

AD-A223 300

REPORT DOCUMENTATION PAGE		Form Approved OPM No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.</small>			
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED Final 21 Sept 89 to 21 Sept 90	
4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: Tartan Laboratories Incorporated Tartan Ada VMS Version V11.008024031 VAX 8350 (Host & Target), 890928I1.10264		5. FUNDING NUMBERS	
6. AUTHOR(S) IABG-AVF Ottobrunn, FEDERAL REPUBLIC OF GERMANY			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) IABG-AVF, Industrieanlagen-Betriebsgesellschaft Dept. SZT Einsteinstrasse 20 D-8012 Ottobrunn FEDERAL REPUBLIC OF GERMANY		8. PERFORMING ORGANIZATION REPORT NUMBER AVF-IABG-028	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Washington, D.C. 20301-3081		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.		12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Tartan Laboratories Incorporated, Tart Ada VMS Version V11.008024031, VAX 8350 (Host & Target), West Germany, ACVC 1.10. <p style="text-align: right;">SEP 21 1990 66</p>			
14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office		15. NUMBER OF PAGES	16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT

Ada Compiler Validation Summary Report:

Compiler Name: TARTAN Ada VMS. Version V11.008024031

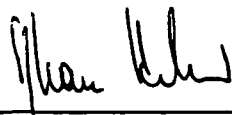
Certificate Number: #890928I1.10264

Host: VAX 8350 under VMS 4.7

Target: VAX 8350 under VMS 4.7

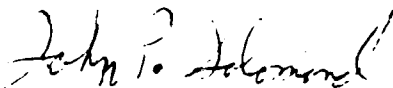
Testing completed 28 September 1989 using ACVC 1.10.

This report has been reviewed and is approved.



Dr. S. Heilbrunner
IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn
West Germany

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

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT	1
1.2	USE OF THIS VALIDATION SUMMARY REPORT	2
1.3	REFERENCES	3
1.4	DEFINITION OF TERMS	3
1.5	ACVC TEST CLASSES	4
CHAPTER 2	CONFIGURATION INFORMATION	7
2.1	CONFIGURATION TESTED	7
2.2	IMPLEMENTATION CHARACTERISTICS	8
CHAPTER 3	TEST INFORMATION	14
3.1	TEST RESULTS	14
3.2	SUMMARY OF TEST RESULTS BY CLASS	14
3.3	SUMMARY OF TEST RESULTS BY CHAPTER	15
3.4	WITHDRAWN TESTS	15
3.5	INAPPLICABLE TESTS	15
3.6	TEST, PROCESSING, AND EVALUATION MODIFICATIONS	19
3.7	ADDITIONAL TESTING INFORMATION	
3.7.1	Prevalidation	20
3.7.2	Test Method	20
3.7.3	Test Site	21
APPENDIX A	DECLARATION OF CONFORMANCE	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	
APPENDIX D	COMPILER AND LINKER OPTIONS	

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 IABG mbH, Abt SZT according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). Testing was completed 28 September 1989 on the premises of IABG.

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:

IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn

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

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

1.3 REFERENCES

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

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer 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 test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

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

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

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

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

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

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

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

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

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and 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: TARTAN Ada VMS, Version V11.008024031

ACVC Version: 1.10

Certificate Number: #890928I1.10264

Host and Target Computer:

Machine: VAX 8350

Operating System: VMS 4.7

Memory Size: 12 MB

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`, `BYTE_INTEGER`, and `LONG_FLOAT` in the package `STANDARD`. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

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

- 1) 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 for predefined and largest integer and no exception is raised for smallest integer 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).)

d. Rounding.

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

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

e. Array types.

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

- 1) Declaration of an array type or subtype declaration with more than `SYSTEM.MAX_INT` components raises `NUMERIC_ERROR` for one dimensional array types, two dimensional array types and two dimensional array subtypes, and no exception for one dimensional array subtypes. (See test C36003A.)
- 2) `NUMERIC_ERROR` is raised when an array type with `INTEGER'LAST + 2` components is declared. (See test C36202A.)
- 3) `NUMERIC_ERROR` is raised when an array type with `SYSTEM.MAX_INT + 2` components is declared. (See test C36202B.)
- 4) A packed `BOOLEAN` array having a `'LENGTH` exceeding `INTEGER'LAST` raises `NUMERIC_ERROR` when the array type is declared. (See test C52103X.)

- 5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared and exceeds INTEGER'LAST. (See test C52104Y.)
- 6) In assigning one-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.)
- 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.)
- 8) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

f. Discriminated types.

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

g. Aggregates.

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

h. Pragmas.

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

i. Generics.

This compiler enforces the following two rules concerning declarations and proper bodies which are individual compilation units:

- o generic bodies must be compiled and completed before their instantiation.
- o recompilation of a generic body or any of its transitive subunits makes all units obsolete which instantiate that generic body.

These rules are enforced whether the compilation units are in separate compilation files or not. AI408 and AI506 allow this behaviour.

- 1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)
- 2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)
- 3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)
- 4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)
- 5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)
- 6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)
- 7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)
- 8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

- 9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)
- j. Input and output.
- 1) The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)
 - 2) The package DIRECT_IO cannot be instantiated with unconstrained array types 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 not given names. (See test CE2108A.)
 - 11) Temporary direct files are given names and deleted when closed. (See test CE2108C.)
 - 12) Temporary text files are not given names. (See test CE3112A.)
 - 13) More than one internal file can be associated with each permanent (not temporary) external file for sequential files when reading only. (See tests CE2107A..D)

(4 tests), CE2102L, CE2110B, and CE2111D.)

- 14) More than one internal file can be associated with each permanent (not temporary) external file for direct files when reading only. (See tests CE2107F..H (3 tests), CE2110D and CE2111H.)
- 15) More than one internal file can be associated with each permanent (not temporary) external file for text files when reading only. (See tests CE3111A..B (2 tests), CE3111D..E (2 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 465 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 285 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 79 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	127	1132	1862	17	24	46	3208
Inapplicable	2	6	453	0	4	0	465
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	192	547	497	245	172	99	162	332	133	36	252	258	283	3208	
N/A	20	102	183	3	0	0	4	0	4	0	0	111	38	465	
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44	
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717	

3.4 WITHDRAWN TESTS

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

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

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

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 465 tests were inapplicable for the reasons indicated:

- a. The following 285 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)

C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

- b. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.
- c. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

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

- d. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 48.
- e. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.
- f. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT.
- g. CA2009A, CA2009C, CA2009F and CA2009D are not applicable because this compiler creates dependencies between generic bodies, and units that instantiate them (see section 2.2i for rules and restrictions concerning generics).
- h. CD1009C, CD2A41A..E (5 tests), and CD2A42A..J (10 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for floating point types.
- i. CD2A61I is not applicable because this implementation imposes restrictions on 'SIZE length clauses for array types.
- j. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for access types.
- k. CD2A91A..E (5 tests) are not applicable because 'SIZE length clauses for task types are not supported.
- l. CD2B11G is not applicable because 'STORAGE_SIZE representation clauses are not supported for access types where the designated type is a task type.
- m. CD2B15B is not applicable because a collection size larger than

the size specified was allocated.

- n. The following 76 tests are inapplicable because address clauses are not supported under the specific conditions of these tests (see also section 5.4.5 of Appendix F contained in Appendix B of this Validation Summary Report):

CD5003B..I (8 tests),	CD5011A..I (9 tests)
CD5011K..N (4 tests),	CD5011Q..S (3 tests)
CD5012A..J (10 tests),	CD5012L..M (2 tests)
CD5013A..I (9 tests),	CD5013K..O (5 tests)
CD5013R..S (2 tests),	CD5014A..O (15 tests)
CD5014R..Z (9 tests).	

