UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A219 441

(2)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
DTIC FILE COPY	
4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
Ada Compiler Validation Summary Report: Loral/Rolm Mil-Spec Computers ADE, Revision 3.01, MV 10000 (Host) to HAWK/32 (Target), 89080wS1.10141	04 Aug. 1989 to 01 Dec. 1990
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
National Institute of Standards and Technology Gaithersburg, Maryland, USA	
9. PERFORMING ORGANIZATION AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
National Institute of Standards and Technology Gaithersburg, Maryland, USA	
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE
Ada Joint Program Office **United States Department of Defense** **Washington, DC 20301-3081**	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office)	15. SECURITY CLASS (of this report)
National Institute of Standards and Technology Gaithersburg, Maryland, USA	UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

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

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

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

Loral/Rolm Mil-Spec Computers, Gaithersburg, Maryland, ADE Revision 3.01, MV 10000 under AOS/VS 7.64 (Host) to HAWK/32 under AOS/VS 7.64 (Target), ACVC 1.10.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AVF Control Number: NIST89ROL535_1_1.10
PRE-VALIDATION: 19 JULY 1989
ON-SITE: 04 AUGUST 1989
LAST REVISION: 14 DECEMBER 1989
LAST REVISION: 04 JANUARY 1990

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890804S1.10141
Loral/Rolm Mil-Spec Computers
ADE, Revision 3.01
MV 10000 Host and HAWK/32 Target

Completion of On-Site Testing:
4 August 1989

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

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

AVF Control Number: NIST89ROL535_1_1.10

Ada Compiler Validation Summary Report:

Compiler Name: ADE Revision 3.01

Certificate Number: 890804S1.10141

Host: MV 10000 under AOS/VS 7.64

Target: HAWK/32 under AOS/VS 7.64

Testing Completed 4 August 1989 Using ACVC 1.10

This report has been reviewed and is approved.

for

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

Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group
Engineering Division
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

for

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

Accesion For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

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

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report. The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

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

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

Testing of this compiler was conducted by GEMMA Corp under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 4 August 1989 at Loral/Rolm Mil-Spec Computers.

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
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

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

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

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada	An Ada Commentary contains all information relevant to the Commentary point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and

technical support for Ada validations to ensure consistent practices.

Compiler — A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

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

Host — The computer on which the compiler resides.

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

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

Target — The computer which executes the code generated by the compiler.

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

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

1.5 ACVC TEST CLASSES

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

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test

to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

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

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

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

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

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

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED,

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

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contair 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: ADE Revision 3.01

ACVC Version: 1.10

Certificate Number: 890804S1.10141

Host Computer:

Machine: MV 10000

Operating System: AOS/VS 7.64

Memory Size: 16 MBytes

Target Computer:

Machine: L.RMSC HAWK/32

Operating System: AOS/VS Revision 7.64

Memory Size: 8 MBytes

2.2 IMPLEMENTATION CHARACTERISTICS

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

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K..)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined types SHORT_INTEGER (B86001V) and LONG_FLOAT (B86001U) in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Based literals.

(1) An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR when a value exceeds SYSTEM.MAX_INT. This implementation raises NUMERIC_ERROR during execution. (See test E24201A.)

d. Expression evaluation.

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

(1) None of the default initialization expressions for record components are evaluated before any value is checked for

membership in a component's subtype. (See test C32117A.)

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

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

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

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

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

e. Rounding.

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

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

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

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

f. Array types.

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

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

(2) NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

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

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises 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) 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.)

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

g. 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.)

h. 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.)

i. 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, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate

does not belong to an index subtype. (See test E43211B.)

j. 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).)

k. Generics.

(1) Generic specifications and bodies cannot be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic subprogram declarations and bodies cannot be compiled in separate compilations. (See tests CA1012A and CA2009F.)

(3) Generic library subprogram specifications and bodies cannot be compiled in separate compilations. (See test CA1012A.)

(4) Generic non-library package bodies as subunits cannot be compiled in separate compilations. (See test CA2009C.)

(5) Generic non-library subprogram bodies cannot be compiled in separate compilations from their stubs. (See test CA2009F.)

(6) Generic unit bodies and their subunits cannot be compiled in separate compilations. (See test CA3011A.)

(7) Generic package declarations and bodies cannot be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

(8) Generic library package specifications and bodies cannot be compiled in separate compilations. (See tests BC3204C and BC3205D.)

(9) Generic unit bodies and their subunits cannot be compiled in separate compilations. (See test CA3011A.)

l. 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 and record types with discriminants without defaults. (See tests AE2101H,

EE2401D, and EE2401G.)

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

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

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

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

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

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

(9) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

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

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

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

(13) More than one internal file can be associated with each external file for sequential files when writing or reading. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

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

(15) More than one internal file can be associated with each external file for text files when writing or reading. (See tests CE3111A, CE31111D..E (2 tests), and CE3114B.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

**

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 572 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for one test was 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	124	1129	1768	15	21	44	3101
Inapplicable	5	9	547	2	7	2	572
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	194	573	533	245	172	99	158	331	131	36	250	90	289	3101
Inapplicable	18	76	147	3	0	0	8	1	6	0	2	279	32	572
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, 572 tests were inapplicable for the reasons indicated:

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

C24113L..Y (14 tests) C35705L..Y (14 tests)

C35706L..Y (14 tests) C35707L..Y (14 tests)
C35708L..Y (14 tests) C35802L..Z (15 tests)
C45241L..Y (14 tests) C45321L..Y (14 tests)
C45421L..Y (14 tests) C45521L..Z (15 tests)
C45524L..Z (15 tests) C45621L..Z (15 tests)
C45641L..Y (14 tests) C46012L..Z (15 tests)

The following 4 tests are not applicable because the tests require a source line of characters which is greater than the 120 character source line which this implementation does support:

C24113H..K (4 tests)

The following 34 tests are not applicable because 'SIZE representation clauses for enumeration types are not supported:

A39005B CD1009B CD1009P CD2A21A CD2A21B
CD2A21C CD2A21D CD2A21E CD2A22A CD2A22B
CD2A22C CD2A22D CD2A22E CD2A22F CD2A22G
CD2A22H CD2A22I CD2A22J CD2A23A CD2A23B
CD2A23C CD2A23D CD2A23E CD2A24A CD2A24B
CD2A24C CD2A24D CD2A24E CD2A24F CD2A24G
CD2A24H CD2A24I CD2A24J ED2A26A

C34006D is not applicable because use of record descriptors for arrays gives larger 'SIZE for array.

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

The following 14 tests are not applicable because 'STORAGE_SIZE not supported:

A39005C C87B62B CD1009J CD1009R CD1009S
CD1C03C CD2B11B CD2B11C CD2B11D CD2B11E
CD2B11F CD2B11G CD2B15B CD2B16A

The following 7 tests are not supported because 'SMALL representation clauses are not supported:

A39005E C87B62C CD1009L CD1C03F CD2D11A
CD2D11B CD2D13A

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

C45231C C45304C C45502C C45503C C45504C
C45504F C45611C C45613C C45614C C45631C
C45632C B52004D C55B07A B55B09C B86001W
CD7101F

C45531I..P (8 tests) and C45532I..P (8 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 11.

C4A013B is not applicable because the evaluation of an expression involving 'MACHINE_RADIX applied to the most precise floating-point type would raise an exception; since the expression must be static, it is rejected at compile time.

D4A002B and D4A004B use 64-bit integer calculations which are not supported by this compiler.

B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER or SHORT_INTEGER.

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

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

The following 24 tests are not applicable because 'SIZE representation clauses for integer types are not supported:

C87B62A	CD1009A	CD1009O	CD1C03A	CD1C04A
CD2A31A	CD2A31B	CD2A31C	CD2A31D	CD2A32A
CD2A32B	CD2A32C	CD2A32D	CD2A32E	CD2A32F
CD2A32G	CD2A32H	CD2A32I	CD2A32J	CD2A64B
CD2A64D	CD2A65B	CD2A65D	CD2A74B	

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

CA1012A, CA2009C, CA2009F, CA3011A, BC3204C, BC3204D, LA5008M and LA5008N are not applicable because this implementation does not permit compilation in separate files of generic specifications and bodies or of specifications and bodies of subunits of generic units.

