AD-A215 281

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) **Ada Compiler Validation Summary Report:** TeleSoft, Ada386 Version 3.23, VAX 8350 (Host) to Intel 386-120 (Target), 890602I1.10136	5. TYPE OF REPORT & PERIOD COVERED 02 June 1989 to 02 June 1990 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) IABG, Ottobrunn, Federal Republic of Germany.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS IABG, Ottobrunn, Federal Republic of Germany.	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) IABG, Ottobrunn, Federal Republic of Germany.	15. SECURITY CLASS (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

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

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

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

TeleSoft, Ada386, Version 3.23, Ottobrunn, West Germany, VAX 8350 under VMS, Version 5.0 - 2 (Host) to Intel 386-120 (80386/387) board (bare machine)(Target), ACVC 1.10.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Compiler Validation Summary Report:

Compiler Name: Ada386 Version 3.03

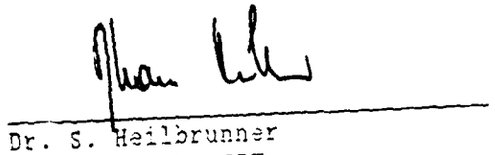
Certificate Number: #890602I1.10136

Host: VAX 8350 under VMS version 5.0 - 2

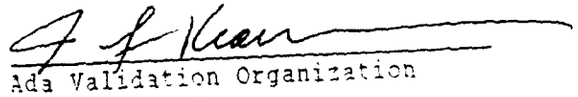
Target: Intel 386-120 (80386/387) board (bare machine).

Testing Completed 2 June 1989 Using ACVC 1.10

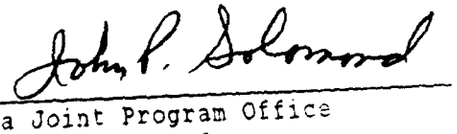
This report has been reviewed and is approved.



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

AVF Control Number: AVF-IABG-035

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #890602I1.10136
TeleSoft
Ada386 Version 3.23
VAX 8350 Host and Intel 386-120 Target

Completion of On-Site Testing:
2 June 1989

Prepared By:
IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn
West Germany

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

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

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

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

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

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

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). On-site testing was completed 2 June 1989 at IABG mbH, Ottobrunn / TeleLOGIC AB, Sweden.

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: Ada386 Version 3.03

ACVC Version: 1.10

Certificate Number: #890602I1.10136

Host Computer:

Machine:	VAX 8350
Operating System:	VMS version 5.0 - 2
Memory Size:	12 MB

Target Computer:

Machine:	Intel 386-120 board with Intel 80386 CPU and Intel 80387 floating point processor.
Operating System:	bare machine
Memory Size:	2 MB

Communications Network: RS 232 interface

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 LONG_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 no extra bits for extra range. (See test C35903A.)

4) CONSTRAINT_ERROR is raised for predefined integer comparison tests, NUMERIC_ERROR is raised for largest integer comparison and membership tests, and no exception is raised for predefined integer membership tests when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

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

6) Underflow is gradual. (See tests C45524A..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 to even. (See tests C46012A..Z (26 tests).)

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

3) The method used for rounding to integer in static universal real expressions is round 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 when a two dimensional array subtype is declared where the large bound is the first one, and no exception for one dimensional array type and subtype declarations, two dimenesional array type declarations and two dimensional array subtype declaration where the large bound is the second one. (See test C36003A.)

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

3) NUMERIC_ERROR is raised when 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 no exception. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

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

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

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 no exception. (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. | index subtype checks are made as choices are evaluated. | the order in which choices are evaluated and index subtype checks are made depends upon the aggregate itself.} (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 (before | 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 procedures, but not for functions. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

This implementation creates a dependance between a generic body and those units which instantiate it. As allowed by AI-408/11, if the body is compiled after a unit that instantiates it, then that unit becomes obsolete.

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 or record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

2) The package DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

3) The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if permanent file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO.

This implementation supports two legal filenames, "console" and "keyboard" which correspond to standard input and standard output respectively (See Appendix F of the Ada Standard in Appendix B of this report).