- o. 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.
- p. 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.
- q. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.
- r. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.
- s. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.
- t. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.
- u. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.
- v. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.
- w. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.
- x. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

- y. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.
- z. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.
- aa. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.
- ab. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.
- ac. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.
- ad. CE2105A is inapplicable because CREATE with IN_FILE mode is not supported by this implementation for SEQUENTIAL_IO.
- ae. CE2105B is inapplicable because CREATE with IN_FILE mode is not supported by this implementation for DIRECT_IO.
- af. CE2108B and CE2108D are inapplicable because temporary files do not have names for SEQUENTIAL_IO or DIRECT_IO.
- ag. CE3102E is inapplicable because text file CREATE with IN_FILE mode is supported by this implementation.
- ah. CE3102F is inapplicable because text file RESET is supported by this implementation.
- ai. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.
- aj. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.
- ak. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.
- al. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is not supported by this implementation.
- am. CE2107B..D (3 tests), CE2107L, CE2107G..H (2 tests), CE3111B, and CE3111D..E (2 tests) are not applicable because multiple internal files cannot be associated with the same external file for the operations attempted in these tests.

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 79 tests.

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

B22003A	B24007A	B24009A	B25002B	B32201A	B34005N
B34005T	B34007H	B35701A	B36171A	B36201A	B37101A
B37102A	B37201A	B37202A	B37203A	B37302A	B38003A
B38003B	B38008A	B38008B	B38009A	B38009B	B38103A
B38103B	B38103C	B38103D	B38103E	B43202C	B44002A
B48002A	B48002B	B48002D	B48002E	B48002G	B48003E
B49003A	B49005A	B49006A	B49007A	B49009A	B4A010C
B54A20A	B54A25A	B58002A	B58002B	B59001A	B59001C
B59001I	B62006C	B67001A	B67001B	B67001C	B67001D
B74103E	B74104A	B85007C	B91005A	B95003A	B95007B
B95031A	B95074E	BC1002A	BC1109A	BC1109C	BC1202E
BC1206A	BC2001E	BC3005B	BC3009C	BD5005B	

- b. For the two tests BC3204C and BC3205D, the compilation order was changed to

BC3204C0, C1, C2, C3M, C4, C5, C6, C3M
and
BC3205D0, D2, D1M

respectively. This change was necessary because of the compiler's rules for separately compiled generic units (see section 2.2i for rules and restrictions concerning generics). When processed in this order the expected error messages were produced for BC3204C3M and BC3205D1M.

- c. The two tests BC3204D and BC3205C consist of several compilation units each. The compilation units for the main procedures are near the beginning of the files. When processing these files unchanged, a link error is reported instead of the expected compiled generic units. Therefore, the compilation files were modified by appending copies of the main procedures to the end of

CONFIGURATION INFORMATION

these files. When processed, the expected error messages were generated by the compiler.

- d. Tests C39005A, CD7004C, CD7005E and CD7006E wrongly presume an order of elaboration of the library unit bodies. These tests were modified to include a PRAGMA ELABORATE (REPORT);
- e. Test E28002B checks that predefined or unrecognized pragmas may have arguments involving overloaded identifiers without enough contextual information to resolve the overloading. It also checks the correct processing of pragma LIST. For this implementation, pragma LIST is only recognised if the compilation file is compiled without errors or warnings. Hence, the test was processed in two versions: the original version, and a version where the pragmas OPTIMIZE and PHIL_BRASHEAR were removed. The modified version demonstrated the correct processing of pragma LIST.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

No prevalidation results were submitted by the customer, because testing was done by IABG on the premises of IABG and the results were nearly identical to those obtained for a previous validation for the same customer.

3.7.2 Test Method

Testing of the TARTAN Ada VMS, Version V11.008024031 compiler using ACVC Version 1.10 was conducted by IABG on the premises of IABG. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host and Target Computer:

Machine:	VAX 8350
Operating System:	VMS 4.7
Memory Size:	12 MB

Compiler:

TARTAN Ada VMS, Version V11.008024031

The original ACVC was loaded to a VAX 8350, where it was customized to remove all withdrawn tests and tests requiring unsupported floating point precisions. Tests that make use of implementation specific values were

also customized.

After the tests were loaded to disk, the full set of tests was compiled, linked and all executable tests were run on the VAX 8350.

The compiler was tested using command scripts provided by TARTAN LABORATORIES INCORPORATED and reviewed by the validation team. The compiler was tested using only default option settings. All chapter B tests were compiled with the listing option on (i.e. /LIST). The linker was called with the command

```
alib link <testname>
```

A full list of compiler and linker options is given in Appendix E.

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on disk and archived at the AVF.

3.7.3 Test Site

Testing was conducted at IABG mbH, Ottobrunn and was completed on 28 September 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

TARTAN LABORATORIES INCORPORATED has submitted the following Declaration of Conformance concerning the Tartan Ada VMS, Version V11.008024031 compiler.



TARTAN

DECLARATION OF CONFORMANCE

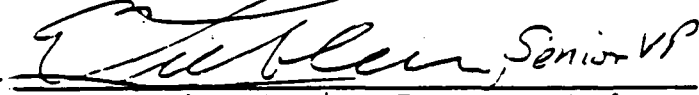
Compiler Implementor: Tartan Laboratories Incorporated
Ada Validation Facility: IABG mbH, Dept. SZT
Ada Compiler Validation Capability (ACVC) Version: 1.10

Ada Implementation

Base Compiler Name: Tartan Ada VMS
Version: V11.008024031
Host Computer System: VAX 8350 under VMS 4.7
Target Computer System: VAX 8350 under VMS 4.7

Implementor's Declaration

I, the undersigned, representing Tartan Laboratories Incorporated, declare that Tartan Laboratories Incorporated has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Tartan Laboratories Incorporated 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.



Tartan Laboratories Incorporated
D. L. Evans, President

Date: Oct. 2, 1989

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 Tartan Ada VMS, Version V11.008024031 compiler as described in this Appendix, are provided by TARTAN LABORATORIES INCORPORATED. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD are also a part of this Appendix.

Chapter 5

Appendix F to MIL-STD-1815A

This chapter contains the required Appendix F to *Military Standard, Ada Programming Language*, ANSI/MIL-STD-1815A (American National Standards Institute, Inc., February 17, 1983).

5.1. PRAGMAS

5.1.1. Predefined Pragma

This section summarizes the effects of and restrictions on predefined pragmas.

- Access collections are not subject to automatic storage reclamation so pragma CONTROLLED has no effect. Space deallocated by means of UNCHECKED_DEALLOCATION will be reused by the allocation of new objects.
- Pragma ELABORATE is supported.
- Pragma INLINE is supported. The body for an inlined subprogram need not appear in the same compilation unit as the call. Inlining will take place only if the subprogram body is present in the library and is not obsolete.
- Pragma INTERFACE is not supported. The implementation-defined pragma FOREIGN_BODY (see Section 5.1.2.2) can be used to interface to subprograms written in other languages.
- Pragma LIST is supported but has the intended effect only if the command qualifier LIST=ALWAYS was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma MEMORY_SIZE is accepted but no value other than that specified in Package SYSTEM (Section 5.3) is allowed.
- Pragma OPTIMIZE is supported except when at the outer level (that is, in a package specification or body).
- Pragma PACK is fully supported.
- Pragma PAGE is supported but has the intended effect only if the command qualifier LIST=ALWAYS was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma PRIORITY is supported.
- Pragma STORAGE_UNIT is accepted but no value other than that specified in Package SYSTEM (Section 5.3) is allowed.
- SHARED is not supported. No warning is issued if it is supplied.
- Pragma SUPPRESS is supported.
- Pragma SYSTEM_NAME is accepted but no value other than that specified in Package SYSTEM (Section 5.3) is allowed.

C

5.1.2. Implementation-Defined Pragmas

Implementation-defined pragmas provided by Tartan are described in the following sections.