The following 16 tests are not applicable because 'SIZE representation clauses for floating-point types are not supported:

CD1009C	CD2A41A	CD2A41B	CD2A41C	CD2A41D
CD2A41E	CD2A42A	CD2A42B	CD2A42C	CD2A42D
CD2A42E	CD2A42F	CD2A42G	CD2A42H	CD2A42I
CD2A42J				

The following 31 tests are not applicable because 'SIZE representation clauses for fixed-point types are not supported:

CD1009D CD1009Q CD1C04C 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
ED2A86A

The following 21 tests are not applicable because 'SIZE representation clauses for array types are not supported:

CD1009E CD1009F CD2A61A CD2A61B CD2A61C
CD2A61D CD2A61E CD2A61F CD2A61G CD2A61H
CD2A61I CD2A61J CD2A61K CD2A61L CD2A62A
CD2A62B CD2A62C CD2A64A CD2A64C CD2A65A
CD2A65C

The following 16 tests are not applicable because 'SIZE representation clauses for record types are not supported:

CD1009G CD2A71A CD2A71B CD2A71C CD2A71D
CD2A72A CD2A72B CD2A72C CD2A72D CD2A74A
CD2A74C CD2A74D CD2A75A CD2A75B CD2A75C
CD2A75D

The following 1 test is not applicable because 'SIZE representation clauses for private types are not supported:

CD1009H

The following 1 test is not applicable because 'SIZE representation clauses for limited private types are not supported:

CD1009I

The following 22 tests are not applicable because 'SIZE representation clauses for access types are not supported:

CD2A81A CD2A81B CD2A81C CD2A81D CD2A81E
CD2A81F CD2A83A CD2A83B CD2A83C CD2A83E
CD2A83F CD2A84B CD2A84C CD2A84D CD2A84E
CD2A84F CD2A84G CD2A84H CD2A84I CD2A84K
CD2A84L CD2A87A

The following 5 tests are not applicable because 'SIZE representation clauses for task types are not supported:

CD2A91A CD2A91B CD2A91C CD2A91D CD2A91E

The following 12 tests are not applicable because of

restrictions on the use of enumeration types for which an enumeration representation clause has been given:

CD3014A	CD3014B	CD3014D	CD3014E	CD3015A
CD3015B	CD3015D	CD3015E	CD3015G	CD3015I
CD3015J	CD3015L			

CD4031A, CD4051C, and CD4051D are not applicable because record representation clauses are not supported for record types with discriminant parts.

The following 46 tests are not applicable because, for this implementation, SYSTEM.ADDRESS clauses for variables are not supported:

CD5003B..I (8 tests)		CD5011A	CD5011C	CD5011E
CD5011G	CD5011I	CD5011K	CD5011M	CD5011Q
CD5012A..B	CD5012E..F	CD5012I..J	CD5012M	CD5013A
CD5013C	CD5013E	CD5013G	CD5013I	CD5013K
CD5013M	CD5013O	CD5013S	CD5014A	CD5014C
CD5014E	CD5014G	CD5014I	CD5014K	CD5014M
CD5014O	CD5014S..T	CD5014V	CD5014X..Z (3 tests)	

The following 30 tests are not applicable because, for this implementation, SYSTEM.ADDRESS clauses for constants are not supported:

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

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.

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

CE2107G, CE2107H, CE2111H, CE3111B, and CE3115A are not applicable because they wrongly assume that input operations are not buffered.

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

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

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

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

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

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

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 one test CC1223A. CC1223A was modified according to AVO instructions to replace the expression "2**T'MANTISSA-1" at line 262 with "2**(T'MANTISSA-1)-1)"; the original expression raised an exception because 2**T'MANTISSA exceeds SYSTEM.MAX_INT.

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 ADE Revision 3.01 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 ADE Revision 3.01 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:	MV 10000
Host operating system:	AOS/VS 7.64
Target computer:	HAWK/32
Target operating system:	AOS/VS Revision 7.64

A tape containing all tests except for withdrawn tests and tests

requiring unsupported floating-point precisior was taken on-site by the validation team for processing.

The contents of the tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the HAWK/32 under AOS/VS 7.64. Results were printed from the MV 10000 computer.

The compiler was tested using command scripts provided by Loral/Rolm Mil-Spec Computers and reviewed by the validation team. See Appendix E for a complete listing of the available compiler options for this implementation. The only option invoked during this validation was:

MAIN_PROGRAM.

Tests were compiled, linked, and executed (as appropriate) using one host computer, the MV 10000, and one target computer, the HAWK/32 under AOS/VS 7.64. Test output, compilation listings, and job logs were captured on tape and archived at the AVF.

3.7.3 Test Site

Testing was conducted at Loral/Rolm Mil-Spec Computers and was completed on 04 August 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Loral/Rolm Mil-Spec Computers has submitted the following Declaration of Conformance concerning the

HOST: MV 10000

TARGET: HAWK/32 under AOS/VS 7.64.

Attachment 4

DECLARATION OF CONFORMANCE

Compiler Implementer: Loral/Rolm Mil-Spec Computers
Ada Validation Facility: Institute for Computer Sci.and Techn.
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name:	ADE		Revision:	3.01
Host Architecture - ISA:	MV 10,000	OS&VER #:	AOS/VS	7.64
Target Architecture - ISA:	HAWK/32	OS&VER #:	AOS/VS	7.64
Target Architecture - ISA:	HAWK/32	OS&VER #:	ARTS/32	2.71

Derived Compiler Registration

Derived Compiler Name:	ADE		Revision:	3.01
Host Architecture - ISA:	MV Family	OS&VER #:	AOS/VS	7.64
Target Architecture - ISA:	MV Family	OS&VER #:	AOS/VS	7.64
Target Architecture - ISA:	HAWK/32	OS&VER #:	AOS/VS	7.64
Target Architecture - ISA:	HAWK/32	OS&VER #:	ARTS/32	2.71

Owner / Implementer's Declaration

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

Jon Elliott

Jon Elliott - Software Product Manager 17 Oct '89
Date

Owners Declaration

I, the undersigned, representing Data General Corporation agree that as part of the joint Marketing Agreement between Rolm Mil-Spec and Data General for the Ada Development Environment, Data General has the responsibility to maintain the Base Compiler listed above. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Deborah Hel

17 Nov '89
Date

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 ADE Revision 3.01 compiler, as described in this Appendix, are provided by Loral/Rolm Mil-Spec Computers. 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 -2_147_483_648  .. 2_147_483_647;
 type SHORT_INTEGER is range -32768 .. 32767;

 type FLOAT is digits 6 range
 -16#0.10000000000000# * 16 **(-64) .. 16#0.FFFFF# * 16 **(63);
 type LONG_FLOAT is digits 15 range
 -16#0.10000000000000# * 16 **(-64)..16#0.FFFFFFFFFFFFFF# * 16 **(63);


 type DURATION is delta 2.0**(-9) range -2**22 .. 2**22;

...

