests:

/SUMMARY /OPTIMIZE /SOURCE

Unless explicitly stated the following are the default options:

NO_SOURCE, NO_MACHINE, NO_ATTRIBUTE, NO_CROSS_REFERENCE, NO_DIAGNOSTICS, NO_SUMMARY, NO_NOTES, PRIVATE, CONTAINER_GENERATION, CODE_ON_WARNING, LIST, NO_MEASURE, DEBUG, NO_OPTIMIZE, CHECKS, NO_EXECUTIVE, NO_RTE_ONLY

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

3.7.3 Test Site

Testing was conducted at Syscon Corporation, Washington, D.C. and was completed on 12-01-89.

APPENDIX A

DECLARATION OF CONFORMANCE

U.S. NAVY has submitted the following Declaration of Conformance concerning the Ada/M, Version 2.0 (/OPTIMIZE Option).

DECLARATION OF CONFORMANCE

Customer: U.S. NAVY

Ada Validation Facility:

Ada Validation Facility
National Computer Systems Laboratory (NCSL)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

Ada Compiler Validation Capability (ACVC) Version: 1.10

Ada Implementation:	Ada/M, Version 2.0 (/OPTIMIZE Option)
Host Computer Systems:	VAX 8550 and VAX 11/785
Host OS and Version:	VMS, Version 5.1
Target Computer System:	AN/UYK-44
Target OS and Version:	Bare machine

Customer's Declaration

I, the undersigned, representing U.S. NAVY, declare that the U.S. NAVY has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declarations.

William L. Wilder Date: November 30, 1989

Signature of:
William L. Wilder,
U.S. NAVY

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 Ada/M, Version 2.0 (/OPTIMIZE Option) compiler, as described in this Appendix, are provided by U.S. NAVY. 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 -32_768 .. 32_767;
 type LONG_INTEGER is range -2_147_483_648 .. 2_147_483_647;

 type FLOAT is digits 6 range
  - (16#0.FFFFF8#E63) .. (16#0.FFFFF8#E63)

 type DURATION is delta 2.0 ** (-14) range
  -131_071.0 .. 131_071.0;

...

end STANDARD;
```

Section 6

The Ada Language For The AN/UYK-44 Target

The source language accepted by the compiler is Ada, as described in the Military Standard, Ada Programming Language, ANSI/MIL-STD-1815A-1983, 17 February 1983 ("Ada Language Reference Manual").

The Ada definition permits certain implementation dependencies. Each Ada implementation is required to supply a complete description of its dependencies, to be thought of as Appendix F to the Ada Language Reference Manual. This section is that description for the AN/UYK-44 target.

6.1 Options

There are several compiler options provided by all ALS/N Compilers that directly affect the pragmas defined in the Ada Language Reference Manual. These compiler options currently include the CHECKS and OPTIMIZE options which affect the SUPPRESS and OPTIMIZE pragmas, respectively. A complete list of ALS/N Compiler options can be found in Section 10 "Options".

The CHECKS option enables all run-time error checking for the source file being compiled, which can contain one or more compilation units. This allows the SUPRESS pragma to be used in suppressing the run-time checks discussed in the Ada Language Reference Manual, but note that the SUPRESS pragma(s) must be applied to each compilation unit. The NO_CHECKS option disables all run-time error checking for all compilation units within the source file and is equivalent to SUPPRESSing all run-time checks within every compilation unit.

The OPTIMIZE option enables all compile-time optimizations for the source file being compiled, which can contain one or more compilation units. This allows the OPTIMIZE pragma to request either TIME-oriented or SPACE-oriented optimizations be performed, but note that the OPTIMIZE pragma must be applied to each compilation unit. If the OPTIMIZE pragma is not present, the ALS/N Compiler's Global Optimizer tends to optimize for TIME over SPACE. The NO_OPTIMIZE option disables all compile-time optimizations for all compilation units within the source file regardless of whether or not the OPTIMIZE pragma is present.

In addition to those Compiler options normally provided by the ALS/N Common Ada Baseline Compilers, the Ada/M Compilers also implement the EXECUTIVE, DEBUG, and MEASURE options.

The EXECUTIVE Compiler option shall enable processing of the

EXECUTIVE pragma. The EXECUTIVE compiler option also allows WITH of units compiled with the /RTE_ONLY option. IF NO_EXECUTIVE is specified on the command line, the pragma will be ignored and will have no effect on the generated code.

The DEBUG Compiler option shall enable processing of the pragma DEBUG to provide debugging support. If NO_DEBUG is specified, the DEBUG pragmas shall have no effect. Program units containing DEBUG pragmas and compiled with the DEBUG Compiler option may be linked with program units containing DEBUG pragmas and compiled with the NO_DEBUG option; only those program units compiled with the DEBUG option shall have additional DEBUG support.

The MEASURE Compiler option shall enable processing of the pragma MEASURE to provide debugging support. If NO_MEASURE is specified, the MEASURE pragmas shall have no effect. Program units containing MEASURE pragmas and compiled with the MEASURE Compiler option may be linked with program units containing MEASURE pragmas and compiled with the NO_MEASURE option; only those program units compiled with the MEASURE option shall have additional MEASURE support.

6.2 Pragmas

These paragraphs describe the pragmas recognized and processed by the Ada/M Compiler. The syntax defined in Section 2.8 of the Ada Language Reference Manual allows pragmas as the only element in a compilation, before a compilation unit, at defined places within a compilation unit, or following a compilation unit. Ada/M associates pragmas with compilation units as follows:

a. If a pragma appears before any compilation unit in a compilation, it will affect all following compilation units, as specified below and in Section 2.8 of the Ada Language Reference Manual

b. If a pragma appears inside a compilation unit, it will be associated with that compilation unit, and with the listings associated with that compilation unit, as described in the Ada Language Reference Manual, or below.

c. If a pragma follows a compilation unit, it will be associated with the preceding compilation unit, and the effects of the pragma will be found in the container of that compilation unit and in listings associated with that container.

6.2.1 Language-Defined Pragmas

This paragraph specifies implementation-specific changes to those pragmas described in Appendix B of the Ada Language Reference Manual. Unmentioned pragmas are implemented as defined in the Ada Language Reference Manual.

The pragmas MEMORY_SIZE (arg), STORAGE_UNIT (arg), and SYSTEM_NAME (arg) must appear at the start of the first compilation when creating a program library, as opposed to the start of any compilation unit. If they appear elsewhere, a diagnostic of severity WARNING is generated and the pragma has no effect.

pragma INLINE (arg {,arg,...});

The arguments designate subprograms. There are three instances in which the INLINE pragma is ignored. Each of these cases produces a warning message which states that in INLINE did not occur.

a. If a call to an INLINEd subprogram is compiled before the actual body of the subprogram has been compiled (a routine call is made instead).

b. If the INLINEd subprograms compilation unit depends on the compilation unit of its caller (a routine call is made instead).

c. If an immediately recursive subprogram call is made within the body of the INLINEd subprogram (the pragma INLINE is ignored entirely).

pragma INTERFACE (arg, arg);

The first argument specifies the language and type of interface to be used in calls used to the externally supplied subprogram specified by the second argument. The only value allowed for the first argument (language name) is MACRO_NORMAL. MACRO_NORMAL indicates that parameters will be passed on the stack and the calling conventions used for normal Ada subprogram calls (see Section 3.4.14.2 of Ada/M_Intf_Spec) will apply.

The user must ensure that an Assembly Language Body Container will exist in the program library before linking.

pragma OPTIMIZE (arg);

This pragma is effective only when the "OPTIMIZE" option has been given to the compiler. The argument is either TIME or SPACE. If TIME is specified, the optimizer concentrates on optimizing code execution time. If SPACE is specified, the optimizer concentrates on optimizing code size.

pragma PRIORITY (arg);

The argument is an integer static expression in the range 0..15, where 0 is the lowest user-specifiable task priority and 15 is the highest. If the value of the argument is out of range, the pragma will have no effect other than to generate a WARNING diagnostic. A value of zero will be used if priority is not defined. The pragma will have no effect when not specified in a task (type) specification or the outermost declarative part of a subprogram. If the pragma appears in the declarative part of a subprogram, it will have no effect unless that subprogram is designated as the main subprogram at link time.

pragma SUPPRESS (arg {,arg});

This pragma is unchanged with the following exceptions:

Suppression of OVERFLOW_CHECK applies only to integer

operations; and a SUPPRESS pragma has effect only within the compilation unit in which it appears, except that suppression of ELABORATION_CHECK applied at the declaration of a subprogram or task unit applies to all calls or activations.

6.2.2 Implementation-Defined Pragmas

This paragraph describes the use and meaning of those pragmas recognized by Ada/M which are not specified in Appendix B of the Ada Language Reference Manual.

pragma DEBUG;

To be supplied.

pragma EXECUTIVE [(arg)];

This pragma allows the user to specify that a compilation unit is to run in the executive state of the machine and/or utilize privileged instructions. The pragma has no effect if the Compiler option NO_EXECUTIVE is enabled, either explicitly or by default.

If PRAGMA EXECUTIVE is specified without an argument, executive state is in effect for the compilation unit and the code generator does not generate privileged instructions for the compilation unit. If PRAGMA EXECUTIVE (INHERIT) is specified, a subprogram in the compilation unit inherits the state of its caller and the code generator does not generate privileged instructions for the compilation unit. If PRAGMA EXECUTIVE (PRIVILEGED) is specified, the executive state is in effect and the code generator may generate privileged instructions for the compilation unit. In the absence of PRAGMA EXECUTIVE, the compilation unit executes in task state and the code generator does not generate privileged instructions.

PRAGMA EXECUTIVE is applied once per compilation unit, so its scope is the entire compilation unit. PRAGMA EXECUTIVE may appear between the context clause and the outermost unit. If there is no context clause, the pragma EXECUTIVE must appear within that unit before the first declaration or statement. The placement of the pragma before the context clause has no effect on any or all following compilatin units. If PRAGMA EXECUTIVE appears in the specification of a compilation unit, it must also appear in the body of that unit, and vice versa. If the pragma appears in a specification but is absent from the body, the user is warned and the

pragma is effective. If the pragma appears in the body of a compilation unit, but is absent from the corresponding specification, the user is warned and the pragma has no effect. PRAGMA EXECUTIVE does not propagate to subunits. If a subunit is compiled without PRAGMA EXECUTIVE and the parent of the subunit is compiled with PRAGMA EXECUTIVE, the user is warned and PRAGMA EXECUTIVE has no effect on the subunit.

```
pragma FAST_INTERRUPT_ENTRY (arg1, arg2);
```

| To be supplied.

```
pragma MEASURE (extraction_set, [arg,...]);
```

| To be supplied.

```
pragma STATIC (arg);
```

| To be supplied.

```
pragma TICK (arg);
```

This is a system configuration pragma. It takes a single argument of type universal_real, which specifies the value of the named number system.tick. This pragma may appear only at the start of the first compilation when creating a program library. If this pragma appears elsewhere, a diagnostic of severity WARNING is generated.

```
pragma TITLE (arg);
```

This is a listing control pragma. It takes a single argument of type string. The string specified will appear on the second line of each page of every listing produced for a compilation unit. At most one such pragma may appear for any compilation unit, and it must be the first lexical unit in the compilation unit (excluding comments).

```
pragma TRIVIAL_ENTRY (NAME: entry_simple_name);
```

| To be supplied.