5.1.2.1. Pragma LINKAGE_NAME

The pragma `LINKAGE_NAME` associates an Ada entity with a string that is meaningful externally; e.g., to a linkage editor. It takes the form

```
pragma LINKAGE_NAME (Ada-simple-name, string-constant)
```

The *Ada-simple-name* must be the name of an Ada entity declared in a package specification. This entity must be one that has a runtime representation; e.g., a subprogram, exception or object. It may not be a named number or string constant. The pragma must appear after the declaration of the entity in the same package specification.

The effect of the pragma is to cause the *string-constant* to be used in the generated assembly code as an external name for the associated Ada entity. It is the responsibility of the user to guarantee that this string constant is meaningful to the linkage editor and that no illegal linkname clashes arise.

5.1.2.2. Pragma FOREIGN_BODY

A subprogram written in another language can be called from an Ada program. Pragma `FOREIGN_BODY` is used to indicate that the body for a non-generic top-level package specification is provided in the form of an object module. The bodies for several subprograms may be contained in one object module.

Use of the pragma `FOREIGN_BODY` dictates that all subprograms, exceptions and objects in the package are provided by means of a foreign object module. In order to successfully link a program including a foreign body, the object module for that body must be provided to the library using the `ALIB FOREIGN_BODY` command described in Section 4.7.

The pragma is of the form:

```
pragma FOREIGN_BODY (language_name [, elaboration_routine_name])
```

The parameter *language_name* is a string intended to allow the compiler to identify the calling convention used by the foreign module (but this functionality is not yet in operation). Currently, the programmer must ensure that the calling convention and data representation of the foreign body procedures are compatible with those used by the Tartan Ada compiler. Subprograms called by tasks should be reentrant.

The optional *elaboration_routine_name* string argument provides a means to initialize the package. The routine specified as the *elaboration_routine_name*, which will be called for the elaboration of this package body, must be a global routine in the object module provided by the user.

A specification that uses this pragma may contain only subprogram declarations, object declarations that use an unconstrained type mark, and number declarations. Pragmas may also appear in the package. The type mark for an object cannot be a task type, and the object declaration must not have an initial value expression. The pragma must be given prior to any declarations within the package specification. If the pragma is not located before the first declaration, or any restriction on the declarations is violated, the pragma is ignored and a warning is generated.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma `FOREIGN_BODY`. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

Pragma `LINKAGE_NAME` should be used for all declarations in the package, including any declarations in a nested package specification to be sure that there are no conflicting link names. If pragma `LINKAGE_NAME` is not used, the cross-reference qualifier, `/CROSS_REFERENCE`, (see Section 3.2) should be used when invoking the compiler and the resulting cross-reference table of linknames inspected to identify the linknames assigned by the compiler and determine that there are no conflicting linknames (see also Section 3.5).

In the following example, we want to call a function `p1mn` which computes polynomials and is written in C.

```

package MATH_FUNCS is
  pragma FOREIGN_BODY ("C");
  function POLYNOMIAL (X:INTEGER) return INTEGER;
    --Ada spec matching the C routine
  pragma LINKAGE_NAME (POLYNOMIAL, "plmn");
    --Force compiler to use name "plmn" when referring to this
  -- function
end MATH_FUNCS;

with MATH_FUNCS; use MATH_FUNCS
procedure MAIN is
  X:INTEGER := POLYNOMIAL(10);
    -- Will generate a call to "plmn"
  begin ...
end MAIN;

```

To compile, link and run the above program, you do the following steps:

1. Compile MATH_FUNCS
2. Compile MAIN
3. Obtain an object module (e.g. math.OBJ) containing the compiled code for plmn.
4. Issue the command

```
ALIB FOREIGN_BODY math_funcs MATH.OBJ
```

5. Issue the command

```
ALIB LINK MAIN
```

Without Step 4, an attempt to link will produce an error message informing you of a missing package body for MATH_FUNCS.

Using an Ada body from another Ada program library. The user may compile a body written in Ada for a specification into the library, regardless of the language specified in the pragma contained in the specification. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a foreign body with an Ada body without recompiling the specification.

The user can either compile an Ada body into the library, or use the command ALIB FOREIGN_BODY (See Section 4.7) to use an Ada body from another library. The Ada body from another library must have been compiled under an identical specification. The pragma LINKAGE_NAME must have been applied to all entities declared in the specification. The only way to specify the linkname for the elaboration routine of an Ada body is with the pragma FOREIGN_BODY.

Using Calls to the Operating System. In some cases, the foreign code is actually supplied by the operating system (in the case of system calls) or by runtime libraries for other programming languages such as C. Such calls may be made using a dummy procedure to supply a file specification to the ALIB FOREIGN_BODY command. You need a dummy .OBJ file which may be obtained in a number of ways. One way is to compile the procedure

```

procedure DUMMY is
begin
  null;
end;

```

Then, use the library command

```
ALIB FOREIGN pkg DUMMY.OBJ
```

where *pkg* is the name of the package that contains the pragma LINKAGE_NAME for the operating system call.

For example to use the VMS system service LIBSGET_VM in the program TEST:

```

Package MEMORY is
  pragma FOREIGN_BODY ("ASM");
  procedure GET_VIRTUAL_MEMORY (MEM: INTEGER);
  pragma LINKAGE_NAME (GET_VIRTUAL_MEMORY, "LIBSGET_VM ");
end MEMORY;

with MEMORY;
procedure TEST is
  ...

begin
  GET_VIRTUAL_MEMORY (MEM);
  ...
end TEST;

```

Obtain the file dummy .OBJ. Then use
 ALIB FOREIGN pkg DUMMY.OBJ
 to include the body for the system call in the library.

5.2. IMPLEMENTATION-DEPENDENT ATTRIBUTES

No implementation-dependent attributes are currently supported.

5.3. SPECIFICATION OF THE PACKAGE SYSTEM

The parameter values specified for the VAX in package SYSTEM [LRM 13.7.1 and Annex C] are:

```

package SYSTEM is
  type ADDRESS is new INTEGER;
  type NAME is (VAX);
  SYSTEM_NAME : constant := VAX;
  STORAGE_UNIT : constant := 8;
  MEMORY_SIZE : constant := 1000000;
  MAX_INT : constant := 2147483647;
  MIN_INT : constant := -MAX_INT - 1;
  MAX_DIGITS : constant := 9;
  MAX_MANTISSA : constant := 31;
  FINE_DELTA : constant := 2#1.0#e-31;
  TICK : constant := 0.01667;
  subtype PRIORITY is INTEGER range 10 .. 200;
  DEFAULT_PRIORITY : constant PRIORITY := PRIORITY'FIRST;
  RUNTIME_ERROR : exception;
end SYSTEM;

```

5.4. RESTRICTIONS ON REPRESENTATION CLAUSES

The following sections explain the basic restrictions for representation specifications followed by additional restrictions applying to specific kinds of clauses.

5.4.1. Basic Restriction

The basic restriction on representation specifications [LRM 13.1] that they may be given only for types declared in terms of a type definition, excluding a *generic_type_definition* (LRM 12.1) and a *private_type_definition* (LRM 7.4). Any representation clause in violation of these rules is not obeyed by the compiler; a diagnostic message is issued.

Further restrictions are explained in the following sections. Any representation clauses violating those restrictions are not obeyed but cause a diagnostic message to be issued.

5.4.2. Length Clauses

Length clauses [LRM 13.2] are, in general, supported. For details, refer to the following sections.

5.4.2.1. Size Specifications for Types

The rules and restrictions for size specifications applied to types of various classes are described below. The following principle rules apply:

1. The size is specified in bits and must be given by a static expression.
2. The specified size is taken as a mandate to store objects of the type in the given size wherever feasible. No attempt is made to store values of the type in a smaller size, even if possible. The following rules apply with regard to feasibility:

- An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine; that is, no attempt is made to create objects of non-referable size on the stack. If such stack compression is desired, it can be achieved by the user by combining multiple stack variables in a composite object; for example

```

type My_Enum is (A,B);
for My_Enum'size use 1;
V,W: My_Enum; -- will occupy two storage
               -- units on the stack
               -- (if allocated at all)

type rec is record
  V,W: My_Enum;
end record;
pragma Pack(rec);
O: rec; -- will occupy one storage unit

```

- A formal parameter of the type is sized according to calling conventions rather than size specifications of the type. Appropriate size conversions upon parameter passing take place automatically and are transparent to the user.