end STANDARD;
```

Addendum to the ANSI Reference Manual for the Ada® Programming Language

086-000070-02

This addendum updates manual 069-000073-00 .
See updating instructions inside.

Ordering No.086-000070
Rev. 02, December 1988

Notice

Addendum to the ANSI Reference Manual for the Ada® Programming Language

086-000070-02

Effective with: AOS/VS ADE, Rev. 3.00

Please insert Appendix F in your copy of the *ANSI Reference Manual for the Ada Programming Language.*

Appendix F: Implementation-Dependent Characteristics

The DGC Ada compiler is distributed as part of the Ada Development Environment (ADE). DGC Ada is a validated implementation of Ada that conforms to the full ANSI/MIL-STD-1815A standard. The ANSI standard allows individual implementations to set or define certain language characteristics, such as pragmas, restrictions on representation clauses, and capacity limits.

This appendix describes the language characteristics defined by the DGC implementation, version 3.00 or higher. In this appendix, the *ANSI Reference Manual for the Ada Programming Language* is referred to as the LRM.

This appendix contains the following information:

ADE-Defined Pragmas

Pragmas tell the compiler how to process your program at compile time. They do not affect the semantics of a program, but they allow you to exercise some control over how the compiler processes your programs.

The *ANSI Reference Manual for the Ada Programming Language* (LRM) describes the standard pragmas and their use. Other pragmas are defined by the various implementations of the language. This section provides additional information on those standard pragmas and defines the pragmas that are unique to DGC Ada.

This section contains two parts. The first part lists pragmas that are not implemented in the current version of the ADE. The second part lists the implemented pragmas in alphabetical order, describes them, and provides examples of their use.

Pragmas That Have No Effect

The following Ada pragmas are not implemented in the current version of the ADE:

Pragma	Explanation
CONTROLLED	This pragma is not needed because the compiler does not reclaim unused storage automatically. To deallocate storage explicitly, use the generic procedure UNCHECKED_DEALLOCATION. Refer to the LRM, Section 13.10, and the *ADE User's Manual* for more information about this procedure.
MEMORY_SIZE	The package SYSTEM defines the MEMORY_SIZE constant as 2 ** 29 words. Use the /MTOP switch on the ADALINK command to adjust the maximum virtual memory size. Refer to the *ADE User's Manual* for more information about the ADALINK command.
OPTIMIZE	The compiler does not currently use time or space optimization criteria.
PACK	This pragma has no effect in the current version.
SHARED	The compiler does not implement indivisible direct read and update operations for any object; therefore, there are no objects to which you can apply this pragma. Refer to the LRM, Section 9.11.
STORAGE_UNIT	The package SYSTEM currently defines the storage unit as a 16-bit word. You can not redefine it.
SYSTEM_NAME	The package SYSTEM defines this as an object of enumeration type NAME, for which only one literal is allowed.

Pragmas Implemented in the ADE

The following pages describe, in alphabetical order, the ADE-specific pragmas.

---

pragma ELABORATE

Specifies which library unit bodies (secondary units) to elaborate before the current compilation unit.

Format

```
pragma ELABORATE (library_unit [,library_unit]);
```

where:

library_unit	Specifies the simple name of the library unit whose body you want elaborated before the current compilation unit.

Description

Pragma ELABORATE tells the compiler to elaborate the body of the specified library unit or units before elaborating the current compilation unit. If the current compilation unit is a subunit, the compiler elaborates the body of the specified library unit before elaborating the unit that is the ancestor of the current compilation subunit.

Pragma ELABORATE must appear after the context clause for the current compilation unit, and it must specify a library unit named in that context clause. The specified library unit must have a body.

For more information, refer to the LRM, Section 10.5.

Example

```
with EARTH_DATA;
pragma ELABORATE (EARTH_DATA);
procedure SOLAR_SYSTEM is
  ...
  EARTH_DATA.TRACK_ORBIT;
  ...
end SOLAR_SYSTEM;
```

pragma ENTRY_POINT

Associates an Ada subprogram name with a specific entry point label so foreign language routines can call or be called by Ada subprograms.

Format

```
pragma ENTRY_POINT (subprogram_name, "entry_point_name");
```

where:

subprogram_name	Specifies the unique name of an Ada subprogram defined in the declarative part of the current compilation unit. Do not use dot notation to specify subprogram_name.
entry_point_name	Specifies the STRING literal denoting the actual external label. Use uppercase letters enclosed in quotes, for example, "FRTN_LIBNAME".

Description

You can use this pragma in either of two ways:

- A subprogram written in another language can refer to an Ada subprogram using the entry point defined by this pragma.
- An Ada subprogram can call a library routine written in another language by giving the name of the routine as an entry point. In this case, you must also use pragma INTERFACE to specify the language of the library routine.

Pragma ENTRY_POINT must appear in the declarative part of a block, in a package specification, or after a compilation unit. You must specify both arguments.

Example

```
procedure MAIN is
  function FRTN_OP (X: INTEGER) return BOOLEAN;
  pragma INTERFACE (F77, FRTN_OP);
  pragma ENTRY_POINT (FRTN_OP, "FRTN_LIBNAME");
  ...
begin
...
end MAIN;
```

pragma INLINE

Specifies the subprograms and generic units that you want expanded inline at each call whenever possible.

Format

```
pragma INLINE (name [, name]);
```

where:

name	Specifies the subprogram or generic unit you want inlined at each call. The subprogram or generic unit must be defined before pragma INLINE in the declarative part of the program.

Description

Pragma INLINE tells the compiler to insert code for the body of the subprogram each time the subprogram is called. If the named subprogram is a generic unit, the compiler inserts code for the bodies of all subprograms that are instantiations of that generic unit.

The following restrictions apply to pragma INLINE:

- The nesting level of inlined procedures cannot exceed 100.
- A program that inlines a function that returns an unconstrained object will not work correctly.

The ADE will not inline the following:

- Recursive subprograms
- Subprograms containing exception handlers
- Any unit that declares a task, task type, or access to a task type.

pragma INLINE (continued)

Example

This example shows two assembly (.SR) files for the following source code. The first assembly file shows the source code compiled with pragma INLINE. The second example shows the assembly file without the pragma.

Source Code

In the following example, pragma INLINE applies to all the calls to SQUARE in WITH_INLINE.

```
procedure WITH_INLINE is
  FIRST, SECOND : INTEGER;
  function SQUARE (S : INTEGER) return INTEGER;
  pragma INLINE (SQUARE);

  function SQUARE (S : INTEGER) return INTEGER is
  begin
    return S * S;
  end SQUARE;
begin
  FIRST  := SQUARE (2);
  SECOND := SQUARE (SQUARE (FIRST));
end WITH_INLINE
```

pragma INLINE (continued)

Assembly File with Pragma INLINE

Each time SQUARE is called, the compiler inserts code for that function. In the following example, SQUARE is called three times. The last eight lines are the inlined subprogram.

```
;; begin
;; FIRST := SQUARE (2);
;; S : constant INTEGER := 2;
;; return S * S                  -- first inline expansion
   NLDAI                         4,0
   XWSTA                         0,12,,3              ;; FIRST

;; SECOND := SQUARE (SQUARE (FIRST));
;; S : constant INTEGER := SQUARE (FIRST);
;; S : constant INTEGER := FIRST;
;; return S * S                  -- second inline expansion
   XWMUL                         0,19,,3              ;; S
   XWSTA                         0,17,,3              ;; S
;; return S * S                  -- third inline expansion
   XWMUL                         0,17,,3              ;; S
   XWSTA                         0,14,,3              ;; SECOND
   WRTN
;; end

;; function SQUARE (S : INTEGER) return INTEGER is
;; begin
;; return S * S;
   XWLDA                         0,@-12,,3
   XWMUL                         0,@-12,,3
   XWSTA                         0,-8,,3
   WRTN
;; end
```

pragma INLINE (continued)

Assembly File Without Pragma INLINE

```
;; begin
;; FIRST := SQUARE (2);
   LPEF            L3              -- push effective address [L3] = 2
   LCALL           L2,1,1          -- first call to SQUARE
   XWSTA           0,12.,3
;; SECOND := SQUARE (SQUARE (FIRST));
   XWSTA           0,17.,3
   XPEF            17.,3           -- push effective address [17] = 4
   LCALL           L2,1,1          -- second call to SQUARE
   XWSTA           0,19.,3
   XPEF            19.,3           -- push effective address [19] = 16
   LCALL           L2,1,           -- third call to SQUARE
   XWSTA           0,14.,3
   WRTN
;; end

;; function SQUARE (S : INTEGER) return INTEGER is
;; begin
;; return S * S;
L2:                                -- beginning of the called function
   XWLDA           0,@-12.,3
   XWMUL           0,@-12.,3
   XWSTA           0,-8.,3
   WRTN
;; end                             -- end of the called function

L3:     2
   .END
```

pragma INTERFACE

Specifies another language (and calling conventions) for interfacing with an Ada program.

Format

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

where:

language_name	Specifies the language of the called subprogram.
subprogram_name	Specifies the name of the called subprogram. The subprogram must be declared earlier in the program.

Description

Pragma INTERFACE allows you to call program units written in other languages (foreign subprograms). A specification for the named subprogram must be written in Ada. The body of the subprogram can be written in another language.

Pragma INTERFACE must be in the declarative part or package specification of the Ada unit that calls the subprogram. The subprogram you specify as an argument must be declared earlier in the same declarative part or package specification.

Your program must include the following pragma LOAD statements in the order shown:

```
pragma LOAD ("ADE_ROOT?:RUNTIMES:INTERFACE_LRT_TRIGGGER");
pragma LOAD ("LANG_RT.LB");
```

You must be able to access LANG_RT.LB through one of the file access methods provided by the system, such as search lists or links. The IMPORT command links LANG_RT.LB automatically. Use it to import routines written in F77, C, or PASCAL.

Ada supports the calling of subprograms written in F77, PASCAL, C, MASM, and ASSEMBLY. In addition, you can call any language that obeys the common calling conventions of DGC languages, but you will receive a compiler warning that the language is not explicitly supported.

The Ada runtime interface traps any runtime errors in the called routine and raises the PROGRAM_ERROR exception in the calling program. The interface also suspends Ada tasking during the call to the non-Ada subroutine.

pragma INTERFACE (continued)

General Notes

- Characters within constructs are packed according to DGC alignment requirements for the called language.
- Booleans, arrays, and records are not packed. Booleans are passed one per word.
- Return values are not checked for validity.
- Procedure and function calls to other languages do not support type conversions. You must do type conversions explicitly.
- You can pass ACCESS types, but exercise caution when changing Ada data structures. Data General may change data formats in a future revision. After receiving any revisions of the ADE, test thoroughly all programs that depend on specific data formats.
- LANG_RT performs the exception handling for foreign subprograms. If a foreign subprogram has an error, that error is propagated to the calling Ada program as a PROGRAM_ERROR.
- Foreign subprograms must be in the same ring as the calling Ada program.
- Foreign subprograms can perform I/O operations, but it is the user's responsibility to use *pragma* LOAD to load all the necessary runtime objects. Alternately, you can use the template facility provided by ADALINK.
- The foreign code interface does not support Ada unconstrained types for any languages.
- All appropriate LB and OB files must be loaded into Ada programs that call foreign programs. The IMPORT function only ensures that the OB containing your function and LANG_RT are loaded with pragma LOAD. If the foreign code requires additional runtime support, such as MULTITASKING.OB, you should add the names of all necessary OB and LB files to *interface_package*_B file. This file is created by IMPORT or by ADALINK templates.

pragma INTERFACE (continued)

Foreign Language Calling Conventions and Data Types

The following sections describe the calling conventions and/or the data types used by DGC Ada to call subprograms written in foreign languages.

MASM or ASSEMBLY

The MASM and ASSEMBLY options provide the standard Ada calling conventions. If either is specified, the called program (which may or may not be MASM or ASSEMBLY) is expected to follow Ada calling conventions and to know how Ada data structures are formatted.

F77

F77 is supported as follows:

F77 Data Type	Ada Data Type
INTEGER*4	INTEGER
INTEGER*2	SHORT_INTEGER
REAL*4	FLOAT
REAL*8	LONG_FLOAT
CHARACTER*1	CHARACTER
CHARACTER*N	STRING(1..N)
ARRAY	ARRAY

Notes:

- Array elements must be of a supported scalar type.
- Scalar parameters are passed copy-in copy-out.
- One-dimensional arrays are passed by reference for copy-in copy-out.
- Multidimensional arrays obey copy-in copy-out rules.

pragma INTERFACE (continued)

C

C is supported as follows:

C Data Type	Ada Data Type
SHORT_INT	SHORT_INTEGER
LONG_INT	LONG_INTEGER
SHORT_FLOAT	FLOAT
LONG_FLOAT	LONG_FLOAT
CHARACTER	CHARACTER
POINTER	ACCESS
ENUMERATION	ENUMERATION
ARRAY OF CHARACTER	STRING
ARRAY	ARRAY
STRUCTURE	RECORD

Note: C calling conventions specify pass by value. Therefore, only copy-in mode is allowed for scalar parameters and structures. The call interface enforces pass by value for arrays.

PASCAL

PASCAL is supported as follows:

PASCAL Data Type	Ada Data Type
SHORT_INTEGER	SHORT_INTEGER
LONG_INTEGER	INTEGER
REAL	FLOAT
DOUBLE_REAL	LONG_FLOAT
BOOLEAN	BOOLEAN
CHAR	CHARACTER
ENUMERATION	ENUMERATION
POINTER	ACCESS
ARRAY	ARRAY
PACKED ARRAY OF CHAR	STRING
RECORD	RECORD

Notes:

- Not supported: RECORD VARIANTS, SET, FILE.
- One-dimensional arrays are passed by reference for copy-in copy-out.
- Multidimensional arrays obey copy-in copy-out rules.

pragma INTERFACE (continued)

PL/1

PL/1 is supported as follows:

PL/1 Data Type	Ada Data Type
FIXED BINARY (15)	SHORT_INTEGER
FIXED BINARY (31)	INTEGER
FLOAT BINARY (21)	FLOAT
FLOAT BINARY (53)	LONG_FLOAT
POINTER	ACCESS
ARRAY	ARRAY
RECORD	RECORD

Notes:

- PL/1 is not explicitly supported; however, the data types listed above can be used if all data follows standard LANG_RT alignment and space characteristics. Specifying PL/1 produces warning messages when you compile the program.
- One-dimensional arrays are passed by reference for copy-in copy-out.
- Multidimensional array obey copy-in copy-out rules.

pragma LIST

Suspends or resumes the compiler listing file output.

Format

```
pragma LIST (ON | OFF);
```

Description

The compiler always produces a listing (.LST) file unless you do one of the following:

- Include the /ERRORS switch with the ADA command (and the compilation units contain no errors)
- Include pragma LIST (OFF); in the compilation unit.

Pragma LIST (OFF); suspends the output in the .LST file during compilation.

Pragma LIST (ON); resumes .LST output.

Example

In the following example, the code for MEMBERS is not printed in the listing file.

```
procedure MAIN is
  type MEMBERS is private;
  procedure SORT (LIST: in out MEMBERS);
  function HEAD (L: LIST) return MEMBERS;
  ...
  pragma LIST (OFF);
  type MEMBERS is
  ...
  end MEMBERS;
  pragma LIST (ON);
begin
...
end MAIN;
```

pragma LOAD

Includes non-Ada object files in the linked program file.

Format

```
pragma LOAD ("object_file_pathname");
```

where:

object_file_pathname	Specifies the STRING literal (in quotes) that denotes the full pathname of the non-Ada object file you want to load. You do not need to include the .OB filename extension.

Description

Pragma LOAD allows you to include foreign (non-Ada) object files in your program. You can use it with pragmas INTERFACE and ENTRY_POINT to allow Ada procedures to call non-Ada subprograms. The Ada Linker includes the named object file when it builds the Ada program (.PR) file.

Pragma LOAD must appear at the head of a compilation for a body. When using pragma LOAD with compilation subunits, always specify the /READ_SUBUNITS switch on the ADALINK command line. If you omit that switch, you may receive this error message from the Linker:

"Can't get [body] tree for <program_unit_name>"

Note: Pragma LOAD does not guarantee the order of the loaded files. If order is important, use the /TEMPLATE switch with the ADALINK command.

Example

In the following example, the file SEVEN_UP.OB must be in the current directory.

```
pragma LOAD ("SEVEN_UP");
with TEXT_IO; use TEXT_IO;
procedure ADA_CALLS_PL1 is
  procedure SEVEN_UP (X: out INTEGER);
  pragma INTERFACE (PL1, SEVEN_UP);
  pragma ENTRY_POINT (SEVEN_UP, "SEVEN_UP");
  N : INTEGER;
begin
  SEVEN_UP (N);
  PUT (N);
end ADA_CALLS_PL1;
```

pragma MAIN

Indicates that a subprogram unit is a main program.

Format

```
pragma MAIN;
```

Description

Pragma MAIN designates the main subprogram unit. Place pragma MAIN immediately after the subprogram you want to be the main subprogram.

Example

The following code designates TEST as the main procedure.