```
pragma UNMAPPED (arg, [arg,...]);
```

| The effect of this pragma is for unmapped (i.e., not consistently mapped within the virtual space) allocation of data in a compilation unit. The arguments of this pragma are access types to be unmapped. If a program tries to allocate more UNMAPPED space than is available in the physical configuration,

then STORAGE_ERROR will be raised at run-time. Pragma UNMAPPED must appear in the same declarative region as the type and after the type declaration.

6.2.3 Scope of Pragmas

The scope for each pragma previously described as differing from the Ada Language Reference Manual is given below.

DEBUG	To be supplied.
EXECUTIVE	Applies to the compilation unit in which the pragma appears, i.e., to all subprograms and tasks within the unit. Elaboration code is not affected. The pragma is not propagated from specifications to bodies, or from bodies to subunits. The pragma must appear consistently in the specification, body, and subunits associated with a library unit.
FAST_INTERRUPT_ENTRY	To be supplied.
INLINE	Applies only to subprograms named in its arguments. If the argument is an overloaded subprogram name, the INLINE pragma applies to all definitions of that subprogram name which appear in the same declarative part as the INLINE pragma.
INTERFACE	Applies to all invocations of the named imported subprogram.
MEASURE	To be supplied.
MEMORY_SIZE	Applies to the entire Program Library in which the pragma appears.
OPTIMIZE	Applies to the entire compilation unit in which the pragma appears.
PRIORITY	Applies to the task specification in which it appears, or to the environment task if it appears in the main subprogram.
STATIC	To be supplied.
STORAGE_UNIT	Applies to the entire Program Library in which the pragma appears.
SUPPRESS	Applies to the block or body that contains the declarative part in which the pragma appears.

SYSTEM_NAME	Applies to the entire Program Library in which the pragma appears.
TICK	Applies to the entire program library in which the pragma appears.
TITLE	Applies to the compilation unit in which the pragma appears.
TRIVIAL_ENTRY	To be supplied.
UNMAPPED	Applies to all objects of the access type named as arguments.

6.3 Attributes

There are two implementation-defined attributes in addition to the predefined attributes found in Appendix A of the Ada Language Reference Manual. These are defined below.

p'PHYSICAL_ADDRESS for a prefix p that denotes a data object:

Yields a value of type system.physical_address, which corresponds to the absolute address in physical memory of the object named by p. This attribute is only defined for static package specification and body data.

p'DISP for a prefix p that denotes a data object:

Yields a value of type universal_integer, which corresponds to the offset of p from the beginning of the frame containing p. For example, if p names a parameter, p'DISP is the offset of p from the argument pointer on the stack (see Section 3.4.14 of Ada/M_Intf_Spec). This attribute differs from the ADDRESS attribute in that ADDRESS supplies the virtual absolute address whereas DISP supplies a displacement. It is the user's responsibility to determine the base value, (i.e., frame pointer, argument pointer, psect), relevant to the object p. The runtime environment is described in Section 3.4.14 of Ada/M_Intf_Spec.

The following notes augment the language-required definitions of the predefined attributes found in Appendix A of the Ada Language Reference Manual.

T'MACHINE_EMAX	is 63.
T'MACHINE_EMIN	is -64.
T'MACHINE_MANTISSA	is 6.
T'MACHINE_OVERFLOWS	is TRUE.
T'MACHINE_RADIX	is 16.
T'MACHINE_ROUNDS	is FALSE.

6.4 Predefined Language Environment

The predefined Ada language environment consists of the packages STANDARD and SYSTEM, which are described below.

6.4.1 Package STANDARD

The package STANDARD contains the following definitions in addition to those specified in Appendix C of the Ada Language Reference Manual.

```
PACKAGE STANDARD IS

   TYPE boolean IS (false, true);

-- The type universal_integer is predefined.

   TYPE integer IS RANGE -32_768 .. 32_767;

   TYPE long_integer IS RANGE -2_147_483_648 .. 2_147_483_647;

   TYPE float IS DIGITS 6 RANGE
                -(16#0.FFFFF8#E63) .. (16#0.FFFFF8#E63);


   -- Predefined subtypes:

   SUBTYPE natural  IS integer RANGE 0 .. integer'LAST;
   SUBTYPE positive IS integer RANGE 1 .. integer'LAST;

   -- Predefined string type:

   TYPE string IS ARRAY (positive RANGE <>) OF character;

   PRAGMA PACK(string);

   TYPE duration IS DELTA 2.0 ** (-14)
                    RANGE -131_071.0 .. 131_071.0;

   -- The predefined operators for the type DURATION are the
   -- same as for any fixed point type.

   -- The predefined exceptions:

   constraint_error : exception;
   numeric_error    : exception;
   program_error    : exception;
   storage_error    : exception;
   tasking_error    : exception;

END STANDARD;
```

6.4.2 Package SYSTEM

The package SYSTEM for Ada/M is as follows:

```
PACKAGE SYSTEM IS

   memory_size : CONSTANT := 65_536;
      -- virtual memory size (not configurable).

   TYPE address IS RANGE 0..system.memory_size - 1;
      -- virtual address.

   TYPE name IS (anuyk44, anayk14);
      -- only one compatible system name.

   system_name : CONSTANT system.name := system.anuyk44;
      -- name of current system.

   storage_unit : CONSTANT := 16;
      -- word-oriented system (not configurable)

   -- System Dependent Named Numbers

   min_int : CONSTANT := -(2**31);
      -- most negative integer.

   max_int : CONSTANT :=  (2**31)-1;
      -- most positive integer.

   max_digits : CONSTANT := 6;
      -- most decimal digits in floating point constraint.

   max_mantissa : CONSTANT := 31;
      -- most binary digits for fixed point subtype.

   fine_delta : CONSTANT :=
                  2#0.0000_0000_0000_0000_0000_0000_0000_001#;
      -- 2**(-31) is minimum fixed point constraint.

   tick : CONSTANT := 3.125e-05;
      -- 1/32000 seconds is the basic clock period.

   -- Other System Dependent Declarations

   SUBTYPE priority IS integer RANGE 0..15;
      -- task priority, lowest = default = 0.

   FOR address'SIZE USE 16;
      -- virtual address is a 16-bit quantity.

   physical_memory_size : CONSTANT :=  2**22;
      -- maximum physical memory size (not configurable).
```

```
   TYPE physical_address IS
      RANGE 0..system.physical_memory_size - 1;
      -- absolute address.

   TYPE external_interrupt_word IS RANGE 0 .. 65_536;
      -- Parameter type for Class III Priority
      -- 2
      -- External Interrupt entry.

   -- This type is used by the Compiler in the definition
   -- of interrupt entries.
   TYPE entry_kind IS (normal, immediate);

   -- implementation-defined exceptions.
   access_check          : EXCEPTION;
   discriminant_check    : EXCEPTION;
   index_check           : EXCEPTION;
   length_check          : EXCEPTION;
   range_check           : EXCEPTION;
   division_check        : EXCEPTION;
   overflow_check        : EXCEPTION;
   elaboration_check     : EXCEPTION;
   storage_check         : EXCEPTION;
   unresolved_reference  : EXCEPTION;
   system_error          : EXCEPTION;
   capacity_error        : EXCEPTION;

END SYSTEM;
```

6.5 Character Set

Ada compilations may be expressed using the following characters in addition to the basic character set:

lower case letters:

a b c d e f g h i j k l m n o p q r s t u v w x y z

special characters:

! $? @ [ ] ^ { } ~ ` (accent grave) %

6.6 Representation and Declaration Restrictions

Representation specifications are described in Section 13 of the Ada Language Reference Manual. Declarations are described in Section 3 of the Ada Language Reference Manual.

In the following specifications, the capitalized word SIZE indicates the number of bits used to represent an object of the type under discussion. The upper case symbols D, L, R, correspond to those discussed in Section 3.5.9 of the Ada Language Reference Manual.

6.6.1 Integer Types

Integer types are specified with constraints of the form:

```
RANGE L..R
```

where:

```
R <= SYSTEM.MAX_INT & L >= SYSTEM.MIN_INT
```

For a prefix "t" denoting an integer type, length specifications of the form:

```
FOR t'SIZE USE n ;
```

may specify integer values n such that

```
n in 2..16,
```

and such that

```
R <= 2**(n-1)-1 & L >= -(2**(n-1))
```

or else such that

```
R <= (2**n)-1 & L >= 0
```

and 1 < n <= 15.

For a standalone object of integer type, a default SIZE of 16 is used when:

```
R <= 2**15-1 and L >= -2**15
```

Otherwise, a SIZE of 32 is used.

For components of integer types within packed composite objects, the smaller of the default standalone SIZE and the SIZE from a length specification is used.

6.6.2 Floating Types

Floating types are specified with constraints of the form:

```
DIGITS D
```

where D is an integer in the range 1 through 6.

For a prefix "t" denoting a floating point type, length specifications of the form:

```
FOR t'SIZE USE n;
```

may specify integer values n = 32 when D <= 6. All floating point values have SIZE = 32.

6.6.3 Fixed Types

Fixed types are specified with constraints of the form:

```
DELTA D RANGE L..R
```

where:

$$\frac{\text{MAX (ABS(R), ABS(L))}}{\text{actual delta}} <= 2^{**}31-1.$$

The actual delta defaults to the largest integral power of 2 less than or equal to the specified delta D. (This implies that fixed values are stored right-aligned.) For specifications of the form:

```
FOR t'SMALL USE n;
```

n must be specified as an integral power of 2 such that n <= D.

For a prefix "t" denoting a fixed point type, length specifications of the form:

```
FOR t'SIZE USE n;
```

are permitted only when n = 32. All fixed values have SIZE = 32.

6.6.4 Enumeration Types

In the absence of a representation specification for an enumeration type "t," the internal representation of t'FIRST = 0.

The default size for a standalone object of enumeration type "t" is 16, so the internal representations of t'FIRST and t'LAST both fall within the range

```
-2**15 .. 2**15 - 1.
```

For enumeration types, length specifications of the form:

```
FOR t'SIZE USE n;
```

and/or enumeration representations of the form:

```
FOR t USE <aggregate>;
```

are permitted for n in 2..16, provided that the representations and the SIZE conform to the relationship specified above, or else for n in 1..16, is supported for enumeration types and provides an internal representation of:

```
t'FIRST >= 0 .. t'LAST <= 2**(t'SIZE) - 1.
```

For components of enumeration types within packed composite objects, the smaller of the default standalone SIZE and the SIZE from a length specification is used.

Enumeration representations on types derived from the predefined type standard.boolean will not be accepted, but length specifications will be accepted.

6.6.5 Access Types

For access type, "t", length specifications of the form:

```
FOR t'SIZE USE n;
```

will not affect the runtime implementation of "t", therefore n=16 is the only value permitted for SIZE, which is the value returned by the attribute.

For collection size specification of the form:

```
FOR t'STORAGE_SIZE USE n;
```

for any value of "n" is permitted for STORAGE_SIZE (and that value will be returned by the attribute call). The collection size specification will affect the implementation of "t" and its collection at runtime by limiting the number of objects for type "t" that can be allocated.

The value of t'STORAGE_SIZE for an access type "t" specifies the maximum number of storage_units used for all objects in the

collection for type "t." This includes all space used by the allocated objects, plus any additional storage required to maintain the collection.

6.6.6 Arrays and Records

For arrays and records, length specification of the form:

```
FOR t'size USE n;
```

is not allowed unless it is the default size.

The PACK pragma may be used to minimize wasted space between components of arrays and records. The pragma causes the type representation to be chosen such that the storage space requirements are minimized at the possible expense of data access time and code space.

A record type representation specification is not allowed.

For records, an alignment clause of the form:

```
AT MOD n
```

specify alignments of 1 word (word alignment) or 2 words (doubleword alignment).

If it is determinable at compile time that the SIZE of a record or array type or subtype is outside the range of standard.long_integer, a diagnostic of severity WARNING is generated. Declaration of such a type or subtype would raise NUMERIC_ERROR when elaborated.

6.6.7 Other Length Specifications

Length Specifications are described in Section 13.2 of the Ada Language Reference Manual.

A length specification for a task type "t" of the form:

```
For t'STORAGE_SIZE USE n;
```

specifies the number of system.storage_untis that are allocated for the execution each task object of type "t". This includes the runtime stack for the task object but does not include objects allocated at runtime by the task object.

6.7 System Generated Names

Refer to Section 13.7 of the Ada Language Reference Manual and the section above on the Predefined Language Environment for a discussion of package SYSTEM.

The system name is chosen based on the target(s) supported, but it cannot be changed. In the case of Ada/M, the system name is "ANUYK44".

6.8 Address Clauses

Refer to Section 13.5 of the Ada Language Reference Manual for a description of address clauses. All rules and restrictions described there apply. In addition, the following restrictions apply.

An address clause designates a single task entry only. The appearance of a data object, subprogram, package, or task unit name in an address clause is not allowed, and will result in the generation of a diagnostic of severity ERROR.

An address clause may designate a single task entry. Such an address clause is allowed only within a task specification compiled with the EXECUTIVE Compiler option. The meaningful values of the simple_expression are the allowable interrupt entry addresses defined in Table 6-1. The use of other values will result in the raising of a PROGRAM_ERROR exception upon creation of the task.

If more than one task entry is equated to the same interrupt entry address, the most recently executed interrupt entry registration permanently overrides any previous registrations.

At most one address clause is allowed for a single task entry. Specification of more than one interrupt address for a task entry is erroneous.

Class 0 interrupts (with interrupt entry address) include:		
o Class I Unhandled Interrupt	16#0800#	
Class I interrupts (with interrupt entry address) include:		
o Class II Unhandled	16#1800#	
o CP Memory Resume	16#1000#	
o CP Memory Parity	16#1400#	
o IOC Memory Parity	16#1700#	
o IOC Memory Resume	16#1A00#	
o Power Fault	16#1F00#	
Class II interrupts (with interrupt entry address) inclue:		
o Class III Unhandled	16#2800#	
o Floating Point Over/Underflow	16#2100#	UNDEFINABLE
o CP Instruction Fault	16#2200#	
o Executive Mode Fault	16#2300#	
o IOC Instruction Fault	16#2400#	
o IOC Protect Fault	16#2500#	
o Executive Return	16#2600#	UNDEFINABLE
o CP Protect Fault	16#2900#	
o Real-Time Clock	16#2E00#	UNDEFINABLE
o Monitor Clock	16#2F00#	UNDEFINABLE
Class III interrupts (with interrupt entry address) include:		
o MMIO Discrete Interrupt	16#38CC#	
o MMIO External Interrupt	16#39CC#	
o MMIO Output Data Ready	16#3ACC#	
o MMIO Input Data Ready	16#3BCC#	
o IOC Intercomputer Timeout	16#3CIC#	
o IOC External Interrupt/Discrete	16#3DIC#	
o IOC Output Chain Interrupt	16#3EIC#	
o IOC Input Chain Interrupt	16#3FIC#	

For all class III interrupts, the following interpretations apply:

IC => IOC, CHANNEL pair, 16#00#..16#0F# indicates IOC 0
16#10#..16#1F# indicates IOC 1
16#20#..16#2F# indicates IOC 2
16#30#..16#3F# indicates IOC 3

CC => CHANNEL number, 16#00#..16#3F# indicates channel 0..63

Table 6-1 - Interrupt Entry Addresses

6.9 Unchecked Conversions

Refer to Section 13.10.2 of the Ada Language Reference Manual for a description of UNCHECKED_CONVERSION. It is erroneous if the user written Ada program performs UNCHECKED_CONVERSION when the source and target objects have different sizes.

6.10 Restrictions on the Main Subprogram

Refer to Section 10.1 (8) of the Ada Language Reference Manual for a description of the main subprogram. The subprogram designated as the main subprogram cannot have parameters. The designation as the main subprogram of a subprogram whose specification contains a formal_part results in a diagnostic of severity ERROR at link time.

The main subprogram can be a function, but the return value will not be available upon completion of the main subprogram's execution. The main subprogram may not be an import unit.

6.11 Input/Output

Refer to Section 14 of the Ada Language Reference Manual for a discussion of Ada Input/Output and to Section 12 of the Ada/M Run Time Environment Handbook for more specifics on the Ada/M input/output subsystem.

6.11.1 Naming External Files

The naming conventions for external files in Ada/M are of particular importance to the user. All of the system-dependent information needed by the I/O subsystem about an external file is contained in the file name. External files may be named using one of three file naming conventions: standard, temporary, and user-derived.

a. Standard File Names:

The standard external file naming convention used in Ada/M identifies the specific location of the external file in terms of the physical device on which it is stored. For this reason, the user should be aware of the configuration of the peripheral devices on the AN/UYK-44 at a particular user site.

Standard file names may be six to twenty characters long; however, the first six characters follow a predefined format. The first and second characters must be either "CT," "MT," or "TT," designating an AN/USH-26 Signal Data Recorder/Reproducer Set, the RD-358 Magnetic Tape Subsystem, or the AN/USQ-69 Data Terminal Set, respectively. These characters must be in upper case.

The third and fourth characters specify the channel on which the peripheral device is connected. Since there are sixty-four channels on the AN/UYK-44, the values for the third and fourth positions must lie in the range "00" to "63."

The range of values for the fifth position in the external file name (the unit number) depends upon the device specified by the characters in the first and second positions of the external file name. If the specified peripheral device is the AN/USH-26 magnetic tape drive, then the character in the fifth position must be one of the characters "0," "1," "2," or "3." This value determines which of the four tape cartridge units available on the AN/USH-26 is to be accessed. If the specified peripheral device is the RD-358 magnetic tape drive, the character in the fifth position must be one of the characters "0," "1," "2," or "3." This value determines which of the four tape units available on the

RD-358 is to be accessed. If the specified peripheral device is the AN/USQ-69 militarized display terminal, the character in the fifth position must be a "0." The AN/USQ-69 has only one unit on a channel.

The colon, ":", is the only character allowed in the sixth position. If any character other than the colon is in this position, the file name will be considered non-standard and the file will reside on the default device defined during the elaboration of CONFIGURE_IO.

Positions seven through twenty are optional to the user written Ada program and may be used as desired. These positions may contain any printable character the user chooses in order to make the file name more intelligible. Embedded blanks, however, are not allowed.

The location of an external file on a peripheral device is thus a function of the first six characters of the file name regardless of the characters that might follow. For example, if the external file "CT000:Old_Data" has been created and not subsequently closed, an attempt to create the external file "CT000:New_Data" will cause the exception DEVICE_ERROR (rather than NAME_ERROR or USE_ERROR) to be raised because the peripheral device on channel "00" and cartridge "0" is already in use.

The user is advised that any file name beginning with "xxxxx:" (where x denotes any printable character) is assumed to be a standard external file name. If this external file name does not conform to the Ada/M standard file naming conventions, the exception NAME_ERROR will be raised.

b. Temporary File Names:

Section 14.2.1 of the Ada Language Reference Manual defines a temporary file to be an external file that is not accessible after completion of the main subprogram. If the null string is supplied for the external file name, the external file is considered temporary. In this case, the high level I/O packages internally create an external file name to be used by the lower level I/O packages. The internal naming scheme used by the I/O subsystem is a function of the type of file to be created (text, direct or sequential), the temporary nature of the external file, and the number of requests made thus far for creating temporary external files of the given type. This scheme is consistent with the requirement specified in the Ada Language Reference Manaul that all external file names be unique.

The first three characters of the file name are "TEX," "DIR," or "SEQ." The next six characters are "_TEMP_." The remaining characters are the image of an integer which denotes the number of temporary files of the given type successfully created. There are two types of temporary files; one is used by SEQUENTIAL_IO and DIRECT_IO, and the other is used by TEXT_IO. For instance, the temporary external file name "TEX_TEMP_10" would be the name of the tenth temporary external file successfully created by the user written Ada program through calls to TEXT_IO.

c. User-Derived File Names:

A random string containing a sequence of characters of length one to twenty may also be used to name an external file. External files with names of this nature are considered to be permanent external files. The user is cautioned to refrain from using names which conform to the scheme used by the I/O subsystem to name temporary external files (see list item "b.").

It is not possible to associate two or more internal files with the same external file. The exception USE_ERROR will be raised if this restriction is violated.

6.11.2 The FORM Specification for External Files

Section 14.2.1 of the Ada Language Reference Manual defines a string argument called the FORM, which supplies system-dependent information that is sometimes required to correctly process a request to create or open a file. In Ada/M, the string argument supplied to the FORM parameter on calls to CREATE and OPEN is retained while the file is open, so that calls to the function FORM can return the string to the user written Ada program. Form options specified on calls to CREATE have the effects stated below. Form options specified on calls to OPEN have no effect.

The REWIND and APPEND options are mutually exclusive; an attempt to specify both options on a call to CREATE will raise the exception USE_ERROR.

The NOHEAD option may be specified in combination with either the REWIND or the APPEND option.

If one form option is specified, the FORM string should contain only the option, without any extraneous characters. If two form options are specified, the FORM string should contain the first form option followed by a comma followed by the second form option. The form options may be specified in any

combination of upper and lower case.

If the supplied FORM string is longer than the maximum allowed FORM string (13 characters), CREATE and OPEN will raise the exception USE_ERROR.

If the procedure CREATE does not recognize the options specified in the FORM string, it raises the exception USE_ERROR. The procedure OPEN does not validate the contents of the supplied FORM string.

Positioning arguments allow control of tape before its use. The following positioning arguments are available to the user:

a. REWIND - specifies that a rewind will be performed prior to the requested operation.

b. NOREWIND - specifies that the tape remains positioned as is.

c. APPEND - specifies that the tape be positioned at the logical end of tape (LEOT) prior to the requested operation. The LEOT is denoted by two consecutive tape_marks.

The formatting argument specifies information about tape format. If a formatting argument is not supplied, the file is assumed to contain a format header record determined by the ALS/N I/O system. The following formatting argument is available to the user:

a. NOHEAD - specifies that the designated file has no header record. This argument allows the reading and writing of tapes used on computer systems using different header formats. Note that files created with the NOHEAD option cannot be opened by the Ada/M I/O subsystem.

6.11.3 File Processing

Processing allowed on Ada/M files is influenced by the characteristics of the underlying device. The following restrictions apply:

a. Only one file may be open on an individual AN/USH-26 tape cartridge at a time.

b. Only one input and one output file may simultaneously be open on an AN/USQ-69 terminal at one time.

c. A user-written Ada program is erroneous if it does not close or delete all files that it creates or opens.

d. The attempt to CREATE a file with the mode IN_FILE is not supported since there will be no data in the file to read.

6.11.4 Text Input/Output

TEXT_IO is invoked by the user-written Ada program to perform sequential access I/O operations on text files (i.e, files whose content is in human-readable form). TEXT_IO is not a generic package and, thus, its subprograms may be invoked directly from the user-written Ada program, using objects with base type or parent type in the language-defined type character. TEXT_IO also provides the generic packages INTEGER_IO, FLOAT_IO, FIXED_IO, and ENUMERATION_IO for the reading and writing of numeric values and enumeration values. The generic packages within TEXT_IO require an instantiation for a given element type before any of their subprograms are invoked. The specification of this package is given in Section 14.3.10 of the Ada Language Reference Manual.

The implementation-defined type COUNT that appears in Section 14.3.10 of the Ada Language Reference Manual is defined as follows:

```
type COUNT is range 0...INTEGER'LAST;
```

The implementation-defined subtype FIELD that appears in Section 14.3.10 of the Ada Language Reference Manual is defined as follows:

```
subtype FIELD is INTEGER range 0...INTEGER'LAST;
```

At the beginning of program execution, the STANDARD_INPUT file and the STANDARD_OUTPUT file are open, and associated with the files specified by the user at export time. Additionally, if a program terminates before an open file is closed (except for STANDARD_INPUT and STANDARD_OUTPUT), then the last line which the user added to the file may be lost; if the file is on magnetic tape, the file structure on the tape may be inconsistent.

A program is erroneous if concurrently executing tasks attempt to perform overlapping GET and/or PUT operations on the same terminal. The semantics of text layout as specified in the Ada Language Reference Manual, Section 14.3.2, (especially the concepts of current column number and current line) cannot be guaranteed when GET operations are interweaved with PUT

operations. A program which relies on the semantics of text layout under those circumstances is erroneous.

For TEXT_IO processing, the line length can be no longer than 1000 characters. An attempt to set the line length through SET_LINE_LENGTH to a length greater than 1000 will result in USE_ERROR.

6.11.5 Sequential Input/Output

SEQUENTIAL_IO is invoked by the user-written Ada program to perform I/O on the records of a file in sequential order. The SEQUENTIAL_IO package also requires a generic instantiation for a given element type before any of its subprograms may be invoked. Once the package SEQUENTIAL_IO is made visible, it will perform any service defined by the subprograms declared in its specification. The specification of this package is given in Section 14.2.3 of the Ada Language Reference Manual.

The following restrictions are imposed on the use of the package SEQUENTIAL_IO:

a. SEQUENTIAL_IO must be instantiated for a constrained type.

b. Ada/M does not raise DATA_ERROR on a read operation if the data input from the external file is not of the instantiating type (see the Ada Language Reference Manual, Section 14.2.2).

6.11.6 Direct Input/Output

Calls to the subprograms of an instantiation of DIRECT_IO have one of three possible outcomes. The exception USE_ERROR is raised if an attempt is made to CREATE and/or OPEN a file since direct access I/O operations are not supported in Ada/M. The exception STATUS_ERROR is raised on calls to subprograms other than CREATE, OPEN, and IS_OPEN. The function IS_OPEN always returns the value FALSE.

The implementation-defined type COUNT that appears in Section 14.2.5 of the Ada Language Reference Manual is defined as follows:

```
type COUNT is range 0..io_support.count'LAST;
```

where io_support.count'LAST is equal to LONG_INTEGER'LAST.

6.11.7 Low Level Input/Output

LOW_LEVEL_IO is invoked by the user-written Ada program to initiate physical operations on peripheral devices, and thus executes as part of the user-written Ada program task. Requests made to LOW_LEVEL_IO from the user-written Ada program are passed through the RTEXEC_GATEWAY to the channel programs in CHANNEL_IO. Any status check or result information is the responsibility of the invoking subprogram and can be obtained from the subprogram RECEIVE_CONTROL within LOW_LEVEL_IO.

The package LOW_LEVEL_IO allows the user written Ada program to send I/O commands to the I/O devices (using SEND_CONTROL) and to receive status information from the I/O devices (using RECEIVE_CONTROL). A program is erroneous if it uses LOW_LEVEL_IO to access a device that is also accessed by high-level I/O packages such as SEQUENTIAL_IO and TEXT_IO. The following is excerpted from the package LOW_LEVEL_IO.

```
PACKAGE LOW_LEVEL_IO IS

   -- IO_CHANNEL_RANGE is the type for the parameter DEVICE for
both
   -- SEND_CONTROL and RECEIVE_CONTROL.  DEVICE identifies which
   -- device to perform the operation for, and the channel number
   -- is a convenient means for identifying a device.

   SUBTYPE io_channel_range IS integer RANGE 0..63;
      -- Range of values allowed for channel number.

   SUBTYPE buffer_address IS system.physical_address;
      -- Type of variables used to specify
      -- address of buffer for the I/O operation.

   SUBTYPE command_word IS long_integer RANGE 0..65535;

-- Data structures used in communication with the AN/USH-26.

   ush26_programs : CONSTANT := 3;
      -- Number of channel programs in CHANNEL_IO for
      -- AN/USH-26 devices.

   SUBTYPE ush26_operation IS integer
      RANGE 0..low_level_io.ush26_programs-1;
      -- Indicates to CHANNEL_IO which channel program to use.

   ush26_read_data  : ush26_operation := 0;
   ush26_write_data : ush26_operation := 1;
   ush26_control    : ush26_operation := 2;

   TYPE ush26_data IS
      -- Data passed to SEND_CONTROL for operations on
      -- AN/USH-26 devices.
      RECORD
         operation   : low_level_io.ush26_operation;
```

```
            -- Kind of operation requested of LOW_LEVEL_IO:
            -- read data, write data, control, or initialize.
         command     : low_level_io.command_word;
            -- Command to send to the device.
         data_length : integer range 0..integer'last;
            -- Number of words of data in the buffer.
         buffer_addr : low_level_io.buffer_address;
            -- Physical address of data buffer.
      END RECORD;

-- Data structures used in communication with the AN/USQ-69.

   usq69_programs : CONSTANT := 4;
      -- Number of channel programs in CHANNEL_IO for
      -- AN/USQ-69 devices.

   SUBTYPE usq69_operation IS integer
      RANGE 0..low_level_io.usq69_programs-1;
      -- Indicates to CHANNEL_IO which channel program to use.

   usq69_header     : usq69_operation := 0;
   usq69_read_data  : usq69_operation := 1;
   usq69_write_data : usq69_operation := 2;
   usq69_eot        : usq69_operation := 3;

   TYPE usq69_data IS
      -- Information needed to do I/O to a AN/USQ-69 device.
      RECORD
         operation   : low_level_io.usq69_operation;
            -- Kind of operation requested of LOW_LEVEL_IO:
            -- read data, write data, control, or initialize.
         command     : low_level_io.command_word;
            -- Command to send to the device.
         data_length : integer range 0..integer'last;
            -- Number of words of data in the buffer.
         buffer_addr : low_level_io.buffer_address;
            -- Physical address of data buffer.
      END RECORD;

   rd358_programs : CONSTANT := 3;
      -- Number of channel programs in CHANNEL_IO for
      -- RD-358 devices.

   SUBTYPE rd358_operation IS integer
      RANGE 0..low_level_io.rd358_programs-1;
      -- Indicates to CHANNEL_IO which channel program to use.

   rd358_read_data  : rd358_operation := 0;
   rd358_write_data : rd358_operation := 1;
   rd358_control    : rd358_operation := 2;

   TYPE rd358_data IS
      -- Information needed to do I/O to an RD-358 device.
```

```
      RECORD
         operation    : low_level_io.rd358_operation;
            -- Kind of operation requested of LOW_LEVEL_IO:
            -- read data, write data, control, or initialization.
         command      : low_level_io.command_word;
            -- Command to send to the device.
         data_length : integer range 0..integer'last;
            -- Number of words of data in the buffer.
         buffer_addr : low_level_io.buffer_address;
            -- Physical address of data buffer.
      END RECORD;

-- Types used for intercomputer I/O operations.

   ic_programs : CONSTANT := 3;
      -- Number of channel programs in CHANNEL_IO for
      -- AN/USH-26 devices.

   SUBTYPE intercomputer_operation IS integer
      RANGE 0..low_level_io.ic_programs-1;
      -- Indicates to CHANNEL_IO which channel program to use.

   ic_read_data  : intercomputer_operation := 0;
   ic_write_data : intercomputer_operation := 1;
   ic_control    : intercomputer_operation := 2;

   TYPE intercomputer_data IS
      -- Information needed to do I/O to an intercomputer
channel.
      RECORD
         operation    : low_level_io.intercomputer_operation;
            -- Kind of operation requested of LOW_LEVEL_IO:
            -- read data, write data, control, or initialization.
         command      : low_level_io.command_word;
            -- Command to send to the other computer.
         data_length : integer range 0..integer'last;
            -- Number of words of data in the buffer.
         buffer_addr : low_level_io.buffer_address;
            -- Physical address of data buffer.
      END RECORD;

-- Data type identifiers for RECEIVE_CONTROL.

   TYPE io_status_word IS NEW long_integer RANGE 0..65535;
      -- Used to pass I/O status word to RECEIVE_CONTROL.

   SUBTYPE external_interrupt_word IS
system.external_interrupt_word;

-- SEND_CONTROL is an overloaded Ada procedure which passes I/O
-- control information to a procedure in CHANNEL_IO in order to
carry
-- out a read, write, or control operation.  In Ada/M, there are
```

```
-- four overloaded subprograms for SEND_CONTROL, one for each of
the
-- following purposes :
--
--      send data/command to an AN/USH-26 device,
--      send data/command to an AN/USQ-69 device,
--      send data/command to an RD-358 device,
--      send data/command to another computer.

-- The following versions of the overloaded procedure
SEND_CONTROL
-- are used for sending data to specific types of devices.  The
-- difference between the various forms of this procedure lies in
-- the DATA parameter, which is a record with a field that
specifies
-- the control command to send to the device.  The data type of
this
-- field is different for each type of device.

   -- SEND_CONTROL for AN/USH-26 devices.
   PROCEDURE SEND_CONTROL (
       device : IN     low_level_io.io_channel_range;
                -- Channel number of the peripheral device.
       data   : IN OUT low_level_io.ush26_data
                -- I/O control information for AN/USH-26 devices.
      );

   -- SEND_CONTROL for AN/USQ-69 devices.
   PROCEDURE SEND_CONTROL (
       device : IN     low_level_io.io_channel_range;
                -- Channel number of the peripheral device.
       data   : IN OUT low_level_io.usq69_data
                -- I/O control information for AN/USQ-69 devices.
      );

   -- SEND_CONTROL for RD-358 devices.
   PROCEDURE SEND_CONTROL (
       device : IN     low_level_io.io_channel_range;
                -- Channel number of the peripheral device.
       data   : IN OUT low_level_io.rd358_data
                -- I/O control information for AN/USQ-69 devices.
      );

   -- SEND_CONTROL for Intercomputer channel.
   PROCEDURE SEND_CONTROL (
       device : IN     low_level_io.io_channel_range;
                -- Channel number of the peripheral device.
       data   : IN OUT low_level_io.intercomputer_data
                -- I/O control information for AN/USQ-69 devices.
      );

-- RECEIVE_CONTROL is a procedure which passes I/O control
-- information to a procedure in CHANNEL_IO in order to obtain
```

```
-- the value for the input transfer count for the specified
-- channel.