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 514 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation and 242 executable tests that use file operations not supported by the implementation. Modifications to the code, processing, or grading for 14 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	1129	1826	17	15	45	3159
Inapplicable	2	9	489	0	13	1	514
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	573	544	245	172	99	160	332	132	36	250	341	77	3159
N/A	14	76	136	3	0	0	6	0	5	0	2	28	244	514
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	CD5011O	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, 514 tests were inapplicable for the reasons indicated:

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

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)

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

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

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

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

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

e. Tests C45531M..P and C45532M..P are not applicable because they require the value of SYSTEM.MAX_MANTISSA to be greater than 32.

f. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

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

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

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

j. CA2009C, CA2009F, BC3204C, and BC3205D are not applicable because this implementation creates a dependance between a generic body and those units that instantiate it (See Section 2.2.i and Appendix F of the Ada Standard)

k. LA3004B, EA3004D, and CA3004F are not applicable because this implementation does not support pragma INLINE for functions.

l. CD1009C, CD2A41A..B (2 tests), CD2A41E and CD2A42A..J (10 tests) are not applicable because of restrictions on 'SIZE length clauses for floating point types.

m. CD2A61I..J (2 tests) are not applicable because of restrictions on 'SIZE length clauses for array types.

n. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because of restrictions on 'SIZE length clauses for access types.

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. The following 238 tests are inapplicable because sequential, text, and direct access files are not supported:

CE2102A..C (3 tests)	CE2102G..H (2 tests)
CE2102K	CE2102N..Y (12 tests)
CE2103C..D (2 tests)	CE2104A..D (4 tests)
CE2105A..B (2 tests)	CE2106A..B (2 tests)
CE2107A..H (8 tests)	CE2107L
CE2108A..B (2 tests)	CE2108C..H (6 tests)
CE2109A..C (3 tests)	CE2110A..D (4 tests)
CE2111A..I (9 tests)	CE2115A..B (2 tests)
CE2201A..C (3 tests)	CE2201F..N (9 tests)
CE2204A..D (4 tests)	CE2205A
CE2208B	CE2401A..C (3 tests)
CE2401E..F (2 tests)	CE2401H..L (5 tests)
CE2404A..B (2 tests)	CE2405B
CE2406A	CE2407A..B (2 tests)
CE2408A..B (2 tests)	CE2409A..B (2 tests)
CE2410A..B (2 tests)	CE2411A
CE3102A..B (2 tests)	EE3102C
CE3102F..H (3 tests)	CE3102J..K (2 tests)
CE3103A	CE3104A..C (3 tests)
CE3107B	CE3108A..B (2 tests)
CE3109A	CE3110A
CE3111A..B (2 tests)	CE3111D..E (2 tests)
CE3112A..D (4 tests)	CE3114A..B (2 tests)
CE3115A	EE3203A
CE3208A	EE3301B
CE3302A	CE3305A
CE3402A	EE3402B
CE3402C..D (2 tests)	CE3403A..C (3 tests)

CE3403E..F (2 tests)	CE3404B..D (3 tests)
CE3405A	EE3405B
CE3405C..D (2 tests)	CE3406A..D (4 tests)
CE3407A..C (3 tests)	CE3408A..C (3 tests)
CE3409A	CE3409C..E (3 tests)
EE3409F	CE3410A
CE3410C..E (3 tests)	EE3410F
CE3411A..B (2 tests)	CE3412A
EE3412C	CE3413A
CE3413C	CE3602A..D (4 tests)
CE3603A	CE3604A..B (2 tests)
CE3605A..E (5 tests)	CE3606A..B (2 tests)
CE3704A..F (6 tests)	CE3704M..O (3 tests)
CE3706D	CE3706F..G (2 tests)
CE3804A..P (16 tests)	CE3805A..B (2 tests)
CE3806A..B (2 tests)	CE3806D..E (2 tests)
CE3806G..H (2 tests)	CE3905A..C (3 tests)
CE3905L	CE3906A..C (3 tests)
CE3906E..F (2 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 14 tests.

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

B71001E	B71001K	B71001Q	B71001W	BA3006A	BA3006B
BA3007B	BA3008A	BA3008B	BA3013A (6 and 7M)		

Tests C34005G, C34005J and C34006D returned the result FAILED because of false assumptions that an element in an array or a record type may not be represented more compactly than a single object of that type. The AVO has ruled these tests PASSED if the only message of failure occurs from the requirements of T'SIZE due to the above assumptions (T is the aray type).