```
procedure TEST is

  procedure FIRST is
  ...
  end FIRST;

  procedure SECOND is
  ...
  end SECOND;
begin
  ...
end TEST;
pragma MAIN;
```

pragma MAIN (continued)

Another way to distinguish the main subprogram in a compilation unit is to use the /MAIN_PROGRAM switch on the ADE command line. For example, you can compile the procedure TEST, located in the source file TEST.ADA, as a main program with this command:

```
-) ADA/MAIN_PROGRAM=TEST TEST
```

You must use the /MAIN_PROGRAM switch in each of the following cases:

- The source file that you are compiling contains more than one library unit
- You specify more than one source file with the same ADA command. The compiler assumes that the first file listed contains the main program. If it does not, you must specify which subprogram is the main program with the /MAIN_PROGRAM switch. For example, the following command compiles the source files FOO.ADA, FOOBAR.ADA, and TEST.ADA. It compiles the subprogram TEST.ADA as the main program:

  ```
  -) ADA/MAIN_PROGRAM=TEST FOO TEST FOOBAR
  ```

For more information about the ADA command, refer to the *ADE User's Manual*.

pragma MAX_TASKS

Specifies the maximum number of Ada tasks you want active simultaneously.

Format

```
pragma MAX_TASKS (n);
```

where:

n	Specifies an integer value greater than zero.

Description

Pragma MAX_TASKS specifies the maximum number of Ada tasks that can be active at the same time. If you do not specify the number, the system gives you a maximum of 50.

This pragma must appear at the head of a compilation. It applies to all units in the compilation.

Example

```
pragma MAX_TASKS(40);
package body TASKS is
  ...
  task ONE is ...;
  task TWO is ...;
  task type THREE_TO_FORTY is ...;
  type REMAINING_TASKS is
    array (3..40) of THREE_TO_FORTY;
  MULTI_TASKS : REMAINING_TASKS;
  ...
end TASKS;
```

You can also specify the maximum number of tasks by using the /MAX_TASKS switch with the ADALINK command. For example:

```
-) ADALINK/MAX_TASKS=40  object_filename
```

If you specify a maximum number of Ada tasks with both a pragma and a switch, the pragma takes precedence. For more information, refer to the *ADE User's Manual*.

pragma MV_ECS

Specifies the use of the Data General MV External Calling Sequence.

Format

```
pragma MV_ECS( unit_name [,unit_name...]);
```

where:

unit_name	Specifies the name of the subprogram for which you need the compiler to generate MV ECS.

Description

To optimize code quality, the compiler does not always generate code that conforms to the Data General MV External Calling Sequence (ECS). In some cases, however, you will need to tell the compiler that MV ECS is necessary. Subroutines that meet any of the following criteria must use MV ECS:

- MACHINE_CODE subroutines with formal arguments
- Subroutines called from other DGC languages
- Subroutines that can be called from outer rings.

Place pragma MV_ECS immediately after the subprogram for which you want the compiler to generate MV ECS.

Example

```
procedure TEST is

  procedure FIRST is
  ...
  end FIRST;

  procedure SECOND is
  ...
  end SECOND;
begin
  ...
end TEST;
pragma MV_ECS( TEST);
```

pragma PAGE

Begins a new page in the compiler output listing file.

Format

```
pragma PAGE;
```

Description

The compiler produces a listing (.LST) file unless you do one of the following:

- Include the /ERRORS switch with the ADA command (and the compilation unit contains no errors)
- Include pragma LIST (OFF); in the compilation unit.

If the compiler is producing a listing of the compilation, pragma PAGE causes the text following the pragma to appear on a new page.

Example

In the following example, procedure SECOND would be printed on a page by itself.

```
procedure FIRST is
...
end FIRST;

pragma PAGE;
procedure SECOND is
...
end SECOND;

pragma PAGE;
...
```

pragma PRIORITY

Specifies the priority of a task or task type.

Format

```
pragma PRIORITY (n);
```

where:

n	Specifies an integer value from 1 to 10. Lower values indicate lower priorities.

Description

You can assign priorities to tasks or task types by including pragma PRIORITY within the appropriate task specifications.

Assigning priorities tells the system how to handle competing tasks. When more that one task is eligible for execution at the same time, the system executes them in the order you specify with pragma PRIORITY. Tasks that are ready for execution are queued first by priority number and, within priorities, by order of their occurrence in the source file (FIFO).

You can assign each task or task type only one priority. If you assign more than one priority, the system recognizes the first assignment and ignores the others.

Assigning priorities is optional. The default priority is 5.

Example

The following code assigns a priority of 7 to TASK_TYPE and a priority of 8 to NEXT_TASK.

```
procedure OUTER is
  ...
  task type TASK_TYPE is
    pragma PRIORITY (7);
    ...
  end TASK_TYPE;
  ...
  task type NEXT_TASK is
    pragma PRIORITY (8);
    ...
  end NEXT_TASK;
...
end OUTER;
```

pragma SUPPRESS

Suppresses specified runtime checks.

Format

```
pragma SUPPRESS (check_identifier [, [ON =>] name]);
```

where:

check_identifier	Specifies the check you want to suppress. Check identifier names are listed in the description that follows.
name	Specifies the name of a type, subtype, object task unit, generic unit, or subprogram.

Description

To suppress certain runtime checks, place pragma SUPPRESS in the declarative part of a program unit or block or immediately within a package specification. For statements in a program unit or block, check suppression extends from the pragma statement to the end of the declarative part associated with that program unit or block. For statements in a package, check suppression extends to the end of the scope of the specified ON => entity. You must declare that entity immediately within the package specification.

The following table shows the extent of check suppression for each named entity.

Check suppression for	Extends over
An unnamed entity (name omitted)	The remaining declarative region
An object	All operations of the object
An object of the base type or subtype	All operations of the object or subtype
A task or task type	All activations of the task
A generic unit	All instantiations of the generic
A subprogram	All calls of the subprogram

pragma SUPPRESS (continued)

Although it is a better programming practice to have runtime exceptions raised automatically, you can suppress them if you need to decrease runtime overhead. When you suppress runtime checks, you turn off certain program exceptions. If an error arises after you have suppressed a check, your compiled program will not work correctly. The following table shows which program exceptions you turn off when you suppress checks:

Suppression of this check_identifier	Turns off this exception	When program detects this runtime error
ACCESS_CHECK	CONSTRAINT_ ERROR	Selection or indexing applied to an object with a null value
DISCIMINANT_ CHECK	CONSTRAINT_ ERROR	Violation of discriminant constraint
INDEX_CHECK	CONSTRAINT_ ERROR	Out-of-range index values
LENGTH_ CHECK	CONSTRAINT_ ERROR	Wrong number of index components
RANGE_CHECK	CONSTRAINT_ ERROR	Values exceed range constraint, or type is incompatible with constraint
DIVISION_ CHECK	NUMERIC_ ERROR	Division, rem, or mod by zero
OVERFLOW_ CHECK	NUMERIC_ ERROR	Operation result exceeds implemented range
ELABORATION_ CHECK	PROGRAM_ ERROR	Attempt to call a unit before it is elaborated
STORAGE_ CHECK	STORAGE_ ERROR	Over-allocation of memory space

pragma SUPPRESS (continued)

Example

In the following example, the pragma suppresses the checks on the indices of variables of the type TABLE. All type TABLE operations in MAIN are affected. No exceptions are raised if X and Y are not in the range of 1 to 8.

```
procedure MAIN is
  type COLOR is (RED, BLACK);
  type TABLE is array (1..8, 1..8) of COLOR;
  pragma SUPPRESS (INDEX_CHECK, ON => TABLE);
  X, Y : INTEGER;
  BOARD : TABLE;
begin
  ...
  BOARD (X, Y) := RED;
  ...
end;
```

pragma TASK_STORAGE_SIZE

Specifies the amount of heap storage space to allocate for task stacks.

Format

```
pragma TASK_STORAGE_SIZE (n);
```

where:

n	Specifies the total number of 2-byte words you want to allocate for all active task stacks. The variable n can be any integer value, but only values greater than -1 have an effect.

Description

Pragma TASK_STORAGE_SIZE allows you to reset the amount of heap space to allocate for all task stacks. The amount of space you specify should exceed the amount of storage you need at one time for all active tasks. By default, the system allocates 128 K words.

The pragma must appear at the head of a compilation. It applies to the entire compilation unit.

You can also use the /TASK_STORAGE_SIZE switch on the ADALINK command line to control the maximum heap space allocated to active task stacks. If you use both the pragma and the command switch, the pragma takes priority.

Resetting MTOP

If you need to set TASK_STORAGE_SIZE to a value greater than the current virtual address space allows, you must reset the maximum virtual address space by specifying the value of MTOP. MTOP defines the maximum virtual address for a program. Use the /MTOP switch with the ADALINK command to specify how many megabytes your program requires. The default value of MTOP is 1 Mbyte.

For example, this command resets MTOP to 20 Mbytes:

```
-) ADALINK/MTOP=20 object_file
```

pragma TASK_STORAGE_SIZE (continued)

Individual Task Storage

By default, the system allocates 2048 words for each active task stack. If you require a larger or smaller stack for a particular task type, use the STORAGE_SIZE representation clause. For example, the following clause tells the compiler to associate task type BIG with a stack of size N:

```
for BIG'STORAGE_SIZE use N;
```

The minimum stack size that you can specify is 512 words.

Example

In the following example, the value given in the pragma exceeds the storage required for all tasks executing at one time.