   PROCEDURE RECEIVE_CONTROL (
       device : IN     low_level_io.io_channel_range;
                -- Device type for which status is requested.
       data   : IN OUT low_level_io.io_status_word
                -- External interrupt word for channel specified.
      );

   -- RECEIVE_CONTROL for getting the external interrupt data
   -- for the specified channel.
   PROCEDURE RECEIVE_CONTROL (
       device : IN     low_level_io.io_channel_range;
                -- Channel number of the peripheral device.
       data   : IN OUT low_level_io.external_interrupt_word
                -- Input count for channel specified.
      );

   -- RECEIVE_CONTROL for getting input transfer count.
   PROCEDURE RECEIVE_CONTROL (
       device : IN     low_level_io.io_channel_range;
                -- Channel number of the peripheral device.
       data   : IN OUT integer
                -- Input count for channel specified.
      );

END LOW_LEVEL_IO;
```

6.12 System-Defined Exceptions

In addition to the exceptions defined in the Ada Language Reference Manual, this implementation pre-defines the exceptions shown in Table 6-2 below.

Name	Significance
UNRESOLVED_REFERENCE	The exception UNRESOLVED_REFERENCE is raised whenever a call is made to a subprogram whose body is not included in the linked program image. In addition, this exception is raised whenever a reference to a data object cannot be resolved by the Linker/Exporter. Since Ada/M provides a selective linking capability through the use of a units list file, the subprogram body may not always be present in the linked program image.
SYSTEM_ERROR	The exception SYSTEM_ERROR signifies an internal error in the Run-Time Operating System that is not the fault of the user-written Ada program.
CAPACITY_ERROR	The exception CAPACITY_ERROR is raised by the Run-Time Executive when pre-runtime-specified resource limits are exceeded.

Table 6-2a - System Defined Exceptions

Name	Significance
ACCESS_CHECK	The ACCESS_CHECK exception has been raised explicitly within the program.
DISCRIMINANT_CHECK	DISCRIMINANT_CHECK exception has been raised explicitly within the program.
INDEX_CHECK	The INDEX_CHECK exception has been raised explicitly within the program.
LENGTH_CHECK	The LENGTH_CHECK exception has been raised explicitly within the program.
RANGE_CHECK	The RANGE_CHECK exception has been raised explicitly within the program.
DIVISION_CHECK	The DIVISION_CHECK exception has been raised explicitly within the program.
OVERFLOW_CHECK	The OVERFLOW_CHECK exception has been raised explicitly within the program.
ELABORATION_CHECK	The ELABORATION_CHECK exception has been raised explicitly within the program.

Table 6-2b - System Defined Exceptions (Continued)

6.13 Machine Code Insertions

The Ada language permits machine code insertions as defined in Section 13.8 of the Ada Language Reference Manual. This section describes the specific details for writing machine code insertions as provided by the predefined package MACHINE_CODE.

The Ada/M user may, if desired, include AN/UYK-44 instructions within an Ada program. This is done by including a procedure in the program which contains only record aggregates defining machine instructions. The package MACHINE_CODE, included in the system program library, contains type, record, and constant declarations which are used to form the instructions. Each field of the aggregate contains a field of the resulting machine instruction. These fields are specified in the order in which they appear in the actual instruction.

A procedure containing machine-code insertions looks similar to this:

```
with machine_code; use machine_code;
procedure machine_samples is
begin
   instr'(OPCODE,A,M,Y); -- first instruction
   instr'(OPCODE,A,M,Y); -- second instruction
   ...
   instr'(OPCODE,A,M,Y); -- last instruction
end;
```

OPCODE, A, M, and Y in all these examples are replaced by the actual opcode, A register, M register, and Y field desired for each AN/UYK-44 instruction. Whenever possible, MACRO/M mnemonics are used to specify the opcode field. The A and M register fields are specified as R0, R1, ... R15. The Y field may be specified by any static expression which will fit in a 16-bit integer. For certain instructions such as unary arithmetic operations, the opcode and either the A or M register determine which instruction is executed. The specification of these instructions and certain others is somewhat more complicated and is explained in detail below. Here are some examples of possible MACRO/M instructions and the Ada/M record aggregates that correspond to them:

MACRO/M	Ada/M
spt A,Y,M	instr'(spt,A,M,Y);
lr A,M	instr'(lr,A,M);
l A,Y,M	instr'(l,A,M,Y);
mi A,M	instr'(mi,A,M);
ork A,Y,M	instr'(ork,A,M,Y);

In some cases, A or M register fields do not appear in the MACRO/M instruction because the field is always zero in the machine instruction. R0 must be used in that field of the record aggregate in Ada/M, however, since no missing fields are allowed. Here are some examples where that occurs:

MACRO/M	Ada/M
lpi M	instr'(lpi,r0,M);
lp Y,M	instr'(lp,r0,M,Y);
sfsc M	instr'(sfsc,r0,M);

Some MACRO/M mnemonics are ambiguous and are assembled into one of two or more opcodes based on the operands specified in the instruction. Ada/M opcode mnemonics must be unambiguous, so either the letter K (indicating an RK format instruction) or the letter X (indicating an RX format instruction) has been added to the end of otherwise ambiguous mnemonics. Some examples of this are as follows:

MACRO/M	Ada/M
jz A,Y,M	instr'(jzk,A,M,Y);
jp A,*Y,M	instr'(jpx,A,M,Y);

For those MACRO/M mnemonics which determine both the opcode and either the A or M register, the MACRO/M mnemonic (disambiguated as above if necessary) is used for the A or M field and an opcode mnemonic is invented. Some examples of this are as follows:

MACRO/M	Ada/M
pr A	instr'(ua_opcode,A,pr);
drtr A	instr'(ua_opcode,A,drtr);
sqr A	instr'(us_opcode,A,sqr);
jne Y,M	instr'(cjk_opcode,jnek,M,Y);
hcr	instr'(ec_opcode,hcr,r0);

The Ada/M user must be able to include data as well as instructions in machine code. The MACHINE_CODE package defines record types which allow the user to create indirect words, signed bytes, unsigned bytes, words, double words, and floating point numbers. The format for including data is as follows:

Data	Ada/M
indirect word (iw J,Y,X)	indirect_word'(J,X,Y);
unsigned byte (0 .. 255)	unsigned_byte_value'(VALUE);
word (16-bit value)	word_value'(VALUE);
double word (32-bit value)	double_word_value'(VALUE);
float value (32-bit value)	float_value'(VALUE);

Table 6-3 contains a list of MACRO/M instructions and their Ada/M machine code equivalents, sorted by MACRO/M mnemonic.

MACRO/M	Ada/M
a A,Y,M	instr'(a,A,M,Y);
acos A	instr'(mf_opcode,A,acos);
acr M	instr'(lpar,r0,M);
ad A,Y,M	instr'(ad,A,M,Y);
adi A,M	instr'(adi,A,M);
adr A,M	instr'(adr,A,M);
ai A,M	instr'(ai,A,M);
ak A,Y,M	instr'(ak,A,M,Y);
ald A,Y,M	instr'(ald,A,M,Y);
aldr A,M	instr'(aldr,A,M);
alog A	instr'(mf_opcode,A,alog);
als A,Y,M	instr'(als,A,M,Y);
alsr A,M	instr'(alsr,A,M);
and A,Y,M	instr'(and,A,M,Y);
andi A,M	instr'(andi,A,M);
andk A,Y,M	instr'(andk,A,M,Y);
andr A,M	instr'(andr,A,M);
ar A,M	instr'(ar,A,M);
ard A,Y,M	instr'(ard,A,M,Y);
ardr A,M	instr'(ardr,A,M);
ars A,Y,M	instr'(ars,A,M,Y);
arsr A,M	instr'(arsr,A,M);
asin A	instr'(mf_opcode,A,asin);
atan A	instr'(mf_opcode,A,atan);
ba A,Y,M	instr'(ba,A,M,Y);
bc A,Y,M	instr'(bc,A,M,Y);
bci A,M	instr'(bci,A,M);
bcx A,Y,M	instr'(bcx,A,M,Y);
bcxi A,M	instr'(bcxi,A,M);
bf Y,M	instr'(bf,r0,M,Y);
bfi M	instr'(bfi,r0,M);
bl A,Y,M	instr'(bl,A,M,Y);
bli A,M	instr'(bli,A,M);
blx A,Y,M	instr'(blx,A,M,Y);
blxi A,M	instr'(blxi,A,M);
bs A,Y,M	instr'(bs,A,M,Y);
bsi A,M	instr'(bsi,A,M);
bsu A,Y,M	instr'(bsu,A,M,Y);
bsx A,Y,M	instr'(bsx,A,M,Y);
bsxi A,M	instr'(bsxi,A,M);
built-in test - dec	instr'(bit_opcode,dec);
built-in test - eec	instr'(bit_opcode,eec);

Table 6-3a - Machine Code Instructions

MACRO/M	Ada/M
built-in test - icp	instr'(bit_opcode,icp);
built-in test - ids	instr'(bit_opcode,ids);
built-in test - imp	instr'(bit_opcode,imp);
built-in test - lrm	instr'(bit_opcode,lrm);
built-in test - rscs	instr'(bit_opcode,rscs);
built-in test - sel	instr'(bit_opcode,sel);
built-in test - srm	instr'(bit_opcode,srm);
c A,Y,M	instr'(c,A,M,Y);
cbr A,M	instr'(cbr,A,M);
ccr A,M	instr'(lpar,A,M);
cd A,Y,M	instr'(cd,A,M,Y);
cdi A,M	instr'(cdi,A,M);
cdr A,M	instr'(cdr,A,M);
ci A,M	instr'(ci,A,M);
ck A,Y,M	instr'(ck,A,M,Y);
cl A,Y,M	instr'(cl,A,M,Y);
cld A,Y,M	instr'(cld,A,M,Y);
cldr A,M	instr'(cldr,A,M);
cli A,M	instr'(cli,A,M);
clk A,Y,M	instr'(clk,A,M,Y);
clr A,M	instr'(clr,A,M);
cls A,Y,M	instr'(cls,A,M,Y);
clsr A,M	instr'(clsr,A,M);
cm A,Y,M	instr'(cm,A,M,Y);
cmi A,M	instr'(cmi,A,M);
cmk A,Y,M	instr'(cmk,A,M,Y);
cmr A,M	instr'(cmr,A,M);
cnt A	instr'(us_opcode,A,cnt);
cos A	instr'(mf_opcode,A,cos);
cr A,M	instr'(cr,A,M);
d A,Y,M	instr'(d,A,M,Y);
data - double word	double_word_value'(VALUE);
data - float	float_value'(VALUE);
data - signed byte	signed_byte_value'(VALUE);
data - unsigned byte	unsigned_byte_value'(VALUE);
data - word	word_value'(VALUE);
dcir	instr'(uc_opcode,r0,dcir);
dcr	instr'(uc_opcode,r0,dcr);
dd A,Y,M	instr'(dd,A,M,Y);
ddi A,M	instr'(ddi,A,M);
ddr A,M	instr'(ddr,A,M);
di A,M	instr'(di,A,M);
dk A,Y,M	instr'(dk,A,M,Y);

Table 6-3b - Machine Code Instructions (Continued)

MACRO/M	Ada/M
dm	instr'(uc_opcode,r0,dm);
dr A,M	instr'(dr,A,M);
dror A	instr'(ua_opcode,A,dror);
drtr A	instr'(ua_opcode,A,drtr);
ecir	instr'(uc_opcode,r0,ecir);
ecr	instr'(uc_opcode,r0,ecr);
er A	instr'(uc_opcode,A,er);
exp A	instr'(mf_opcode,A,exp);
fa A,Y,M	instr'(fa,A,M,Y);
fai A,M	instr'(fai,A,M);
far A,M	instr'(far,A,M);
fc A,Y	instr'(mp_opcode,A,fc); word_value'(Y);
fd A,Y,M	instr'(fd,A,M,Y);
fdi A,M	instr'(fdi,A,M);
fdr A,M	instr'(fdr,A,M);
flc A	instr'(mp_opcode,A,flc);
flcd A	instr'(mp_opcode,A,flcd);
fm A,Y,M	instr'(fm,A,M,Y);
fmi A,M	instr'(fmi,A,M);
fmr A,M	instr'(fmr,A,M);
fsu A,Y,M	instr'(fsu,A,M,Y);
fsui A,M	instr'(fsui,A,M);
fsur A,M	instr'(fsur,A,M);
fxc A	instr'(mp_opcode,A,fxc);
fxcd A	instr'(mp_opcode,A,fxcd);
ib A	instr'(us_opcode,A,ib);
ick A,Y	instr'(e6_opcode,A,ick,Y);
ioc A,Y,M	instr'(iocr,A,M); word_value'(Y);
iocr	instr'(iocr,r0,r0);
iror A	instr'(ua_opcode,A,iror);
irtr A	instr'(ua_opcode,A,irtr);
is A	instr'(us_opcode,A,is);
iw Y,Y,X	indirect_word'(J,X,Y);
j *Y,M	instr'(cjx_opcode,jx,M);
j Y,M	instr'(cjk_opcode,jk,M);
jb *Y,M	instr'(cjx_opcode,jbx,M);
jb Y,M	instr'(cjk_opcode,jbk,M);
jbr M	instr'(cjr_opcode,jbr,M);
jc *Y,M	instr'(cjx_opcode,jcx,M);
jc Y,M	instr'(cjk_opcode,jck,M);
jcr M	instr'(cjr_opcode,jcr,M);
je *Y,M	instr'(cjx_opcode,jex,M);
je Y,M	instr'(cjk_opcode,jek,M);

Table 6-3c - Machine Code Instructions (Continued)

MACRO/M	Ada/M
jer M	instr'(cjr_opcode,jer,M);
jge *Y,M	instr'(cjx_opcode,jgex,M);
jge Y,M	instr'(cjk_opcode,jgek,M);
jger M	instr'(cjr_opcode,jger,M);
jks 1,*Y,M	instr'(cjx_opcode,jksx1,M);
jks 1,Y,M	instr'(cjk_opcode,jksk1,M);
jks 2,*Y,M	instr'(cjx_opcode,jksx2,M);
jks 2,Y,M	instr'(cjk_opcode,jksk2,M);
jksr 1,M	instr'(cjr_opcode,jksr1,M);
jksr 2,M	instr'(cjr_opcode,jksr2,M);
jlm *Y,M	instr'(jlmx,r0,M,Y);
jlm Y,M	instr'(jlmk,r0,M,Y);
jlr A,*Y,M	instr'(jlrx,A,M,Y);
jlr A,Y,M	instr'(jlrk,A,M,Y);
jlrr A,M	instr'(jlrr,A,M);
jls *Y,M	instr'(cjx_opcode,jlsx,M);
jls Y,M	instr'(cjk_opcode,jlsk,M);
jlsr M	instr'(cjr_opcode,jlsr,M);
jn A,*Y,M	instr'(jnx,A,M,Y);
jn A,Y,M	instr'(jnk,A,M,Y);
jne *Y,M	instr'(cjx_opcode,jnex,M);
jne Y,M	instr'(cjk_opcode,jnek,M);
jner M	instr'(cjr_opcode,jner,M);
jnr A,M	instr'(jnr,A,M);
jnz A,*Y,M	instr'(jnzx,A,M,Y);
jnz A,Y,M	instr'(jnzk,A,M,Y);
jnzr A,M	instr'(jnzr,A,M);
jo *Y,M	instr'(cjx_opcode,jox,M);
jo Y,M	instr'(cjk_opcode,jok,M);
jor M	instr'(cjr_opcode,jor,M);
jp A,*Y,M	instr'(jpx,A,M,Y);
jp A,Y,M	instr'(jpk,A,M,Y);
jpr A,M	instr'(jpr,A,M);
jpt *Y,M	instr'(cjx_opcode,jptx,M);
jpt Y,M	instr'(cjk_opcode,jptk,M);
jptr M	instr'(cjr_opcode,jptr,M);
jr M	instr'(cjr_opcode,jr,M);
js *Y,M	instr'(cjx_opcode,jsx,M);
js Y,M	instr'(cjk_opcode,jsk,M);
jsr M	instr'(cjr_opcode,jsr,M);
jz A,*Y,M	instr'(jzx,A,M,Y);
jz A,Y,M	instr'(jzk,A,M,Y);
jzr A,M	instr'(jzr,A,M);

Table 6-3d - Machine Code Instructions (Continued)

MACRO/M	Ada/M
l A,Y,M	instr'(l,A,M,Y);
la A,M	instr'(la,A,M);
lad A,M	instr'(lad,A,M);
lald A,M	instr'(lald,A,M);
lals A,M	instr'(lals,A,M);
lard A,M	instr'(lard,A,M);
lari A,M	instr'(lari,A,M);
larm A,Y,M	instr'(larm,A,M,Y);
larr A,M	instr'(larr,A,M);
lars A,M	instr'(lars,A,M);
lbxi A,Y,M	instr'(lbxi,A,M,Y);
lc A,M	instr'(lc,A,M);
lcep A	instr'(us_opcode,A,lcep);
lclc A,M	instr'(lclc,A,M);
lcld A,M	instr'(lclc,A,M);
lcr A	instr'(uc_opcode,A,lcr);
lcrd A	instr'(uc_opcode,A,lcrd);
ld A,Y,M	instr'(ld,A,Y,M);
ldi A,M	instr'(ldi,A,M);
ldiv A,M	instr'(ldiv,A,M);
ldx A,Y,M	instr'(ldx,A,M,Y);
ldxi A,M	instr'(ldxi,A,M);
lem A	instr'(uc_opcode,A,lem);
li A,M	instr'(li,A,M);
lir A,M	instr'(lir,A,M);
lj D	instr'(lj,D);
lje D	instr'(lje,D);
ljge D	instr'(ljge,D);
lji D	instr'(lji,D);
ljlm D	instr'(ljlm,D);
ljls D	instr'(ljls,D);
ljne D	instr'(ljne,D);
lk A,Y,M	instr'(lk,A,M,Y);
ll A,M	instr'(ll,A,M);
llrd A,M	instr'(llrd,A,M);
llrs A,M	instr'(llrs,A,M);
lm A,Y,M	instr'(lm,A,M,Y);
lmap A,Y,M	instr'(lmap,A,M,Y);
lmr A,Y,M	instr'(lmr,A,M,Y);
lmul A,M	instr'(lmul,A,M);
lp Y,M	instr'(lp,r0,M,Y);
lpa A,Y,M	instr'(lpa,A,M,Y);
lpai A,M	instr'(lpai,A,M);

Table 6-3e - Machine Code Instructions (Continued)

MACRO/M	Ada/M
lpak A,Y,M	instr'(lpak,A,M,Y);
lpar A,M	instr'(lpar,A,M);
lpi M	instr'(lpi,r0,M);
lpl A,Y,M	instr'(lpl,A,M,Y);
lpli A,M	instr'(lpli,A,M);
lpr A	instr'(uc_opcode,A,lpr);
lr A,M	instr'(lr,A,M);
lrd A,Y,M	instr'(lrd,A,M,Y);
lrdr A,M	instr'(lrdr,A,M);
lrs A,Y,M	instr'(lrs,A,M,Y);
lrsr A,M	instr'(lrsr,A,M);
lsor A	instr'(uc_opcode,A,lsor);
lstr A	instr'(uc_opcode,A,lstr);
lsu A,M	instr'(lsu,A,M);
lsud A,M	instr'(lsud,A,M);
lx A,Y,M	instr'(lx,A,M,Y);
lxi A,M	instr'(lxi,A,M);
m A,Y,M	instr'(m,A,M,Y);
mb A,M	instr'(mb,A,M);
mdi A,M	instr'(mdi,A,M);
mdm A,Y,M	instr'(mdm,A,M,Y);
mdr A,M	instr'(mdr,A,M);
mi A,M	instr'(mi,A,M);
mk A,Y,M	instr'(mk,A,M,Y);
mr A,M	instr'(mr,A,M);
ms A,Y,M	instr'(ms,A,M,Y);
msi A,M	instr'(msi,A,M);
msk A,Y,M	instr'(msk,A,M,Y);
msr A,M	instr'(msr,A,M);
nf A	instr'(mp_opcode,A,nf);
nr A	instr'(ua_opcode,A,nr);
ock A,Y	instr'(e6_opcode,A,ock,Y);
ocr A	instr'(ua_opcode,A,ocr);
or A,Y,M	instr'(or,A,M,Y);
ori A,M	instr'(ori,A,M);
ork A,Y,M	instr'(ork,A,M,Y);
orr A,M	instr'(orr,A,M);
pr A	instr'(ua_opcode,A,pr);
qal A,Y	instr'(mp_opcode,A,qal); word_value'(Y);
qar A,Y	instr'(mp_opcode,A,qar); word_value'(Y);
qgt A,Y,M	instr'(qgt,A,M,Y);
qpb A,Y,M	instr'(qpb,A,M,Y);

Table 6-3f - Machine Code Instructions (Continued)

MACRO/M	Ada/M
qpt A,Y,M	instr'(qpt,A,M,Y);
rex Y,M	instr'(rex,r0,M,Y);
rf A	instr'(mp_opcode,A,rf);
rfp A	instr'(mp_opcode,A,rfp);
rh A	instr'(mp_opcode,A,rh);
rhp A	instr'(mp_opcode,A,rhp);
rim A,Y,M	instr'(smap,A,M,Y);
rr A	instr'(ua_opcode,A,rr);
rvr A	instr'(us_opcode,A,rvr);
s A,Y,M	instr'(s,A,M,Y);
sari A,M	instr'(sari,A,M);
sarm A,Y,M	instr'(sarm,A,M,Y);
sarr A,M	instr'(sarr,A,M);
sbr A,M	instr'(sbr,A,M);
sbxi A,Y,M	instr'(sbxi,A,M,Y);
scr A	instr'(uc_opcode,A,scr);
scrd A	instr'(uc_opcode,A,scrd);
sd A,Y,M	instr'(sd,A,M,Y);
sdi A,M	instr'(sdi,A,M);
sdx A,Y,M	instr'(sdx,A,M,Y);
sdxi A,M	instr'(sdxi,A,M);
sedr A,M	instr'(sedr,A,M);
ser A,M	instr'(ser,A,M);
sfr A	instr'(us_opcode,A,sfr);
sgt A,Y,M	instr'(sgt,A,M,Y);
si A,M	instr'(si,A,M);
sin A	instr'(mf_opcode,A,sin);
sir A,M	instr'(sir,A,M);
sm A,Y,M	instr'(sm,A,M,Y);
smap A,Y,M	instr'(smap,A,M,Y);
smc A	instr'(us_opcode,A,smc);
smr A,Y,M	instr'(smr,A,M,Y);
spl A,Y,M	instr'(spl,A,M,Y);
spli A,M	instr'(spli,A,M);
spt A,Y,M	instr'(spt,A,M,Y);
sqr A	instr'(us_opcode,A,sqr);
sqrt A	instr'(us_opcode,A,sqrt);
ssor A	instr'(uc_opcode,A,ssor);
sstr A	instr'(uc_opcode,A,sstr);
su A,Y,M	instr'(su,A,M,Y);
sud A,Y,M	instr'(sud,A,M,Y);
sudi A,M	instr'(sudi,A,M);
sudr A,M	instr'(sudr,A,M);

Table 6-3g - Machine Code Instructions (Continued)

MACRO/M	Ada/M
sui A,M	instr'(sui,A,M);
suk A,Y,M	instr'(suk,A,M,Y);
sur A,M	instr'(sur,A,M);
sx A,Y,M	instr'(sx,A,M,Y);
sxi A,M	instr'(sxi,A,M);
sz Y,M	instr'(sz,r0,M,Y);
szi M	instr'(szi,r0,M);
tan A	instr'(mf_opcode,A,tan);
tcdr A	instr'(ua_opcode,A,tcdr);
tcr A	instr'(ua_opcode,A,tcr);
vf A	instr'(mp_opcode,A,vf);
vfp A	instr'(mp_opcode,A,vfp);
vh A	instr'(mp_opcode,A,vh);
vhp A	instr'(mp_opcode,A,vhp);
wcm A,Y,M	instr'(lmap,A,M,Y);
wcmk AM,Y	instr'(e6_opcode,A,M,Y);
wim A,Y,M	instr'(lmap,A,M,Y);
wimk A,Y,M	instr'(e6_opcode,A,M,Y);
xj A,*Y,M	instr'(xjx,A,M,Y);
xj A,Y,M	instr'(xjk,A,M,Y);
xjr A,M	instr'(xjr,A,M);
xor A,Y,M	instr'(xor,A,M,Y);
xori A,M	instr'(xori,A,M);
xork A,Y,M	instr'(xork,A,M,Y);
xorr A,M	instr'(xorr,A,M);
xsdi A,M	instr'(xsdi,A,M);
xsi A,M	instr'(xsi,A,M);
zbr A,M	instr'(zbr,A,M);

Table 6-3h - Machine Code Instructions (Continued)

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.