- Adjacent bits to an object that is a component of a composite object, but whose size is non-referable, may be affected by assignments to the object, unless these bits are occupied by other components of the composite object; that is, whenever possible, a component of non-referable size is made referable.

In all cases, the compiler generates correct code for all operations on objects of the type, even if they are stored with differing representational size in different contexts.

Note: A size specification cannot be used to force a certain size in value operations of the type; for example

```

type my_int is range 0..65535;
for my_int'size use 16; -- o.k.
A,B: my_int;
...A + B... -- this operation will generally be
             -- executed on 32-bit values

```

3. A size specification for a type specifies the size for objects of this type and of all its subtypes. For components of composite types, whose subtype would allow a shorter representation of the component, no attempt is made to take advantage of such shorter representations. In contrast, for types without a length clause, such components may be represented in a lesser number of bits than the number of bits required to represent all values of the type. Thus, in the example

```

type MY_INT is range 0..2**15-1;
for MY_INT'SIZE use 16; -- (1)
subtype SMALL_MY_INT is MY_INT range 0..255;
type R is record
    ...
    X: SMALL_MY_INT;
    ...
end record;

```

the component R.X will occupy 16 bits. In the absence of the length clause at (1), R.X may be represented in 8 bits.

For the following type classes, the size specification must coincide with the default size chosen by the compiler for the type:

- access types
- floating-point types
- task types

No useful effect can be achieved by using size specifications for these types.

5.4.2.2. Size Specification for Scalar Types

The specified size must accommodate all possible values of the type including the value 0 (even if 0 is not in the range of the values of the type). For numeric types with negative values the number of bits must account for the sign bit. No skewing of the representation is attempted. Thus

```
type my_int is range 100..101;
```

requires at least 7 bits, although it has only two values, while

```
type my_int is range -101..-100;
```

requires 8 bits to account for the sign bit.

A size specification for a real type does not affect the accuracy of operations on the type. Such influence should be exerted via the `accuracy_definition` of the type (LRM 3.5.7, 3.5.9).

A size specification for a scalar type may not specify a size larger than the largest operation size supported by the target architecture for the respective class of values of the type.

5.4.2.3. Size Specification for Array Types

A size specification for an array type must be large enough to accommodate all components of the array under the densest packing strategy explained below in adherence to any alignment constraints on the component type (see Section 5.4.7).

The size of the component type cannot be influenced by a length clause for an array. Within the limits of representing all possible values of the component subtype (but not necessarily of its type), the representation of components may, however, be reduced to the minimum number of bits, unless the component type carries a size specification.

If there is a size specification for the component type, but not for the array type, the component size is rounded up to a referable size, unless `pragma PACK` is given. This applies even to boolean types or other types that require only a single bit for the representation of all values.

5.4.2.4. Size Specification for Record Types

A size specification for a record type does not influence the default type mapping of a record type. The size must be at least as large as the number of bits determined by type mapping. Influence over packing of components can be exerted by means of (partial) record representation clauses or by `Pragma PACK`.

Neither the size of component types, nor the representation of component subtypes can be influenced by a length clause for a record.

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record or contain relative offsets of components within a record layout for record components of dynamic size. These implementation-dependent components cannot be named or sized by the user.

A size specification cannot be applied to a record type with components of dynamically determined size.

Note: Size specifications for records can be used only to widen the representation accomplished by padding at the beginning or end of the record. Any narrowing of the representation over default type mapping must be accomplished by representation clauses or pragma `PACK`.

5.4.2.5. Specification of Collection Sizes

The specification of a collection size causes the collection to be allocated with the specified size. It is expressed in storage units and need not be static; refer to package `SYSTEM` for the meaning of storage units.

Any attempt to allocate more objects than the collection can hold causes a `STORAGE_ERROR` exception to be raised. Dynamically sized records or arrays may carry hidden administrative storage requirements that must be accounted for as part of the collection size. Moreover, alignment constraints on the type of the allocated objects may make it impossible to use all memory locations of the allocated collection. Furthermore, some administrative overhead for the allocator must be taken into account by the user (currently 1 word per allocated object).

In the absence of a specification of a collection size, the collection is extended automatically if more objects are allocated than possible in the collection originally allocated with the compiler-established default size. In this case, `STORAGE_ERROR` is raised only when the available target memory is exhausted. If a collection size of zero is specified, no access collection is allocated.

5.4.2.6. Specification of Task Activation Size

The specification of a task activation size causes the task activation to be allocated with the specified size. It is expressed in storage units; refer to package `SYSTEM` for the meaning of storage units.

Any attempt to exceed the activation size during execution causes a `STORAGE_ERROR` exception to be raised. Unlike collections, there is generally no extension of task activations.

5.4.2.7. Specification of 'SMALL

Only powers of 2 are allowed for 'SMALL.

The length of the representation may be affected by this specification. If a size specification is also given for the type, the size specification takes precedence; the specification of 'SMALL must then be accommodatable within the specified size.

5.4.3. Enumeration Representation Clauses

For enumeration representation clauses [LRM 13.3], the following restrictions apply:

- The internal codes specified for the literals of the enumeration type may be any integer value between `INTEGER' FIRST` and `INTEGER' LAST`. It is strongly advised to not provide a representation clause that merely duplicates the default mapping of enumeration types, which assigns consecutive numbers in ascending order starting with 0, since unnecessary runtime cost is incurred by such duplication. It should be noted that the use of attributes on enumeration types with user-specified encodings is costly at run time.
- Array types, whose index type is an enumeration type with non-contiguous value encodings, consist of a contiguous sequence of components. Indexing into the array involves a runtime translation of the index value into the corresponding position value of the enumeration type.

5.4.4. Record Representation Clauses

The alignment clause of record representation clauses [LRM 13.4] is observed. The specified expression must yield a target-dependent value.

Static objects may be aligned at powers of 2 up to a page boundary. The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record forced by the component allocation and the minimum alignment requirements of the components is already more stringent than the specified alignment.

The component clauses of record representation clauses are allowed only for components and discriminants of statically determinable size. Not all components need to be present. Component clauses for components of variant parts are allowed only if the size of the record type is statically determinable for every variant.

The size specified for each component must be sufficient to allocate all possible values of the component subtype (but not necessarily the component type). The location specified must be compatible with any alignment constraints of the component type: an alignment constraint on a component type may cause an implicit alignment constraint on the record type itself.

If some, but not all, discriminants and components of a record type are described by a component clause, then the discriminants and components without component clauses are allocated after those with component clauses: no attempt is made to utilize gaps left by the user-provided allocation.

5.4.5. Address clauses

Address clauses [LRM 13.5] are supported with the following restrictions:

- When applied to an object, an address clause becomes a linker directive to allocate the object at the given address. For any object not declared immediately within a top-level library package, the address clause is meaningless. Address clauses applied to local packages are not supported by Tartan Ada. Address clauses applied to library packages are prohibited by the syntax; therefore, an address clause can be applied only to a package if it is a body stub.
- Address clauses applied to subprograms and tasks are implemented according to the LRM rules. When applied to an entry, the specified value identifies an interrupt in a manner customary for the target. Immediately after a task is created, a runtime call is made for each of its entries having an address clause, establishing the proper binding between the entry and the interrupt.
- Specified addresses must be constants.

5.4.6. Pragma PACK

Pragma PACK [LRM 13.1] is supported. For details, refer to the following sections.