```
pragma TASK_STORAGE_SIZE(56_000)
procedure MAIN is
...
  task type ONE is ...;
  for ONE'STORAGE_SIZE use 1_000;

  task type TWO is ...;
  for TWO'STORAGE_SIZE use 2_000;
  ...

  task type TEN is ...;
  for TEN'STORAGE_SIZE use 10_000;

end MAIN;
```

Package SYSTEM

The predefined library package SYSTEM defines certain types, subtypes, and objects that are specific to DGC Ada. The package SYSTEM is described in the LRM, Section 13.7.

SYSTEM contains the following declarations:

```
package SYSTEM is

  type ADDRESS is new INTEGER;
  type NAME is (MV);
  SYSTEM_NAME              : constant : = NAME  : = MV;
  STORAGE_UNIT             : constant : = 16;
  MEMORY_SIZE              : constant : = 2 ** 29;

  MAX_INT                  : constant : = (2**30) - 1 + (2**30);
  MIN_INT                  : constant : = -MAX_INT - 1;
  MAX_DIGITS               : constant : = 15;
  MAX_MANTISSA             : constant : = 31;
  FINE_DELTA               : constant : = 2.0 ** (-31);
  TICK                     : constant : = 0.1;

  subtype PRIORITY is INTEGER range 1..10;

end SYSTEM;
```

The following table describes these types and constants and gives the value of each.

Type or Constant	Defined as	Explanation
ADDRESS	INTEGER	Address clauses and attributes (P'ADDRESS) return objects of the derived type ADDRESS.
NAME	MV	The enumeration type NAME declares one object: the literal MV.
SYSTEM_NAME	MV	SYSTEM_NAME is an object of type NAME and is initialized to MV.
STORAGE_UNIT	16	Denotes the number of bits per storage unit.
MEMORY_SIZE	2**29	Denotes the number of available storage units.
MAX_INT	(2**30)-1+(2**30) = 2_147_483_647	Denotes the highest value of predefined INTEGER types.

Type or Constant	Defined as	Explanation
MIN_INT	-MAX_INT - 1 = -2_147_483_648	Denotes the lowest (most negative) value of predefined INTEGER types.
MAX_DIGITS	15	Denotes the largest number of significant decimal digits in a floating-point constraint.
MAX_MANTISSA	31	Denotes the largest allowed number of binary digits in the mantissa of model numbers of a fixed-point subtype.
FINE_DELTA	2.0**(-31)	Denotes the smallest delta allowed in a fixed-point constraint that has the range constraint -1.0..1.0.
TICK	0.1	Denotes the basic clock period in seconds.
PRIORITY	1..10	Declares the range of values you can use on pragma PRIORITY statements. PRIORITY is a subtype of the base type INTEGER.

Representation Clauses

This section describes the use of representation clauses in the ADE. You can use representation clauses for either of two purposes:

- To specify a more efficient representation of data in the underlying machine
- To communicate with features outside the domain of the Ada language, for example, peripheral hardware.

The Ada programming language provides four classes of representation clauses:

Clause Class	Specifies
Length clause	The amount of storage you want associated with a type.
Enumeration representation	The internal codes for the literals of an enumeration type.
Record representation	The storage order, relative position, and size of record components.
Address clause	The required address in storage for an entity. Address clauses are not supported by the ADE. To assign internal names, use pragma ENTRY_POINT whenever possible.

The following paragraphs describe the use of each class of representation clauses.

Length Clauses

You can use the 'STORAGE_SIZE attribute only for reserving storage for activating a task type. For example:

```
BITS        :constant := 1;
BYTES       :constant := 8*BITS;
KBYTES      :constant := 1024*BYTES;

task type MONITOR is ...;

for MONITOR'STORAGE_SIZE use 4*KBYTES;
```

The ADE does not support the 'SIZE and 'SMALL attributes.

Enumeration Representations

The ADE supports enumeration representation clauses as specified in the LRM, Section 13.3. All enumeration literals must be provided with distinct, static integer codes. The sequence of integer codes specified for the enumeration type must consistently increase in value.

There are two restrictions:

- The range of internal codes must be a SHORT_INTEGER.
- Enumeration types with representation clauses are not allowed as the index type of an array type definition (refer to the LRM, Section 3.6).

Change of Representation

To change the representation clause of a type, you can declare a second type, derived from the first, and assign the variables of the first type to the second type. This process is described in the LRM, Section 13.6.

Operations of Discrete Types

If you use the attributes 'POS, 'VAL, 'SUCC, and 'PRED, executing the program may involve additional runtime overhead. Since potentially noncontiguous internal codes must be mapped to position numbers, executing the program involves additional overhead if the argument is nonstatic or is a discrete type or subtype whose base type is enumeration representation. Refer to the LRM, Section 13.3 for more information.

Conversions that Cause Overhead

Explicit conversions between enumeration types in which either base type has a representation clause may cause additional runtime overhead. The argument itself and the method of conversion both effect the amount of overhead.

You can perform explicit conversions between enumeration types by using an attribute such as 'POS or 'SUCC to evaluate an argument and assign the results to a variable of the target type. You can also perform explicit conversions by using the attribute and its argument as the actual parameter in a subprogram call. Each method of converting between types causes additional overhead if the argument is nonstatic. In the latter case, Ada performs checks on the actual parameter that may also add overhead.

Sections 3.5.5, 4.6, and 6.4.1 of the LRM provide more information about explicit conversions and parameter associations.

Case Statements

If the base type of the case statement expression is an enumeration type with a representation clause, the resulting code is optimized with respect to space rather than time. The value of the case statement expression is compared with case alternatives until a match is found.

Case statements with types other than enumeration with a representation clause are unaffected.

Loop Statements

FOR loops for which the base type of the loop parameter is an enumeration type with a representation clause causes additional runtime overhead. (For more information refer to the LRM, Section 3.5.5.).

Loop statements for which the base type is not an enumeration type with a representation clause do not cause additional overhead.

Record Representations

Representation of record types in the ADE is the same as in standard Ada with certain restrictions. Specifically, you cannot use record representation clauses to specify alignment and component locations for the following:

- Record types with discriminants
- Record types with variant parts
- Record types with array components.

When specifying component storage, you can cross only one 16-bit word boundary. You cannot specify the storage for composite, FLOAT, or LONG_FLOAT components. For components of these types, the compiler automatically determines the storage required. You can specify storage for all the remaining component types the same way as in standard Ada.

The following example shows a valid record representation specification:

```
type IUFL is
  record
    RETURN_FLAGS          : INTEGER range 0 .. 15;
    TERMINATION_FIELD     : INTEGER range 0 .. 7;
    PROCESS_ID            : INTEGER range 1 .. 255;
  end record;
for IUFL use
  record
    RETURN_FLAGS          at 0 range 0 .. 4;
    TERMINATION_FIELD     at 0 range 5 .. 7;
    PROCESS_ID            at 0 range 8 .. 15;
  end record;
```

These component clauses specify the order, position, and size of IUFL fields relative to the start of the IUFL record. They also ensure that the IUFL fields match the structure of the ?IUFL offset (user flag word) in a ?IREC system call:

Field Boundaries	Field Contents
0-4	RETURN_FLAGS
5-7	TERMINATION_FIELD
8-15	PROCESS_ID

The ADE does not allow components to overlap storage boundaries; that is, record fields cannot cross more than one 16-bit word boundary.

Unchecked Programming

The ADE implements the predefined, generic library subprograms UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION. The following sections explain how to use these subprograms.

Procedure UNCHECKED_DEALLOCATION

You can use the generic procedure UNCHECKED_DEALLOCATION to deallocate dynamic objects explicitly that are designated by values of access types. To deallocate dynamic objects explicitly, your program must instantiate this procedure for a particular object and access type. In the program body, a call to the instantiated procedure specifies the dynamic object as a parameter. When that call is executed, the specified object is deallocated, and its value is set to null. The following example shows how this works:

Example

In the following example, the call to the procedure DISPOSE deallocates the dynamic object designated by the access value ROOT1 and resets ROOT1 to null. However, if the enclosing procedure uses the other access value, ROOT2, to designate the same object as ROOT1, this code causes a program error because the object no longer exists. You must watch for similar dangling references when using the procedure UNCHECKED_DEALLOCATION.