Test CE3901A was modified to replace the call of the function REPORT.LEGAL_FILENAME with the string literal "console". Upon execution, the test returned the result PASSED.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the Ada386 Version 3.23 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Ada386 Version 3.23 compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	VAX 8350
Host operating system:	VMS version 5.0 - 2
Target computer:	Intel 386-120 board with Intel 80386 CPU and Intel 80387 floating point processor
Target operating system:	bare machine
Compiler:	Ada386 Version 3.23

The host and target computers were linked via RS 232 interface.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

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

After the test files were loaded to disk, the full set of tests was compiled and linked on the VAX 8350 at IABG mbH, Ottobrunn. All executable test images were then written on magnetic tape and transferred to a second VAX at TeleLOGIC AB, Sweden, where they were downloaded and executed on the target computer. Results were printed from the host computers.

The compiler was tested using command scripts provided by TeleLOGIC AB and reviewed by the validation team. The compiler was tested using the compiler call

```
ADA386/COMPILE/MON/PROC/LIBFILE='BIN_NAME'/VIRT=3000/BIND='TESTNAME'-
      /'OPTIONS' 'TESTFILE'
```

and linked with the command

```
ADA386/LINK/LIBFILE='BIN_NAME'/OPT='PHANT_OPT'-
   /LOAD_MODULE=[]'TESTNAME' 'MAINNAME'
```

B tests were compiled with the /LIST option. See Appendix E for an explanation of compiler and linker options.

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

3.7.3 Test Site

Testing was conducted at IABG mbH, Ottobrunn / TeleLOGIC AB, Sweden and was completed on 2 June 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

TeleSoft and INTEL have submitted the following Declarations of Conformance concerning the Ada386 Version 3.23 compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: TELESOFT
Ada Validation Facility: IABG, West Germany
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name:	Ada386
Version:	3.23
Host Architecture ISA:	VAX 8630
OS & VER #:	VMS 5.0
Target Architecture ISA:	Intel 80386 on Intel 386-120 Board with Intel 80387 coprocessor
OS & VER #:	Bare Machine

Licensee's Declaration

I, the undersigned, representing INTEL CORPORATION, assume shared responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that the Ada language compiler listed, and its host/target performance, is in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

All certificates and registrations for the Ada language compiler listed in this Declaration shall be made in the name of INTEL CORPORATION.

Date: 7/31/89

INTEL CORPORATION
Chris Hughes
General Manager, Development Tools Operation

DECLARATION OF CONFORMANCE

Compiler Implementor: TELESOFT
Ada Valdation Facility: IABG, West-Germany
ACVC Version: 1.10

Base Compiler Name: Ada386
Version: 3.23
Host Architecture ISA: VAX 8350
OS & version #: VMS 5.0-2

Target Architecture ISA: Intel 386-120 board with Intel 80386 and Intel 80387 fpp
OS & version #: Bare Machine

Implementor's Declaration

I, the undersigned, representing TELESOFT, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD 1815A in the compiler listed in this declaration.

I declare that TELESOFT is the owner and that INTEL Corporation is TELESOFT's Licensee of the Ada language compiler listed above and as such shares TELESOFT's responsibility for maintaining said compiler in conformance to ANSI/MIL-STD 1815A. All certificates and registrations for the Ada language compiler listed in this declaration shall be made only in the Licensee's corporate name.

20 July, 1989
Telelogic AB, Ada Products Division

Stefan Bjornson, Manager, Systems Software

Owner's Declaration

I, the undersigned, representing TELESOFT take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that the Ada language compiler listed, and its host/target performance is in compliance with the Ada Language Standard ANSI/MIL-STD 1815A.

A Licensee's Declaration from INTEL Corporation will be filed separately.

20 July, 1989
Telelogic AB, Ada Products Division

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 Ada386 Version 3.23 compiler, as described in this Appendix, are provided by TeleSoft. Unless specifically noted otherwise, references and page numbers in this appendix are consistent with compiler documentation and not with this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

     type INTEGER is range -32768 .. 32767;
     type LONG_INTEGER is range -2147483648 .. 2147483647;

     type FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;
     type LONG_FLOAT is digits 15
        range -8.98846567431158E+307 .. 8.98846567431158E+307;

     type DURATION is delta 2#1.0#E-14 range -86400 .. 86400;

   ...