```
-- MACRO.DFS                      -- ACVC VERSION 1.10

-- THIS FILE CONTAINS THE MACRO DEFINITIONS USED IN THE ACVC TESTS.
-- THESE DEFINITIONS ARE USED BY THE ACVC TEST PRE-PROCESSOR,
-- MACROSUB. MACROSUB WILL CALCULATE VALUES FOR THOSE MACRO SYMBOLS
-- WHOSE DEFINITIONS DEPEND ON THE VALUE OF MAX_IN_LEN (NAMELY, THE
-- VALUES OF THE MACRO SYMBOLS BIG_ID1, BIG_ID2, BIG_ID3, BIG_ID4,
-- BIG_STRING1, BIG_STRING2, MAX_STRING_LITERAL, BIG_INT_LIT,
-- BIG_REAL_LIT, MAX_LEN_INT_BASED_LITERAL, MAX_LEN_REAL_BASED_LITERAL,
-- AND BLANKS).  THEREFORE, ANY VALUES GIVEN IN THIS FILE FOR THOSE
-- MACRO SYMBOLS WILL BE IGNORED BY MACROSUB.

-- NOTE: THE MACROSUB PROGRAM EXPECTS THE FIRST MACRO IN THIS FILE TO
--       BE MAX_IN_LEN.

-- EACH DEFINITION IS ACCORDING TO THE FOLLOWING FORMAT:

-- A. A NUMBER OF LINES PRECEDED BY THE ADA COMMENT DELIMITER, --.
--    THE FIRST OF THESE LINES CONTAINS THE MACRO SYMBOL AS IT APPEARS
--    IN THE TEST FILES (WITH THE DOLLAR SIGN).  THE NEXT FEW "COMMENT"
--    LINES CONTAIN A DESCRIPTION OF THE VALUE TO BE SUBSTITUTED.
--    THE REMAINING "COMMENT" LINES, THE FIRST OF WHICH BEGINS WITH THE
--    WORDS "USED IN: " (NO QUOTES), CONTAIN A LIST OF THE TEST FILES
--    (WITHOUT THE .TST EXTENSION) IN WHICH THE MACRO SYMBOL APPEARS.
--    EACH TEST FILE NAME IS PRECEDED BY ONE OR MORE BLANKS.
-- B. THE IDENTIFIER (WITHOUT THE DOLLAR SIGN) OF THE MACRO SYMBOL,
--    FOLLOWED BY A SPACE OR TAB, FOLLOWED BY THE VALUE TO BE
--    SUBSTITUTED.  IN THE DISTRIBUTION FILE, A SAMPLE VALUE IS
--    PROVIDED; THIS VALUE MUST BE REPLACED BY A VALUE APPROPRIATE TO
--    THE IMPLEMENTATION.

-- DEFINITIONS ARE SEPARATED BY ONE OR MORE EMPTY LINES.
-- THE LIST OF DEFINITIONS BEGINS AFTER THE FOLLOWING EMPTY LINE.

-- $MAX_IN_LEN
-- AN INTEGER LITERAL GIVING THE MAXIMUM LENGTH PERMITTED BY THE
-- COMPILER FOR A LINE OF ADA SOURCE CODE (NOT INCLUDING AN END-OF-LINE
-- CHARACTER).
-- USED IN:  A26007A
MAX_IN_LEN   120

-- $BIG_ID1
-- AN IDENTIFIER IN WHICH THE NUMBER OF CHARACTERS IS $MAX_IN_LEN.
-- THE MACROSUB PROGRAM WILL SUPPLY AN IDENTIFIER IN WHICH THE
-- LAST CHARACTER IS '1' AND ALL OTHERS ARE 'A'.
-- USED IN:  C23003A  C23003B  C23003C  B23003D  B23003E  C23003G
--           C23003H  C23003I  C23003J  C35502D  C35502F
BIG_ID1      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA1

-- $BIG_ID2
-- AN IDENTIFIER IN WHICH THE NUMBER OF CHARACTERS IS $MAX_IN_LEN,
-- DIFFERING FROM $BIG_ID1 ONLY IN THE LAST CHARACTER.  THE MACROSUB
-- PROGRAM WILL USE '2' AS THE LAST CHARACTER.
-- USED IN:  C23003A  C23003B  C23003C  B23003F  C23003G  C23003H
--           C23003I  C23003J
BIG_ID2      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA2

-- $BIG_ID3
-- AN IDENTIFIER IN WHICH THE NUMBER OF CHARACTERS IS $MAX_IN_LEN.
-- MACROSUB WILL USE '3' AS THE "MIDDLE" CHARACTER; ALL OTHERS ARE 'A'.
-- USED IN:  C23003A  C23003B  C23003C  C23003G  C23003H  C23003I
--           C23003J
```

```
-- $BIG ID4
-- AN IDENTIFIER IN WHICH THE NUMBER OF CHARACTERS IS $MAX IN LEN,
-- DIFFERING FROM $BIG ID3 ONLY IN THE MIDDLE CHARACTER.  MACROSUB
-- WILL USE '4' AS THE MIDDLE CHARACTER.
-- USED IN:  C23003A  C23003B  C23003C  C23003G  C23003H  C23003I
--           C23003J
BIG ID4      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

-- $BIG STRING1
-- A STRING LITERAL (WITH QUOTES) WHOSE CATENATION WITH $BIG STRING2
-- ($BIG STRING1 & $BIG STRING2) PRODUCES THE IMAGE OF $BIG ID1.
-- USED IN:  C35502D  C35502F
BIG STRING1 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

-- $BIG STRING2
-- A STRING LITERAL (WITH QUOTES) WHOSE CATENATION WITH $BIG STRING1
-- ($BIG STRING1 & $BIG STRING2) PRODUCES THE IMAGE OF $BIG ID1.
-- USED IN:  C35502D  C35502F
BIG STRING2 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA1"

-- $MAX STRING LITERAL
-- A STRING LITERAL CONSISTING OF $MAX IN LEN CHARACTERS (INCLUDING THE
-- QUOTE CHARACTERS).
-- USED IN:  A26007A
MAX_STRING_LITERAL "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

-- $NEG BASED INT
-- A BASED INTEGER LITERAL (PREFERABLY BASE 8 OR 16) WHOSE HIGHEST ORDER
-- NON-ZERO BIT WOULD FALL IN THE SIGN BIT POSITION OF THE
-- REPRESENTATION FOR SYSTEM.MAX INT, I.E., AN ATTEMPT TO WRITE A
-- NEGATIVE VALUED LITERAL SUCH AS -2 BY TAKING ADVANTAGE OF THE
-- BIT REPRESENTATION.
-- USED IN:  E24201A
NEG_BASED_INT        16#FFFE#

-- $BIG INT LIT
-- AN INTEGER LITERAL WHOSE VALUE IS 298, BUT WHICH HAS
-- ($MAX IN LEN - 3) LEADING ZEROES.
-- USED IN:  C24003A
BIG INT LIT  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000298

-- $BIG REAL LIT
-- A UNIVERSAL REAL LITERAL WHOSE VALUE IS 690.0, BUT WHICH HAS
-- ($MAX_IN_LEN - 5) LEADING ZEROES.
-- USED IN:  C24003B  C24003C
BIG REAL LIT        0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000690.0

-- $MAX LEN INT BASED LITERAL
-- A BASED INTEGER LITERAL (USING COLONS) WHOSE VALUE IS 2:11:, HAVING
-- ($MAX IN LEN - 5) ZEROES BETWEEN THE FIRST COLON AND THE FIRST 1.
-- USED IN:  C2A009A
MAX LEN INT BASED LITERAL   2:0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011:

-- $MAX LEN REAL BASED LITERAL
-- A BASED REAL LITERAL (USING COLONS) WHOSE VALUE IS 16:F.E:, HAVING
-- ($MAX IN LEN - 7) ZEROES BETWEEN THE FIRST COLON AND THE F.
-- USED IN:  C2A009A
MAX LEN REAL BASED LITERAL 16:00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000F.E:

-- $BLANKS
-- A SEQUENCE OF ($MAX IN LEN - 20) BLANKS.
```

```
--           B22001N
--
--             10        20        30        40        50        60        70        80        90       100
--     123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
--    <          LIMITS OF SAMPLE SHOWN BY ANGLE BRACKETS                                                    >
BLANKS

-- $MAX DIGITS
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MAX DIGITS.
-- USED IN:  B35701A  CD7102B
MAX_DIGITS   6

-- $NAME
-- THE NAME OF A PREDEFINED INTEGER TYPE OTHER THAN INTEGER,
-- SHORT_INTEGER, OR LONG_INTEGER.
-- (IMPLEMENTATIONS WHICH HAVE NO SUCH TYPES SHOULD USE AN UNDEFINED
-- IDENTIFIER SUCH AS   NO SUCH TYPE AVAILABLE.)
-- USED IN:  AVAT007  C45231D  B86001X C7D101G
NAME NO_SUCH_INTEGER_TYPE

-- $FLOAT_NAME
-- THE NAME OF A PREDEFINED FLOATING POINT TYPE OTHER THAN FLOAT,
-- SHORT_FLOAT, OR LONG_FLOAT.  (IMPLEMENTATIONS WHICH HAVE NO SUCH
-- TYPES SHOULD USE AN UNDEFINED IDENTIFIER SUCH AS NO_SUCH_TYPE.)
-- USED IN:  AVAT013  B86001Z
FLOAT_NAME   NO_SUCH_FLOAT_TYPE

-- $FIXED_NAME
-- THE NAME OF A PREDEFINED FIXED POINT TYPE OTHER THAN DURATION.
-- (IMPLEMENTATIONS WHICH HAVE NO SUCH TYPES SHOULD USE AN UNDEFINED
-- IDENTIFIER SUCH AS NO_SUCH_TYPE.)
-- USED IN:  AVAT015  B86001Y
FIXED_NAME   NO_SUCH_FIXED_TYPE

-- $INTEGER_FIRST
-- AN INTEGER LITERAL, WITH SIGN, WHOSE VALUE IS INTEGER'FIRST.
-- THE LITERAL MUST NOT INCLUDE UNDERSCORES OR LEADING OR TRAILING
-- BLANKS.
-- USED IN:  C35503F  B54B01B
INTEGER_FIRST      -32768

-- $INTEGER_LAST
-- AN INTEGER LITERAL WHOSE VALUE IS INTEGER'LAST.  THE LITERAL MUST
-- NOT INCLUDE UNDERSCORES OR LEADING OR TRAILING BLANKS.
-- USED IN:  C35503F  C45232A  B45B01B
INTEGER_LAST        32767

-- $INTEGER_LAST_PLUS_1
-- AN INTEGER LITERAL WHOSE VALUE IS INTEGER'LAST + 1.
-- USED IN:  C45232A
INTEGER_LAST_PLUS_1 32_768

-- $MIN_INT
-- AN INTEGER LITERAL, WITH SIGN, WHOSE VALUE IS SYSTEM.MIN_INT.
-- THE LITERAL MUST NOT CONTAIN UNDERSCORES OR LEADING OR TRAILING
-- BLANKS.
-- USED IN:  C35503D  C35503F  CD7101B
MIN_INT      -2147483648

-- $MAX_INT
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MAX_INT.
-- THE LITERAL MUST NOT INCLUDE UNDERSCORES OR LEADING OR TRAILING
```

```
MAX INT      2147483647

-- $MAX INT PLUS 1
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MAX INT + 1.
-- USED IN:  C45232A
MAX INT PLUS 1       2147483648

-- $LESS THAN DURATION
-- A REAL LITERAL (WITH SIGN) WHOSE VALUE (NOT SUBJECT TO
-- ROUND-OFF ERROR IF POSSIBLE) LIES BETWEEN DURATION'BASE'FIRST AND
-- DURATION'FIRST.  IF NO SUCH VALUES EXIST, USE A VALUE IN
-- DURATION'RANGE.
-- USED IN:  C96005B
LESS THAN DURATION  -131071.5