```
with UNCHECKED_DEALLOCATION;
package TREE_LABELER is

  type LABEL_TYPE is private;
  type NODE;
  type TREE is access NODE;
  type NODE is record
    LABEL          : LABEL_TYPE;
    LEFT           : TREE;
    RIGHT          : TREE;
  end record;

  procedure DISPOSE is new UNCHECKED_DEALLOCATION (NODE, TREE);
  procedure LABEL_ROOT (LABEL          : in LABEL_TYPE;
                        ROOT           : in out TREE;
                        LABELLED_TREE  : out TREE);
end TREE_LABELER;

package body TREE_LABELER is
  procedure LABEL_ROOT (LABEL          : in LABEL_TYPE;
                        ROOT           : in out TREE;
                        LABELLED_TREE  : out TREE);
                        ROOT1, ROOT2   : NODE;

  begin
    ...
    DISPOSE (ROOT1);
    ...
  end LABEL_ROOT;
end TREE_LABELER;
```

Function UNCHECKED_CONVERSION

The generic function UNCHECKED_CONVERSION allows you to return the value of a copy-in parameter as a value of a target type. The actual bit pattern corresponding to that parameter value does not change.

The function UNCHECKED_CONVERSION is a unit in the ADE SYSTEM library. The visible part of that function is listed below:

```
generic
type SOURCE is limited private;
type TARGET is limited private;
function UNCHECKED_CONVERSION (S : SOURCE) return TARGET;

function UNCHECKED_CONVERSION (S : SOURCE) return TARGET is
  pragma SUPPRESS (RANGE_CHECK);
begin
  return S;
end UNCHECKED_CONVERSION;
```

For instantiations of this generic function, types SOURCE and TARGET must be of the same class and the same length. SOURCE and TARGET cannot be array types.

For more information about unchecked conversions, refer to the LRM, Section 13.10.

Example

The following example shows source code that uses the function UNCHECKED_CONVERSION.

```
with UNCHECKED_CONVERSION, ALPHA;
package BETA is
type TEST_NAME is private;
type DATA is record
    IS_VALID          : BOOLEAN;
    TEST_OBJECT       : TEST_NAME;
  end record;
  ...
function CONVERT_TO_BETA_DATA is new
  UNCHECKED_CONVERSION (ALPHA.INFO, DATA);
function CONVERT_FROM_BETA_DATA is new
  UNCHECKED_CONVERSION (DATA, ALPHA.INFO);
...
end BETA;
```

Characteristics of ADE Input/Output Packages

The standard input and output files in TEXT_IO correspond to the AOS/VS generic files @INPUT and @OUTPUT, respectively. For more information about AOS/VS generic files, refer to the DGC manual, *Learning to Use Your AOS/VS System*.

When you are using the ADE I/O packages, remember the following:

- The maximum value for TEXT_IO.COUNT and TEXT_IO.FIELD is SYSTEM.MAX_INT.
- The FORM parameter of the TEXT_IO.OPEN procedure is not used.
- Type TEXT_IO.FILE_TYPE is an access type.

For more information about input/output operations in the ADE, refer to the *ADE User's Manual*.

Maximum Size Limits in the ADE

The ADE places the following absolute limits on the use of Ada language features:

Compilation step	Language Feature	Maximum or amount
Syntax parsing	Length of identifiers	120
	Length of line	120
Semantics checking	Discriminants in constraint	256
	Associations in record aggregate	256
	Fields in record aggregate	256
	Formals in generic	256
	Nested contexts	250
Generating machine code	Indices in array aggregate	128
	Parameters in call	128
	Nesting depth of expressions	100
	Nesting depth of inlined expressions	100
	Nesting depth of packages with tasks	100

Summary of the ADE Real Type Attributes

The following section lists the name and value for each ADE specific real attribute.

Float Type	Value
T'MACHINE_RADIX	16
T'MACHINE_MANTISSA	6 for FLOAT 14 for LONG_FLOAT It is the number of T'MACHINE_RADIX (hex) digits in mantissa.
T'MACHINE_EMAX	63 It is the maximum exponent for MV floating types, base 16.
T'MACHINE_EMIN	-64 It is the minimum exponent for MV floating types, base 16.
T'MACHINE_ROUNDS	TRUE
T'MACHINE_OVERFLOWS	TRUE
T'SAFE_EMAX	252 The formula is: $\log_2$ (T'MACHINE_RADIX) * T'MACHINE_EMAX
T'SAFE_SMALL	2.0 ** (-T'SAFE_EMAX - 1)
T'SAFE_LARGE	2.0 ** T'SAFE_EMAX * (1.0 - 2.0 ** (-T'BASE'MANTISSA))

Fixed Types	Value
T'MACHINE_ROUNDS	TRUE
T'MACHINE_OVERFLOWS	TRUE
T'BASE'SMALL	= T'SMALL
T'BASE'MANTISSA	31 (Same as SYSTEM.MAX_MANTISSA)
T'SAFE_SMALL	= T'BASE'SMALL
T'SAFE_LARGE	= T'BASE'LARGE also = (2 ** T'BASE'MANTISSA - 1) * T'BASE'SMALL

General Notes

- All fixed-point numbers are stored in 32-bit integers.
- Floating-point types requiring 5 digits or less of precision are stored in FLOAT; those requiring 6 to 14 digits are stored in LONG_FLOAT.
- FLOAT and LONG_FLOAT use 1 bit for the sign and 7 bits for the exponent (of 16) in excess-64 notation. FLOAT has 24 bits available for the mantissa; LONG_FLOAT has 56.
- For FLOAT and LONG_FLOAT, the smallest number that can be represented in the MV architecture is given by the following formula:

  T'MACHINE_RADIX ** (T'MACHINE_EMIN - 1).

  This is equal to 16 ** (-65) or 16#0.10000000000000# * 16 ** (-64).

- For FLOAT and LONG_FLOAT, the largest number that can be represented in the MV architecture is given by the following formula:

  (1.0 - T'MACHINE_RADIX ** (-T'MACHINE_MANTISSA)) *
  (T'MACHINE_RADIX ** T'MACHINE_EMAX).

  For FLOAT, this is equal to 16#0.FFFFF# * 2 ** (63).

  For LONG_FLOAT, this is equal to the following:

  16#0.FFFFFFFFFFFFFF# * 2 ** (63) for LONG_FLOAT.

Type Definitions in the ADE

The ADE defines the types INTEGER, FLOAT, and DURATION as follows:

Type	Definition
INTEGER	The set of integers begins with the value MIN_INT and ends with MAX_INT. The formulas for MIN_INT and MAX_INT are described under "Package SYSTEM."
FLOAT	The type FLOAT is defined by the values described in the notes under "Summary of the ADE Real Type Attributes."
DURATION	The type DURATION is defined as follows: 2.0 ** (-9) range -2 ** 22... 2 **22;

End of Appendix

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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

-- $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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

-- $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_ID3 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA3AAAAAAAAAAA
```

```
-- $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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4AAAAAAAAAAA

-- $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     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

-- $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     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA1"

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

-- $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   8#37777777776#

-- $BIG_INT_LIT
-- AN INTEGER LITERAL WHOSE VALUE IS 298, BUT WHICH HAS
-- ($MAX_IN_LEN - 3) LEADING ZEROES.
-- USED IN:  C24003A
BIG_INT_LIT     000000000000000000000000000000000000000000000000000000000000000

-- $BIG_REAL_LIT
-- A UNIVERSAL_REAL LITERAL WHOSE VALUE IS 690.0, BUT WHICH HAS
-- ($MAX_IN_LEN - 5) LEADING ZEROES.
-- USED IN:  C24003B  C24003C
BIG_REAL_LIT    000000000000000000000000000000000000000000000000000000000000000

-- $MAX_LEN_INT_BASED_LITERAL
-- A BASED INTEGER LITERAL (USING COLONS) WHOSE VALUE IS 2:11:, HAVING
-- ($MAX_IN_LEN - 5) ZEROES BETWEEN THE FIRST COLON AND THE FIRST 1.
-- USED IN:  C2A009A
MAX_LEN_INT_BASED_LITERAL       2:000000000000000000000000000000000000000000000

-- $MAX_LEN_REAL_BASED_LITERAL
-- A BASED REAL LITERAL (USING COLONS) WHOSE VALUE IS 16:F.E:, HAVING
-- ($MAX_IN_LEN - 7) ZEROES BETWEEN THE FIRST COLON AND THE F.
-- USED IN:  C2A009A
MAX_LEN_REAL_BASED_LITERAL      16:00000000000000000000000000000000000000000000

-- $BLANKS
-- A SEQUENCE OF ($MAX_IN_LEN - 20) BLANKS.
-- USED IN:  B22001A  B22001B  B22001C  B22001D  B22001E  B22001F
```