5.4.6.1. Pragma PACK for Arrays

If pragma PACK is applied to an array, the densest possible representation is chosen. For details of packing, refer to the explanation of size specifications for arrays (Section 5.4.2.3).

If, in addition, a length clause is applied to

1. the array type, the pragma has no effect, since such a length clause already uniquely determines the array packing method.
2. the component type, the array is packed densely, observing the component's length clause. Note that the component length clause may have the effect of preventing the compiler from packing as densely as would be the default if pragma PACK is applied where there was no length clause given for the component type.

5.4.6.2. The Predefined Type String

Package STANDARD applies Pragma PACK to the type string. However, when applied to character arrays, this pragma cannot be used to achieve denser packing than is the default for the target: 1 character per 8-bit word.

5.4.6.3. Pragma PACK for Records

If pragma PACK is applied to a record, the densest possible representation is chosen that is compatible with the sizes and alignment constraints of the individual component types. Pragma PACK has an effect only if the sizes of some component types are specified explicitly by size specifications and are of non-referable nature. In the absence of pragma PACK, such components generally consume a referable amount of space.

It should be noted that default type mapping for records maps components of boolean or other types that require only a single bit to a single bit in the record layout, if there are multiple such components in a record. Otherwise, it allocates a referable amount of storage to the component.

If pragma PACK is applied to a record for which a record representation clause has been given detailing the allocation of some but not all components, the pragma PACK affects only the components whose allocation has not been detailed. Moreover, the strategy of not utilizing gaps between explicitly allocated components still applies.

5.4.7. Minimal Alignment for Types

Certain alignment properties of values of certain types are enforced by the type mapping rules. Any representation specification that cannot be satisfied within these constraints is not obeyed by the compiler and is appropriately diagnosed.

Alignment constraints are caused by properties of the target architecture, most notably by the capability to extract non-aligned component values from composite values in a reasonably efficient manner. Typically, restrictions exist that make extraction of values that cross certain address boundaries very expensive, especially in contexts involving array indexing. Permitting data layouts that require such complicated extractions may impact code quality on a broader scale than merely in the local context of such extractions.

Instead of describing the precise algorithm of establishing the minimal alignment of types, we provide the general rule that is being enforced by the alignment rules:

- No object of scalar type including components or subcomponents of a composite type, may span a target-dependent address boundary that would mandate an extraction of the object's value to be performed by two or more extractions.

5.5. IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record. These components cannot be named by the user.

5.6. INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES

Address clauses to entries are not supported.

5.7. RESTRICTIONS ON UNCHECKED CONVERSIONS

Tartan supports UNCHECKED_CONVERSION with a restriction that requires the sizes of both source and target types to be known at compile time. The sizes need not be the same. If the value in the source is wider than that in the target, the source value will be truncated. If narrower, it will be zero-extended. Calls on instantiations of UNCHECKED_CONVERSION are made inline automatically.

5.8. IMPLEMENTATION-DEPENDENT ASPECTS OF INPUT-OUTPUT PACKAGES

Tartan Ada supports all predefined input/output packages [LRM Chapter 14] with the exception of `LOW_LEVEL_IO`.

`SEQUENTIAL_IO` and `DIRECT_IO` may not be instantiated on types whose representation size is greater than 32255 bytes. Any attempt to read or write values of such types raises `USE_ERROR`.

`SEQUENTIAL_IO` and `DIRECT_IO` may not be instantiated on unconstrained array types, nor on record types with discriminants without default values.

An attempt to delete an external file while more than one internal file refers to this external file raises `USE_ERROR`.

When an external file is referenced by more than one internal file, an attempt to reset one of those internal files to `OUT_FILE` raises `USE_ERROR`.

An attempt to create a file with `FILE_MODE IN_FILE` raises `USE_ERROR`.

Since the implementation of the input-output packages uses buffers, output to one file cannot necessarily be read immediately from another file associated with the same external file.

The `FORM` parameter of file management subprograms is ignored.

An attempt to read a non-existent data record through the operations of `SEQUENTIAL_IO` or `DIRECT_IO` raises `DATA_ERROR`, except that `END_ERROR` is raised when reading beyond the end of file.

If the VMS record management services (RMS) return a status value that cannot be mapped onto a predefined Ada exception, the exception `DEVICE_ERROR` is raised.

5.9. OTHER IMPLEMENTATION CHARACTERISTICS

The following information is supplied in addition to that required by Appendix F to MIL-STD-1815A.

5.9.1. Definition of a Main Program

Any Ada library subprogram unit may be designated the main program for purposes of linking (using the `ALIB` command) provided that the subprogram has no parameters.

Tasks initiated in imported library units follow the same rules for termination as other tasks (described in LRM 9.4 (6-10)). Specifically, these tasks are not terminated simply because the main program has terminated. Terminate alternatives in selective wait statements in library tasks are therefore strongly recommended.

5.9.2. Implementation of Generic Units

All instantiations of generic units, except the predefined generic `UNCHECKED_CONVERSION` and `UNCHECKED_DEALLOCATION` subprograms, are implemented by code duplications. No attempt at sharing code by multiple instantiations is made in this release of Tartan Ada. (Code sharing will be implemented in a later release.)

Tartan Ada enforces the restriction that the body of a generic unit must be compiled before the unit can be instantiated. It does not impose the restriction that the specification and body of a generic unit must be provided as part of the same compilation. A recompilation of the body of a generic unit will obsolete any units that instantiated this generic unit.

5.9.3. Implementation-Defined Characteristics in Package STANDARD

The implementation-dependent characteristics in package STANDARD [Annex C] are:

package STANDARD is

```

...
type BYTE_INTEGER is range -128 .. 127;
type SHORT_INTEGER is range -32768 .. 32767;
type INTEGER is range -2_147_483_648 .. 2_147_483_647;
type FLOAT is digits 6 range -16#0.7FFF_FF8#E+32 .. 16#0.7FFF_FF8#E+32;

type LONG_FLOAT is digits 9 range -16#0.7FFF_FFFF_FFFF_FE#E+256 ..
16#0.7FFF_FFFF_FFFF_FE0#E+256 ;
type DURATION is delta 0.0001 range -86400.0 .. 86400.0;
-- DURATION' SMALL = 2#1.0#E-14 (that is, 6.103516E-5 sec)
...
end STANDARD;
```

5.9.4. Attributes of Type Duration

The type DURATION is defined with the following characteristics:

```

DURATION' DELTA is 0.0001 sec
DURATION' SMALL is 6.103516E-5 sec
DURATION' FIRST is -86400.0 sec
DURATION' LAST is 86400.0 sec
```

5.9.5. Values of Integer Attributes

Taran Ada supports the predefined integer type INTEGER. The range bounds of the predefined type INTEGER are:

```

INTEGER' FIRST = -2**31
INTEGER' LAST = 2**31-1
```

The range bounds for subtypes declared in package TEXT_IO are:

```

COUNT' FIRST = 0
COUNT' LAST = INTEGER' LAST - 1

POSITIVE_COUNT' FIRST = 1
POSITIVE_COUNT' LAST = INTEGER' LAST - 1

FIELD' FIRST = 0
FIELD' LAST = 20
```

The range bounds for subtypes declared in packages DIRECT_IO are:

```

COUNT' FIRST = 0
COUNT' LAST = INTEGER' LAST

POSITIVE_COUNT' FIRST = 1
POSITIVE_COUNT' LAST = COUNT' LAST
```

5.9.6. Values of Floating-Point Attributes

Tartan Ada supports the predefined floating-point types `FLOAT` and `LONG_FLOAT`. `FLOAT` maps onto the VAX F-format floating-point representation; `LONG_FLOAT`, onto the D-format.