-- $GREATER THAN DURATION
-- A REAL LITERAL WHOSE VALUE (NOT SUBJECT TO ROUND-OFF ERROR
-- IF POSSIBLE) LIES BETWEEN DURATION'BASE'LAST AND DURATION'LAST.  IF
-- NO SUCH VALUES EXIST, USE A VALUE IN DURATION'RANGE.
-- USED IN:  C96005B
GREATER THAN DURATION     131071.5

-- $LESS THAN DURATION BASE FIRST
-- A REAL LITERAL (WITH SIGN) WHOSE VALUE IS LESS THAN
-- DURATION'BASE'FIRST.
-- USED IN:  C96005C
LESS THAN DURATION BASE FIRST      -131073.0

-- $GREATER THAN DURATION BASE LAST
-- A REAL LITERAL WHOSE VALUE IS GREATER THAN DURATION'BASE'LAST.
-- USED IN: C96005C
GREATER THAN DURATION BASE LAST   131073.0

-- $COUNT LAST
-- AN INTEGER LITERAL WHOSE VALUE IS TEXT_IO.COUNT'LAST.
-- USED IN:  CE3002B
COUNT LAST   32767

-- $FIELD LAST
-- AN INTEGER LITERAL WHOSE VALUE IS TEXT IO.FIELD'LAST.
-- USED IN:  CE3002C
FIELD LAST   32767

-- $ILLEGAL EXTERNAL FILE NAME1
-- AN ILLEGAL EXTERNAL FILE NAME (E.G., TOO LONG, CONTAINING INVALID
-- CHARACTERS, CONTAINING WILD-CARD CHARACTERS, OR SPECIFYING A
-- NONEXISTENT DIRECTORY).
-- USED IN:  CE2103A  CE2102C  CE2102H  CE2103H  CE3102B  CE3107A
ILLEGAL EXTERNAL FILE NAME1 BAD-CHARS^#.%!X@*()+&^%$#@!@

-- $ILLEGAL EXTERNAL FILE NAME2
-- AN ILLEGAL EXTERNAL FILE NAME, DIFFERENT FROM $EXTERNAL FILE NAME1.
-- USED IN:  CE2102C  CE2102H  CE2103A  CE2103B
ILLEGAL EXTERNAL FILE NAME2 ANOTHER BAD-CHARS^#.%!X@*()+&^%$#@!@

-- $ACC SIZE
-- AN INTEGER LITERAL WHOSE VALUE IS THE MINIMUM NUMBER OF BITS
-- SUFFICIENT TO HOLD ANY VALUE OF AN ACCESS TYPE.
-- USED IN:  CD2A81A  CD2A81B  CD2A81C  CD2A81D  CD2A81E
--           CD2A81F  CD2A81G  CD2A83A  CD2A83B  CD2A83C  CD2A83E
--           CD2A83F  CD2A83G  ED2A86A  CD2A87A
ACC SIZE     16
```

```
-- 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.
-- USED IN:  CD2A91A  CD2A91B  CD2A91C  CD2A91D  CD2A91E
TASK_SIZE     32

-- $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.
-- USED IN:  CD2A96A
MIN_TASK_SIZE         32

-- $NAME_LIST
-- A LIST OF THE ENUMERATION LITERALS IN THE TYPE SYSTEM.NAME, SEPARATED
-- BY COMMAS.
-- USED IN:  CD7003A
NAME_LIST     ANUYK44,ANAYK14

-- $DEFAULT_SYS_NAME
-- THE VALUE OF THE CONSTANT SYSTEM.SYSTEM_NAME.
-- USED IN:  CD7004A  CD7004C  CD7004D
DEFAULT_SYS_NAME      ANUYK44

-- $NEW_SYS_NAME
-- A VALUE OF THE TYPE SYSTEM.NAME, OTHER THAN $DEFAULT_SYS_NAME.  IF
-- THERE IS ONLY ONE VALUE OF THE TYPE, THEN USE THAT VALUE.
-- NOTE:  IF THERE ARE MORE THAN TWO VALUES OF THE TYPE, THEN THE
--        PERTINENT TESTS ARE TO BE RUN ONCE FOR EACH ALTERNATIVE.
-- USED IN:  ED7004B1  CD7004C
NEW_SYS_NAME ANAYK14

-- $DEFAULT_STOR_UNIT
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.STORAGE_UNIT.
-- USED IN:  CD7005B  ED7005D3M  CD7005E
DEFAULT_STOR_UNIT    16

-- $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 THE VALUE OF
-- $SYSTEM.STORAGE_UNIT.  IF THERE IS MORE THAN ONE ALTERNATIVE,
-- THEN THE PERTINENT TESTS SHOULD BE RUN ONCE FOR EACH ALTERNATIVE.
-- USED IN:  ED7005C1  ED7005D1  CD7005E
NEW_STOR_UNIT         16

-- $DEFAULT_MEM_SIZE
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MEMORY_SIZE.
-- USED IN:  CD7006B  ED7006D3M  CD7006E
DEFAULT_MEM_SIZE      65_536

-- $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.  IF THERE IS MORE THAN
-- ONE ALTERNATIVE, THEN THE PERTINENT TESTS SHOULD BE RUN ONCE FOR
-- EACH ALTERNATIVE.  IF THE NUMBER OF PERMITTED VALUES IS LARGE, THEN
-- SEVERAL VALUES SHOULD BE USED, COVERING A WIDE RANGE OF
-- POSSIBILITIES.
-- USED IN:  ED7006C1  ED7006D1  CD7006E
NEW_MEM_SIZE 65_536

-- $LOW_PRIORITY
```

```
-- USED IN:  CD7007C
LOW PRIORITY       0

-- $HIGH PRIORITY
-- AN INTEGER LITERAL WHOSE VALUE IS THE UPPER BOUND OF THE RANGE
-- FOR THE SUBTYPE SYSTEM.PRIORITY.
-- USED IN:  CD7007C
HIGH PRIORITY      15

-- $MANTISSA DOC
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MAX MANTISSA AS SPECIFIED
-- IN THE IMPLEMENTOR'S DOCUMENTATION.
-- USED IN:  CD7013B
MANTISSA DOC 31

-- $DELTA DOC
-- A REAL LITERAL WHOSE VALUE IS SYSTEM.FINE DELTA AS SPECIFIED IN THE
-- IMPLEMENTOR'S DOCUMENTATION.
-- USED IN:  CD7013D
DELTA DOC    2#0.0000 0000 0000 0000 0000 0000 0000 0001#

-- $TICK
-- A REAL LITERAL WHOSE VALUE IS SYSTEM.TICK AS SPECIFIED IN THE
-- IMPLEMENTOR'S DOCUMENTATION.
-- USED IN:  CD7104B
TICK 0.00003125
```

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

U.S. NAVY

Compiler:	Ada/M, Version 2.0 (/OPTIMIZE Option)
ACVC Version:	1.10

10.2 Options

Options control the type of processing the compiler performs. They enable the selection of listings produced as part of the compilation process, make special processing requests, and indicate when special compilation units are being compiled. The compiler options, their functions and defaults are summarized in Table 10-1. Each option may be specified as shown or preceded by the three characters NO_ to specify the opposite option. For example, SOURCE turns the source listing on; NO_SOURCE turns the source listing off.

The Compiler produces a diagnostic of severity level WARNING if any of the following conditions are encountered during the processing of options:

a. The complement of an option already specified is specified. The first option will be ignored. For example, if NO_SOURCE is specified, then SOURCE is specified later in the option list, SOURCE will be in effect;

b. An option already specified is re-specified. The first option will be ignored; or

c. An undefined option is specified.

There is no examination of options to determine whether redundant combinations of options are specified. For example, specifying both NO_SOURCE and NO_PRIVATE will not result in a diagnostic.

Some options impact the speed with which the compilation process is completed and the efficiency of the object code produced by the Compiler. The remainder of this section discusses the implications of options for the compilation process, how options affect the quality of object code generated by the Compiler, and guidelines for using them.

Option	Function
Listing Control Options:	
ATTRIBUTE	Produce a symbol attribute listing. (Produces an attribute cross-reference listing when both ATTRIBUTE and CROSS_REFERENCE are specified.) Default: NO_ATTRIBUTE.
CROSS_REFERENCE	Produce a cross-reference listing. (Produces an attribute cross-reference listing when both ATTRIBUTE and CROSS_REFERENCE are specified.) Default: NO_CROSS_REFERENCE.
DIAGNOSTICS	Produce a diagnostic summary listing. Default: NO_DIAGNOSTICS.
MACHINE	Produce a machine code listing if code generated. Code is generated when CONTAINER_GENERATION is in effect and there are no diagnostics of severity ERROR, SYSTEM, or FATAL; and if there are diagnostics of severity level WARNING and CODE_ON_WARNING is in effect. If a machine code listing is requested and no code is generated, a diagnostic of severity NOTE is reported. Default: NO_MACHINE.
NOTES	Include diagnostics of severity NOTE in the source listings and in the diagnostic summary listing. Default: NO_NOTES.
PRIVATE	If there is a source listing, text in the private part of a package specification is to be listed in accordance with the selected SOURCE option, subject to requirements of LIST pragmas. Default: PRIVATE.
SOURCE	Produce a listing of the source text. Default: NO_SOURCE.
SUMMARY	Produce a summary listing, always produced when there are errors in the compilation. Default: NO_SUMMARY

Table 10-1a - Ada Compiler Options

Option	Function
Special Processing Options:	
CHECKS	Provide runtime error checking. NO_CHECKS suppresses all runtime error checking. Please refer to the Pragma SUPPRESS description for further information on runtime error checking. Default: CHECKS.
CODE_ON_WARNING	Generate code (and, if requested, a machine code listing) when there are diagnostics of severity level WARNING, provided there are no FATAL, SYSTEM, or ERROR diagnostics. NO_CODE_ON_WARNING means generate no code (and, if requested, no machine code listing) when there are diagnostics of severity level WARNING. Default: CODE_ON_WARNING.
CONTAINER_GENERATION	Produce a container if diagnostic severity permits. NO_CONTAINER_GENERATION means that no container is to be produced, regardless of diagnostic severity. If a container is not produced because NO_CONTAINER_GENERATION is in effect, code is not generated (nor is a machine code listing, if requested). Default: CONTAINER_GENERATION.
DEBUG	Generates debugger symbolic information and, as required, changes the code being generated If NO_DEBUG is specified, the compiler output includes only that information needed to link, export, and execute the current unit. Default: NO_DEBUG. This option is ignored for a unit that: o is a package or subprogram specification, o is a subprogram body for which there is no previous declaration, or o contains a body stub, pragma INLINE, generic declaration, or a generic body. A diagnostic of severity NOTE is issued when the option is ignored.

Table 10-1b - Ada Compiler Options (Continued)

Option	Function
Special Processing Options (continued):	
EXECUTIVE	Enable pragma EXECUTIVE and allow visibility to units which have been compiled with the /RTE_ONLY option. Default: NO_EXECUTIVE.
MEASURE	Deferred.
OPTIMIZE	Enable global optimizations in accordance with the optimization pragmas specified in the source program. Default: NO_OPTIMIZE When NO_OPTIMIZE is in effect, no global optimizations are performed, regardless of pragmas specified. The OPTIMIZE option enables global optimization. The goals of global optimization may be influenced by the user through the Ada-defined OPTIMIZE pragma. If TIME is specified, the global optimizer concentrates on optimizing execution time. If SPACE is specified, the global optimizer concentrates on optimizing code size. If the user does not include pragma OPTIMIZE, the optimizations emphasize TIME over SPACE. If NO_OPTIMIZE is in effect, no optimizations are performed, regardless of the pragma.
RTE_ONLY	Restrict visibility of this unit only to those units compiled with the /EXECUTIVE option. Default: NO_RTE_ONLY.

Table 10-1c - Ada Compiler Options (Continued)