B22001G B22001I B22001J B22001K B22001L B22001M
B22001N
< LIMITS OF SAMPLE SHOWN BY ANGLE BRACKETS >
KS <

MAX_DIGITS
N INTEGER LITERAL WHOSE VALUE IS SYSTEM.MAX_DIGITS.
SED IN: B35701A CD7102B
DIGITS 15

NAME
HE NAME OF A PREDEFINED INTEGER TYPE OTHER THAN INTEGER,
HORT_INTEGER, OR LONG_INTEGER.
IMPLEMENTATIONS WHICH HAVE NO SUCH TYPES SHOULD USE AN UNDEFINED
DENTIFIER SUCH AS NO_SUCH_TYPE_AVAILABLE.)
SED IN: AVAT007 C45231D B86001X C7D101G
NO_SUCH_TYPE_AVAILABLE

FLOAT_NAME
HE NAME OF A PREDEFINED FLOATING POINT TYPE OTHER THAN FLOAT,
HORT_FLOAT, OR LONG_FLOAT. (IMPLEMENTATIONS WHICH HAVE NO SUCH
YPES SHOULD USE AN UNDEFINED IDENTIFIER SUCH AS NO_SUCH_TYPE.)
SED IN: AVAT013 B86001Z
T_NAME NO_SUCH_TYPE

FIXED_NAME
HE NAME OF A PREDEFINED FIXED POINT TYPE OTHER THAN DURATION.
IMPLEMENTATIONS WHICH HAVE NO SUCH TYPES SHOULD USE AN UNDEFINED
DENTIFIER SUCH AS NO_SUCH_TYPE.)
SED IN: AVAT015 B86001Y
D_NAME NO_SUCH_FIXED_TYPE

INTEGER_FIRST
N INTEGER LITERAL, WITH SIGN, WHOSE VALUE IS INTEGER'FIRST.
HE LITERAL MUST NOT INCLUDE UNDERSCORES OR LEADING OR TRAILING
LANKS.
SED IN: C35503F B54B01B
GER_FIRST -2147483648

INTEGER_LAST
N INTEGER LITERAL WHOSE VALUE IS INTEGER'LAST. THE LITERAL MUST
OT INCLUDE UNDERSCORES OR LEADING OR TRAILING BLANKS.
SED IN: C35503F C45232A B45B01B
GER_LAST 2147483647

INTEGER_LAST_PLUS_1
N INTEGER LITERAL WHOSE VALUE IS INTEGER'LAST + 1.
SED IN: C45232A
GER_LAST_PLUS_1 2147483648

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
_INT -2147483648

MAX_INT
AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MAX_INT.
THE LITERAL MUST NOT INCLUDE UNDERSCORES OR LEADING OR TRAILING
BLANKS.
USED IN: C35503D C35503F C4A007A CD7101B
_INT 2147483647

```
-- $TASK_SIZE
-- AN INTEGER LITERAL WHOSE VALUE IS THE NUMBER OF BITS REQUIRED TO
-- HOLD A TASK OBJECT WHICH HAS A SINGLE ENTRY WITH ONE INOUT PARAMETER.
-- 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:  CD2A95A
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       MV

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

-- $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    MV

-- $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        536_870_912

-- $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    536_870_912
```

```
-- $LOW_PRIORITY
-- AN INTEGER LITERAL WHOSE VALUE IS THE LOWER BOUND OF THE RANGE
-- FOR THE SUBTYPE SYSTEM.PRIORITY.
-- USED IN:  CD7007C
LOW_PRIORITY    1

-- $HIGH_PRIORITY
-- AN INTEGER LITERAL WHOSE VALUE IS THE UPPER BOUND OF THE RANGE
-- FOR THE SUBTYPE SYSTEM.PRIORITY.
-- USED IN:  CD7007C
HIGH_PRIORITY   10

-- $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**(-31)

-- $TICK
-- A REAL LITERAL WHOSE VALUE IS SYSTEM.TICK AS SPECIFIED IN THE
-- IMPLEMENTOR'S DOCUMENTATION.
-- USED IN:  CD7104B
TICK    0.1
```

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 implememtation 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, CD2A84N & M, & 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

Loral/Rolm Mil-Spec Computers

Compiler: ADE Revision 3.01

ACVC Version: 1.10

The following compiler switches are available for DG ADE 3.01:

/ASSEMBLY Preserves the assembly language for an Ada program in a .SR file. If this switch is not given, the assembly-language source may be deleted after the compilation; this option is controlled by the ADE configuration. (For details on ADE configuration, see the ADE release notice.) When the user supplies this switch, the Ada source code will appear as comments in the .SR file. Use this switch for machine-level debugging only.

/AUTO_INLINING=n Tells the compiler to inline any subroutine called n or fewer times. For the compiler to perform automatic inlining on a subroutine, the subroutine must not be visible outside its compilation unit, and must also pass some implementation restrictions which ensure the code will be duplicated no more than n times. Automatic inlining will not occur when the /NO_INLINING switch is present. Do not use /AUTO_INLINING on a source which contains MACHINE_CODE subroutines which manipulate parameters, because parameters are not passed on the stack to an inlined subroutine. When /AUTO_INLINING=0, the compiler will not generate code for unreferenced subroutines which pass the automatic inlining implementation restrictions.

/CONFIGURATION=configname

Generate code for the configuration whose source text statements begin: "--/configname". You may give multiple confignames by separating them with underscores (for example: /CONFIGURATION=config1_config2_config3).

/CPL=n Controls listing columns-per-line. The value of n may be from 40 to 200, and includes eight columns per line used by the compiler. Lines that are longer than n columns are split so that indentation is preserved when possible.

/DEBUG Compiles filename for use with the Ada Source Code Debugger. (The Ada Debugger is sold separately with the ADEX product and may not be available at your site.) NOTE: Compiling with the /DEBUG switch will increase the volume of generated code and decrease runtime performance.

/ERRORS Inhibits a full listing. Puts only error messages (if any) in the .LST. If there are no errors, the listing file will be empty.

/IDIR=dirname Specifies the directory where otherwise unqualified input filenames may be obtained. When input pathnames include a directory prefix, the IDIR= switch is ignored.

/LIBRARY=libname Names the target Ada library into which the source is to be compiled. If omitted, ADE uses the current directory's default library. All binaries output by the compiler are placed in the same directory as the

one in which the target library reside.

/LPP=n — Controls listing lines-per-page, where n is an integer in the range 0..66. A value of 0 disables page ejects and headings. Default n is 66.

/MAIN_PROGRAM[=name] — Specifies the source is a main program. If the source file contains more than one library unit, the /MAIN_PROGRAM=name keyword switch must be used.

/NO_SYSTEM — Prevents automatic inclusion of Ada system library in the library searchlist for this compile.

/NO_INLINING — Overrides /AUTO_INLINING and pragma INLINE. Since the Ada Source Code Debugger cannot debug inlined subprograms, use of this switch will help in using the Debugger.

/SUPPRESS — Suppresses all run-time checking in the code output by the compiler, including range checking and record variant checking. This makes your compiled program run faster, but also makes debugging more difficult.

/TABLE — Generates information needed by the Ada Source Code Debugger to view information, but not set breakpoints nor step. You need not include this switch if the /DEBUG switch is specified. NOTE: This switch increases the generated code size and decreases runtime performance, but not as much as the /DEBUG switch.