<u>Attribute</u>	<u>Value for FLOAT</u>
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON approximately	16#0.1000_000#E-4 9.53674E-07
SMALL approximately	16#0.8000_000#E-21 2.58494E-26
LARGE approximately	16#0.FFFF_F80#E+21 1.93428E+25
SAFE_EMAX	127
SAFE_SMALL approximately	16#0.1000_000#E-31 2.93874E-39
SAFE_LARGE approximately	16#0.7FFF_FC0#E+32 1.70141E+38
FIRST approximately	-16#0.7FFF_FF8#E+32 -1.70141E+38
LAST approximately	16#0.7FFF_FF8#E+32 1.70141E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	127
MACHINE_EMIN	-127
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

<u>Attribute</u>	<u>Value for LONG FLOAT</u>
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON approximately	16#0.4000_0000_0000_000#E-7 9.3132257461548E-10
SMALL approximately	16#0.8000_0000_0000_000#E-31 2.3509887016446E-38
LARGE approximately	16#0.FFFF_FFFE_0000_000#E+31 2.1267647922655E+37
SAFE_EMAX	127
SAFE_SMALL approximately	16#0.1000_0000_0000_000#E-31 2.9387358770557E-39
SAFE_LARGE approximately	16#0.7FFF_FFFF_0000_000#E+32 1.7014118338124E+38
FIRST approximately	-16#0.7FFF_FFFF_FFFF_FF8#E+32 -1.7014118346047E+38
LAST approximately	16#0.7FFF_FFFF_FFFF_FF8#E+32 1.7014118346047E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	56
MACHINE_EMAX	127
MACHINE_EMIN	-127
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

C

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. The use of the '*' operator signifies a multiplication of the following character, and the use of the '&' character signifies concatenation of the preceding and following strings. The values within single or double quotation marks are to highlight character or string values:

Name and Meaning	Value
\$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
\$BIG_ID1 An identifier the size of the maximum input line length which is identical to \$BIG_ID2 except for the last character.	239 * 'A' & '1'
\$BIG_ID2 An identifier the size of the maximum input line length which is identical to \$BIG_ID1 except for the last character.	239 * 'A' & '2'
\$BIG_ID3 An identifier the size of the maximum input line length which is identical to \$BIG_ID4 except for a character near the middle.	120 * 'A' & '3' & 119 * 'A'

Name and Meaning	Value
\$BIG_ID4 An identifier the size of the maximum input line length which is identical to \$BIG_ID3 except for a character near the middle.	120 * 'A' & '4' & 119 * 'A'
\$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	237 * '0' & "298"
\$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	235 * '0' & "690.0"
\$BIG_STRING1 A string literal which when catenated with \$BIG_STRING2 yields the image of \$BIG_ID1.	'"' & 120 * 'A' & '"'
\$BIG_STRING2 A string literal which when catenated to the end of \$BIG_STRING1 yields the image of \$BIG_ID1.	'"' & 119 * 'A' & '1' & '"'
\$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	220 * ' '
\$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483646
\$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	1_000_000
\$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
\$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	VAX
\$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
\$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	20
\$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
\$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
\$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	131_073.0
\$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	200
\$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	ILLEAL_EXTERNAL_FILE_NAME1@
\$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	ILLEAL_EXTERNAL_FILE_NAME2*

Name and Meaning	Value
\$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
\$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
\$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
\$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
\$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-131_073.0
\$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	10
\$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
\$MAX_DIGITS Maximum digits supported for floating-point types.	9
\$MAX_IN_LEN Maximum input line length permitted by the implementation.	240
\$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
\$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648

Name and Meaning	Value
<p>\$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.</p>	"2:" & 235 * '0' & "11:"
<p>\$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.</p>	"16:" & 233 * '0' & "F.E:"
<p>\$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.</p>	''' & 238 * 'A' & '''
<p>\$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.</p>	-2147483648
<p>\$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.</p>	32
<p>\$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.</p>	BYTE_INTEGER
<p>\$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.</p>	VAX
<p>\$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.</p>	16#FFFFFFFE#

Name and Meaning	Value
<p><code>\$NEW_MEM_SIZE</code> An integer literal whose value is a permitted argument for pragma <code>MEMORY_SIZE</code>, other than <code>\$DEFAULT_MEM_SIZE</code>. If there is no other value, then use <code>\$DEFAULT_MEM_SIZE</code>.</p>	1_000_000
<p><code>\$NEW_STOR_UNIT</code> An integer literal whose value is a permitted argument for pragma <code>STORAGE_UNIT</code>, other than <code>\$DEFAULT_STOR_UNIT</code>. If there is no other permitted value, then use value of <code>SYSTEM.STORAGE_UNIT</code>.</p>	8
<p><code>\$NEW_SYS_NAME</code> A value of the type <code>SYSTEM.NAME</code>, other than <code>\$DEFAULT_SYS_NAME</code>. If there is only one value of that type, then use that value.</p>	VAX
<p><code>\$TASK_SIZE</code> An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.</p>	96
<p><code>\$TICK</code> A real literal whose value is <code>SYSTEM.TICK</code>.</p>	0.01667

APPENDIX D

WITHDRAWN TESTS

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

- a. E28005C This test expects that the string "-- TOP OF PAGE. -- 63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.
- b. A39005G This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).
- c. B97102E This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).
- d. C97116A This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING-OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.
- e. BC3009B This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).
- f. CD2A62D This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).
- g. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests] These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

- h. CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests] These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).
- i. 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.
- j. CD2D11B This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.
- k. CD5007B This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).
- l. 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.
- m. CD7105A This test requires that successive calls to CALENDAR.-CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).
- n. CD7203B, & CD7204B These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.
- o. CD7205D This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.
- p. CE2107I This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

- q. CE3111C This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.
- r. CE3301A This test contains several calls to END_OF_LINE & 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).
- s. CE3411B This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

Chapter 3

Compiling Ada Programs

The TADA command is used to compile and assemble Ada compilation units.

3.1. THE TADA COMMAND FORMAT

The TADA command has this format:

```
TADA [/qualifier [(option, ...) ...]] file-spec [/qualifier [(option, ...) ...]]
```

By default, if no qualifier is supplied, the compiler invokes the VMS-targeted compiler if it has been installed on your system. If an attempt is made to invoke a compiler that has not been installed on your system, the following error message is issued:

```
S TADA RADAR_SCREEN.ADA
%DCL-W-ACTIMAGE, error activating image TADAHOME:TADA.EXE
-CLI-E-IMAGEFNF, image file not found DRA0:[COMPILERS.ADA]TADA.EXE;
```

The parameter *file-spec* is a source file name. Since the source files need not reside in the directory in which the compilation takes place, *file-spec* must include sufficient directory information to locate the file. If no extension is supplied with the file name, a default extension of .ADA will be supplied by the compiler.

TADA will accept only one source file per compilation. The source file may contain more than one compilation unit, but it is considered good practice to place only one compilation unit in a file. The compiler sequentially processes all compilation units in the file. Upon successful compilation of a unit,

- the Ada program library ADA.DB is updated to reflect the new compilation time and any new dependencies
- one or more separate compilation files and/or object files are generated

If no errors are detected in a compilation unit, the compiler produces an object module and updates the library. If any error is detected, no object code file is produced, a source listing is produced, and no library entry is made for that compilation unit. If warnings are generated, both an object code file and a source listing are produced, and the library is updated.

The output from Tartan Ada VMS is a file whose type depends on both the Tool Set selected in creating the program library (see Section 4.2) and whether the unit compiled is a specification or body unit. See Section 3.4 for a list of these file types as well as the extensions of other files that may be left in the directory.

3.2. COMMAND QUALIFIERS

Command qualifiers indicate special actions to be performed by the compiler or special output file properties. The following qualifiers are available:

```
/CROSS_REFERENCE
/NO_CROSS_REFERENCE [default]
```

Controls whether the compiler generates a cross-reference table of linknames for the compilation unit. The table will be placed in the file *unit-name.XRF* (See Section 3.5).

/DEBUG(=*option*{,...})
/NODEBUG (default)

Controls whether the compiler makes a local symbol table and traceback information available to the debugger and the runtime error reporting mechanism. The available options are:

- ALL** The compiler provides both local symbol definitions and traceback information in the object module for the debugger. The **/DEBUG=ALL** qualifier is equivalent to specifying **/DEBUG=(SYMBOLS, TRACEBACK)**.
- NONE** The compiler provides no debugging information in the object module. The **/DEBUG=NONE** qualifier is equivalent to **/NODEBUG**.
- SYMBOLS** The compiler provides local symbol definitions in the object module.
- TRACEBACK** The compiler provides traceback information in the object module.

If neither form of this qualifier is given, the default is **/NODEBUG**, equivalent to **/DEBUG=NONE**. If the **/DEBUG** qualifier is given without options, the options specified by default are **SYMBOLS, TRACEBACK**, equivalent to **/DEBUG=ALL**.

/LIST(=*option*)
/NOLIST

Controls whether a listing file is produced. If produced, the file has the source file name and a **.LIS** extension. The available options are:

- ALWAYS** Always produce a listing file
- NEVER** Never produce a listing file, equivalent to **/NOLIST**
- ERROR** Produce a listing file only if a compilation error or warning occurs

When no form of this qualifier is supplied in the command line, the default condition is **/LIST=ERROR**. When the **LIST** qualifier is supplied without an option, the default option is **ALWAYS**.

/MACHINE_CODE
/NOMACHINE_CODE (default)

Controls whether the assembly code files produced by the compiler are retained in the user's directory after compilation is complete. This qualifier is useful if the user wishes to inspect the compiler output for code correctness and quality.

/NOENUMIMAGE

Controls whether compiler omits data segments with the text of enumeration literals. This text is normally produced for exported enumeration types in order to support the **'IMAGE** attribute. You should use **/NOENUMIMAGE** only when you can guarantee that no unit that will import the enumeration type will use **'IMAGE**. However, if you are compiling a unit with an enumeration type that is not visible to other compilation units, this qualifier is not needed. The compiler can recognize when **'IMAGE** is not used and will not generate the supporting strings. The **/NOENUMIMAGE** qualifier cannot be negated.

OVERFLOW_CHECK	Will suppress compile-time checks for overflow, but the hardware does not permit efficient runtime checks, so none are done.
RANGE_CHECK	As specified in the Ada LRM, Section 11.7.
STORAGE_CHECK	As specified in the Ada LRM, Section 11.7. Suppresses only stack checks in generated code, not the checks made by the allocator as a result of a new operation.

3.3. WHAT UPDATES ARE MADE TO THE PROGRAM LIBRARY

Simply stated, upon successful compilation of a unit,

- the Ada program library ADA.DB is updated to reflect the new compilation time and any new dependencies
- one or more separate compilation files and/or object files are generated.

However, more complicated situations can arise. The following items list the types of compilation units and address the range of situations that can arise.

- In all cases the transitive closure of the dependencies of a compilation unit in the library must be consistent; that is, the unit must be compiled consistently as defined in section 10.3 of the LRM. A secondary unit can have the specification of its ancestor in its context clause, although it is redundant. For a more complete discussion of closure, see Section 4.5.
- A **package specification** replaces any library unit in the library with the same name, or is simply added if no such library unit exists.
- A **package body** replaces any existing body of a package specification with the same name. If no such specification exists, an error message is issued. If such a specification exists, but the body does not match the specification in the sense of Section 7.1 of the LRM, error messages are issued.
- A **subprogram specification** replaces any library unit in the library with the same name, or is simply added if no such library unit exists.
- A **subprogram body** replaces any existing body of a (generic or non-generic) subprogram specification with the same name. If no such specification exists, an implicit specification is derived from the body and entered into the library. If such a specification exists, but the body does not match the specification in the sense of Section 6.3 of the LRM, error messages are issued. If any library unit other than a subprogram specification exists with the same name, the new implicit specification replaces that library unit.
- **Generic package specifications and subprogram specifications** act as explicit specifications. i.e., corresponding bodies must match their specifications. If a generic unit is instantiated, a dependency is created on the generic body.
- **Generic instances** compiled as library units are treated in the same way as their non-generic counterparts.
- When an instantiation replaces the body of a library unit, all secondary units of that library unit are now obsolete and are deleted.
- A **subunit** with a parent unit containing an appropriate body stub existing in the library replaces any subunit with the same subunit name, comprised of the stub's name and the name of the ancestor unit, or is simply added, if no such subunit exists. A unit containing stubs will only be entered into the library if the simple names of all its stubs are unique for all stubs derived from its common ancestor. An error message is issued if no parent unit exists in the library, the parent unit exists but does not have a relevant stub, or the parent unit and body stub exist but the subunit does not match the stub or its specification.
- When the parent unit of a subunit is recompiled and the parent no longer contains a stub for the subunit, the subunit, which is now obsolete, is deleted.

used. A reference to this file specification (by means of the supplied Ada-name) will be entered into the current Ada library. The file specification must contain the file name and type. No wildcard characters are allowed in the file specification.

The following command qualifier may be used:

/OLB Informs the Ada Librarian that the *file-spec* is actually that of an object library. All object files brought into the Tartan Ada Library environment with the **/OLB** qualifier will be linked with the VMS **/LIBRARY** option.

If a specification for a foreign body is deleted from the program library, the database entry for the body is also deleted. The object file is *not* deleted.

Examples:

1. The ALIB FOREIGN_BODY command replaces the standard package *adatetime* with a reference to a timing module written in FORTRAN.

```
S ALIB FOREIGN_BODY adatetime USER01:[PROJECT]fortran_time.OBJ
```

A pointer for this package in the file USER01:[PROJECT]fortran_time.OBJ has now been inserted in the ADA.DB file.

2. The ALIB FOREIGN_BODY creates a reference to the graphics object library *testgraphics*.

```
S ALIB FOREIGN_BODY newplot USER01:[PROJECT]testgraphics.olb/OLB
```

It may now be referenced with the Ada-name *newplot*.

4.8. THE LINK SUBCOMMAND

The LINK command checks that the unit within the library specified by the user has the legal form for a main unit, checks all its dependencies, finds all required object files, and links the main program with its full closure (See Section 4.5) producing an executable image. The format of the LINK command is

```
S ALIB LINK[/qualifier...] library-unit-name[/qualifier...]
```

where the parameter *library-unit-name* specifies the unit in the library to be made the main program and must be supplied.

The ALIB LINK command calls the VMS Linker from which the output file is *library-unit-name.OBJ*.

The message

```
Ada Librarian error 6500 (Link failed)
```

with no additional messages, can be clarified by relinking the program after typing to VMS:

```
S SET MESSAGE /FACILITY/IDENTIFICATION/SEVERITY/TEXT
```

The following qualifiers may be used:

/BRIEF Requests that the linker produce a brief form of the map (memory allocation) file. **/BRIEF** is valid only if **/MAP** is also specified. The brief form of the map contains:

- A summary of the image characteristics
- A list of all object modules included in the image
- A summary of link-time performance statistics

/CONTIGUOUS

/NOCONTIGUOUS [default]

Controls whether the output image file is contiguous.

/CROSS_REFERENCE

/NOCROSS_REFERENCE [default]

Controls whether the memory allocation listing (map) contains a symbol cross-reference. The **/CROSS_REFERENCE** qualifier is valid only if **/MAP** is also specified and **/BRIEF** is not specified. A symbol cross-reference lists each global symbol referenced in the image, its value, and all modules in the image that refer to it.

/DEBUG[=*file-spec*]

/NODEBUG [default]

Controls whether a debugger is included in the output image. If you specify **/DEBUG** without a file specification, the VMS Symbolic Debugger is linked with the image. For information on using the VMS Symbolic Debugger, see the *VAX/VMS Utilities Reference Volume*. The **/DEBUG** qualifier optionally accepts the name of an alternate, user-specified debugger supplied as a *file-spec*. No wildcard characters are allowed in the file specification. If a file specification is entered without a file type, the linker assumes the default file type of **OBJ**. If the object module contains local symbol table or traceback information for the debugger, you must specify **/DEBUG** to include the information in the image as well. If the object module does not contain symbol table or traceback information, and you specify **/DEBUG**, only global symbols are available for symbolic debugging.

/EXECUTABLE[=*file-spec*] [default]

/NOEXECUTABLE

Controls whether the linker creates an executable image and optionally provides a file specification for the output image file. No wildcard characters are allowed in the file specification. By default, the linker creates an executable image with the same file name as the main program (truncated to 32 characters) and a file type of **EXE**. You can use **/NOEXECUTABLE** or **/EXECUTABLE=NL:** to test a set of qualifiers, options, or input object modules, without creating an image file.

/FULL

Requests that the linker produce a full form of the map (memory allocation) listing. **/FULL** is valid only if **/MAP** is specified. A full listing contains the following information:

- All the information included in the brief listing (see **/BRIEF**)
- Detailed descriptions of each program section and image section in the image file
- Lists of global symbols by name and by value

/HEADER

Provides a header on a system image when used in conjunction with the **/SYSTEM** qualifier. All other images always have headers. However, by default, system images do not have headers.

/MAP[=*file-spec*]

/NOMAP

Controls whether a memory allocation listing (map) is produced and optionally defines the file specification. By default, the file will have the same file name as the executable code and will have the file type **MAP**. If you specify **/MAP**, you can also specify **/BRIEF**, **/FULL**, or **/CROSS_REFERENCE** to control the contents of the map. If you do not specify any of these qualifiers, the map contains:

- All the information contained in a brief listing (see **/BRIEF**)

- A list of user-defined global symbols by name
- A list of user-defined program sections

When you specify `/MAP`, you can control the defaults applied to the output file specification, as described in the *VAX/VMS DCL Dictionary*, Section 2.4.4. In interactive mode, the default is `/NOMAP`. However, in batch mode, the default is `/MAP`.

`/OPTIONS[=file-spec]`
`/NOOPTIONS`

Indicates that the associated input file contains a list of options to control the linking. If you specify `/OPTIONS`, an options file must be supplied. For complete details on the contents of an options file, see the discussion of the linker in the *VAX/VMS Utilities Reference Volume*. The format of this qualifier used for ALIB LINK differs from the VAX/VMS linker syntax.

`/POIMAGE`

Directs the linker to create an image that is stored only in P0 address space. The linker places the stack and VAX RMS buffers that usually go in P1 address space in P0 address space. The `/POIMAGE` qualifier is used to create executable images that modify P1 address space. For a description of P0 and P1 address space, see the *VAX Architecture Handbook*.

`/PROTECT`

When used in conjunction with the `/SHAREABLE` qualifier, the `/PROTECT` qualifier directs the linker to create a protected shareable image. A protected shareable image can execute privileged change mode instructions even when it is linked into a nonprivileged executable image.

`/SELECTIVE_SEARCH`

Indicates that the associated input file is an object module and that any symbols defined in the module that are not necessary to resolve outstanding references should be excluded from the symbol table of the output image file. These symbols are also excluded from the symbol table file, if `/SYMBOL_TABLE` is specified. The binary code in the object module is always included.

`/SHAREABLE[=file-spec]`

`/NOSHAREABLE [default]`

Requests that the linker produce a shareable image file rather than an executable image. By default, the linker creates an executable image. If you specify both `/EXECUTABLE` and `/SHAREABLE`, the `/SHAREABLE` qualifier always takes precedence. No wildcard characters are allowed in the file specification. Shareable images should not be run with the RUN command. However, they can be linked with object modules to create executable images. To specify an input shareable image, the `/SHAREABLE` qualifier must be used as an input file qualifier in an options file. See the discussion of the linker in the *VAX/VMS Utilities Reference Volume*.

`/SYMBOL_TABLE[=file-spec]`

`/NOSYMBOL_TABLE [default]`

Requests that the linker create a separate file containing symbol definitions for all global symbols in the image. The output file will be in object module format. No wildcard characters are allowed in the file specification. By default, the file will have the same file name as the executable code and will have the file type STB. If you also specify `/DEBUG`, the linker includes the global symbol definitions in the image for use by the debugger, and also creates a separate symbol table file. The

symbol table file can be used as input to subsequent LINK commands, to provide the symbol definitions to other images.

/SYSLIB(default)
/NOSYSLIB

Controls whether the default system library is automatically searched for unresolved references. For further details, see the discussion of the linker in the *VAX/VMS Utilities Reference Volume*.

/SYSSHR(default)
/NOSYSSHR

Controls whether the linker searches the default system shareable image library, `IMAGELIB.OLB` when it cannot resolve references in the input file(s) specified.

/SYSTEM[=*base-address*]
/NOSYSTEM [default]

Requests the linker to produce a system image and optionally defines a base address for the image. If you specify `/SYSTEM`, you cannot specify either `/SHAREABLE` or `/DEBUG`. The base address specifies the virtual memory location at which the image is to be loaded. The address can be expressed in decimal, hexadecimal, or octal format, using the radix specifiers `%D`, `%X`, or `%O`, respectively. If you do not specify a base address, the linker uses the default address of `%X80000000`. A system image cannot be run with the `RUN` command; it must be bootstrapped or otherwise loaded into memory. System images are intended for special purposes, such as stand-alone operating system diagnostics. When the linker creates a system image, it orders the program sections in alphanumeric order and ignores all program section attributes.

/TRACEBACK (default)
/NOTRACEBACK

Controls whether the linker includes traceback information in the image file. By default, the linker includes traceback information so that the system can trace the call stack when an error occurs. If you specify `/NOTRACEBACK`, there is no traceback reporting when an error occurs. If you specify `/DEBUG`, `/TRACEBACK` is assumed.

/USERLIBRARY [= (*table* [, . . .])]
/NOUSERLIBRARY

Controls whether the linker searches any user-defined default libraries after it has searched any specified user libraries. When you specify the `/USERLIBRARY` qualifier, the linker searches the process, group, and system logical name tables to find the file specifications of the user-defined libraries. (The discussion of the linker in the *VAX/VMS Utilities Reference Volume* explains user-defined default libraries.) You can specify the following tables for the linker to search:

- ALL** The linker searches the process, group, and system logical name tables for user-defined library definitions.
- GROUP** The linker searches the group logical name table for user-defined library definitions.
- NONE** The linker does not search any logical name table; this specification is equivalent to `/NOUSERLIBRARY`.
- PROCESS** The linker searches the process logical name table for user-defined library definitions.

SYSTEM The linker searches the system logical name table for user-defined library definitions.

The /NOUSERLIBRARY qualifier tells the linker not to search any user-defined default libraries. If you do not specify either /NOUSERLIBRARY or /USERLIBRARY=(table), the linker assumes /USERLIBRARY=ALL by default.

Examples:

1. This example uses the qualifiers /MAP and /FULL.

```
S ALIB LINK/MAP/FULL PAYROLL
```

The linker links the Ada unit PAYROLL and creates an executable image named PAYROLL.EXE. The /MAP and /FULL qualifiers request a full map of the image, with descriptions of each program section, lists of global symbols by name and by value, and a summary of the image characteristics. The map file is named PAYROLL.MAP.

2. This command specifies that the VAX linker should use the LINKING.OPT options file in linking the executable image PAYROLL.EXE.

```
S ALIB LINK PAYROLL/OPTIONS=LINKING.OPT/MAP=NEWPAYROLL/FULL/CROSS_REF
```

The map file will be named NEWPAYROLL.MAP and will contain complete information on program sections, image sections, and global symbols as well as a symbol cross reference. The Ada program to be linked is identified by its ada-name, PAYROLL. Note that the map file name does *not* include the file extension in the command.