end STANDARD;
```

ATTACHMENT B

APPENDIX F OF THE LANGUAGE REFERENCE MANUAL

1 Implementation Dependent Pragmas
2 Implementation Dependent Attributes
3 Specification of Package SYSTEM
4 Restrictions on representation clauses
5 Implementation dependent naming
6 Interpretation of expressions in address clauses
7 Restrictions on unchecked conversions
8 I/O Package characteristics

ATTACHMENT B

APPENDIX F

1. Implementation Dependent Pragmas

pragma COMMENT(<string_literal>);
It may only appear within a compilation unit.
The pragma comment has the effect of embedding the given sequence of characters in the object code of the compilation unit.

pragma LINKNAME(<subprogram_name>, <string_literal>);
It may appear in any declaration section of a unit.
This pragma must also appear directly after an interface pragma for the same <subprogram_name>. The pragma linkname has the effect of making string_literal apparent to the linker.

pragma INTERRUPT(Function_Mapping);
It may only appear immediately before a simple accept statement, a while loop directly enclosing only a single accept statement, or a select statement that includes an interrupt accept alternative. The pragma interrupt has the effect that entry calls to the associated entry, on behalf of an interrupt, are made with a reduced call overhead.

pragma IMAGES(<enumeration_type>,Deferred) or
pragma IMAGES(<enumeration_type>,Immediate);
It may only appear within a compilation unit.
The pragma images controls the creation and allocation of the image table for a specified enumeration type. The default is Deferred, which saves space in the literal pool by not creating an image table for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If one of these attributes is used, an image table is generated in the literal pool of the compilation unit in which the attribute appears. If the attributes are used in more than one compilation unit, more than one image table is generated, eliminating the benefits of deferring the table.

pragma SUPPRESS_ALL;
It may appear anywhere that a Suppress pragma may appear as defined by the Language Reference Manual. The pragma Suppress_All has the effect of turning off all checks defined in section 11.7 of the Language Reference Manual. The scope of applicablility of this pragma is the same as that of the pre-defined pragma Suppress.

ATTACHMENT B

APPENDIX F, Cont.

2. Implementation Dependent Attributes

'Offset Attribute

'Offset along with the attribute 'Address, facilitates machine code insertions. For a prefix P that denotes a declared parameter object, P'Offset yields the statically known portion of the address of the first of the storage units allocated to P. The value is the object's offset relative to a base register and is of type Long_Integer.

INTEGER ATTRIBUTES

'Extended_Image Attribute

Usage: X'Extended_Image(Item,Width,Base,Based,Space_IF_Positive)

Returns the image associated with Item as per the Text_Io definition. The Text_Io definition states that the value of Item is an integer literal with no underlines, no exponent, no leading zeros (but a single zero for the zero value) and a minus sign if negative. If the resulting sequence of characters to be output has fewer than Width characters then leading spaces are first output to make up the difference. (LRM 14.3.7:10,14.3.7:11)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more than one parameter. The parameter Item must be an integer value. The resulting string is without underlines, leading zeros, or trailing spaces.

ATTACHMENT B

APPENDIX F, Cont.

Parameter Descriptions:

Item	-- The user specifies the item that he wants the image of and passes it into the function. This parameter is required.
Width	-- The user may specify the minimum number of characters to be in the string that is returned. If no width is specified then the default (0) is assumed.
Base	-- The user may specify the base that the image is to be displayed in. If no base is specified then the default (10) is assumed.
Based	-- The user may specify whether he wants the string returned to be in base notation or not. If no preference is specified then the default (false) is assumed.
Space_If_Positive	-- The user may specify whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified then the default (false) is assumed.

Examples:

Suppose the following subtype was declared:

```
Subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Image(5)                    = "5"
X'Extended_Image(5,0)                  = "5"
X'Extended_Image(5,2)                  = " 5"
X'Extended_Image(5,0,2)                = "101"
X'Extended_Image(5,4,2)                = " 101"
X'Extended_Image(5,0,2,True)           = "2#101#"
X'Extended_Image(5,0,10,False)         = "5"
X'Extended_Image(5,0,10,False,True)    = " 5"
X'Extended_Image(-1,0,10,False,False)  = "-1"
X'Extended_Image(-1,0,10,False,True)   = "-1"
X'Extended_Image(-1,1,10,False,True)   = "-1"
X'Extended_Image(-1,0,2,True,True)     = "-2#1#"
X'Extended_Image(-1,10,2,True,True)    = "    -2#1#"
```

ATTACHMENT B

APPENDIX F, Cont.

'Extended_Value Attribute

Usage: X'Extended_Value(Item)

Returns the value associated with Item as per the Text_Io definition. The Text_Io definition states that given a string, it reads an integer value from the beginning of the string. The value returned corresponds to the sequence input. (LRM 14.3.7:14)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a CONSTRAINT_ERROR is raised.

Parameter Descriptions:

Item -- The user passes to the function a parameter of the predefined type string. The type of the returned value is the base type X.

Examples:

Suppose the following subtype was declared:

```
Subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Value("5")          = 5
X'Extended_Value(" 5")         = 5
X'Extended_Value("2#101#")     = 5
X'Extended_Value("-1")         = -1
X'Extended_Value(" -1")        = -1
```

ATTACHMENT B

APPENDIX F, Cont.

'Extended_Width Attribute

Usage: X'Extended_Width(Base,Based,Space_If_Positive)

Returns the width for subtype of X.

For a prefix X that is a discrete subtype; this attribute is a function that may have multiple parameters. This attribute yields the maximum image length over all values of the type or subtype X.

Parameter Descriptions:

Base	-- The user specifies the base for which the width will be calculated. If no base is specified then the default (10) is assumed.
Based	-- The user specifies whether the subtype is stated in based notation. If no value for based is specified then the default (false) is assumed.
Space_If_Positive	-- The user may specify whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified then the default (false) is assumed.

Examples:

Suppose the following subtype was declared:

```
Subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Width                  = 3  -- "-10"
X'Extended_Width(10)              = 3  -- "-10"
X'Extended_Width(2)               = 5  -- "10000"
X'Extended_Width(10,True)         = 7  -- "-10#10#"
X'Extended_Width(2,True)          = 8  -- "2#10000#"
X'Extended_Width(10,False,True)   = 3  -- " 16"
X'Extended_Width(10,True,False)   = 7  -- "-10#10#"
X'Extended_Width(10,True,True)    = 7  -- " 10#16#"
X'Extended_Width(2,True,True)     = 9  -- " 2#10000#"
X'Extended_Width(2,False,True)    = 6  -- " 10000"
```

ATTACHMENT B

APPENDIX F, Cont.

ENUMERATION ATTRIBUTES

'Extended_Image Attribute

Usage: X'Extended_Image(Item,Width,Uppercase)

Returns the image associated with Item as per the Text_Io definition. The Text_Io definition states that given an enumeration literal, it will output the value of the enumeration literal (either an identifier or a character literal). The character case parameter is ignored for character literals. (LRM 14.3.9:9)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more that one parameter. The parameter Item must be an enumeration value. The image of an enumeration value is the corresponding identifier which may have character case and return string width specified.

Parameter Descriptions:

Item -- The user specifies the item that he wants the image of and passes it into the function. This parameter is required.

Width -- The user may specify the minimum number of characters to be in the string that is returned. If no width is specified then the default (0) is assumed. If the Width specified is larger than the image of Item, then the return string is padded with trailing spaces; if the Width specified is smaller than the image of Item then the default is assumed and the image of the enumeration value is output completely.

Uppercase -- The user may specify whether the returned string is in uppercase characters. In the case of an enumeration type where the enumeration literals are character literals, the Uppercase is ignored and the case specified by the type definition is taken. If no preference is specified then the default (true) is assumed.

ATTACHMENT B

APPENDIX F, Cont.

Examples:

Suppose the following types were declared:

```
Type X is (red, green, blue, purple);
Type Y is ('a', 'B', 'c', 'D');
```

Then the following would be true:

```
X'Extended_Image(red)              = "RED"
X'Extended_Image(red, 4)           = "RED "
X'Extended_Image(red,2)            = "RED"
X'Extended_Image(red,0,false)       = "red"
X'Extended_Image(red,10,false)      = "red       "
Y'Extended_Image('a')             = "'a'"
Y'Extended_Image('B')             = "'B'"
Y'Extended_Image('a',6)           = "'a'  "
Y'Extended_Image('a',0,true)       = "'a'"
```

'Extended_Value Attribute

Usage: X'Extended_Value(Item)

Returns the image associated with Item as per the Text_Io definition. The Text_Io definition states that it reads an enumeration value from the beginning of the given string and returns the value of the enumeration literal that corresponds to the sequence input. (LRM 14.3.9:11)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a CONSTRAINT_ERROR is raised.

ATTACHMENT B

APPENDIX F, Cont.

Parameter Descriptions:

Item -- The user passes to the function a parameter of the predefined type string. The type of the returned value is the base type of X.

Examples:

Suppose the following type was declared:

```
Type X is (red, green, blue, purple);
```

Then the following would be true:

```
X'Extended_Value("red")          = red
X'Extended_Value(" green")       = green
X'Extended_Value("   Purple")    = purple
X'Extended_Value(" GreEn ")      = green
```

'Extended_Width Attribute

Usage: X'Extended_Width

Returns the width for subtype of X.

For a prefix X that is a discrete type or subtype; this attribute is a function. This attribute yields the maximum image length over all values of the enumeration type or subtype X.

Parameter Descriptions:

There are no parameters to this function. This function returns the width of the largest (width) enumeration literal in the enumeration type specified by X.

ATTACHMENT B

APPENDIX F, Cont.

Examples:

Suppose the following types were declared:

```
Type X is (red, green, blue, purple);
Type Z is (X1, X12, X123, X1234);
```

Then the following would be true:

```
X'Extended_Width        = 6 -- "purple"
Z'Extended_Width        = 5 -- "X1234"
```

FLOATING POINT ATTRIBUTES

'Extended_Image Attribute

Usage: X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)

Returns the image associated with Item as per the *Text_Io definition.* The Text_Io definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative then a minus sign is included in the integer part of the value of Item. If Exp is 0 then the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more than one parameter. The parameter Item must be a Real value. The resulting string is without underlines or trailing spaces.

ATTACHMENT B

APPENDIX F, Cont.

Parameter Descriptions:

Item -- The user specifies the item that he wants the image of and passes it into the function. This parameter is required.

Fore -- The user may specify the minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, then leading spaces are output first to make up the difference. If no Fore is specified then the default (2) value is assumed.

Aft -- The user may specify the minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1 then Aft is one. If no Aft is specified then the default (X'Digits-1) is assumed. If based notation is specified the trailing '#' is included in aft.

Exp -- The user may specify the minimum number of digits in the exponent; the exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified then the default (3) is assumed. If Exp is 0 then no exponent is used.

Base -- The user may specify the base that the image is to be displayed in. If no base is specified then the default (10) is assumed.

Based -- The user may specify whether he wants the string returned to be in based notation or not. If no preference is specified then the default (false) is assumed.

Examples:

Suppose the following type was declared:

```
Type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true:

ATTACHMENT B

APPENDIX F, Cont.

```
X'Extended_Image(5.0)                = " 5.0000E+00"
X'Extended_Image(5.0,1)              = "5.0000E+00"
X'Extended_Image(-5.0,1)             = "-5.0000E+00"
X'Extended_Image(5.0,2,0)            = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)          = " 5.0"
X'Extended_Image(5.0,2,0,0,2)        = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)   = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)   = "2#1.1#E+02"
```

'Extended_Value Attribute

Usage: X'Extended_Value(Item)

Returns the value associated with Item as per the Text_Io definition. The Text_Io definition states that it skips any leading zeros, then reads a plus or minus sign if present then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a CONSTRAINT_ERROR is raised.

Parameter Descriptions:

Item – The user passes to the function a parameter of the predefined type string. The type of the returned value is the base type of the input string.

Examples:

Suppose the following type was declared:

Type X is digits 5 range -10.0 .. 16.0:

Then the following would be true:

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2#1.01#E2")    = 5.0
```

ATTACHMENT B

APPENDIX F, Cont.

'Extended_Digits Attribute

Usage: X'Extended_Digits(Base)

Returns the number of digits using base in the mantissa of model numbers of the subtype X.

Parameter Descriptions:

Base -- The user may specify the base that the subtype is defined in. If no base is specified then the default (10) is assumed.

Examples:

Suppose the following type was declared:

```
Type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true:

```
X'Extended_Digits                = 5
```

FIXED POINT ATTRIBUTES

'Extended_Image Attribute

Usage: X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)

Returns the image associated with Item as per the Text_Io definition. The Text_Io definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative then a minus sign is included in the integer part of the value of Item. If Exp is 0 then the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

ATTACHMENT B

APPENDIX F, Cont.

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more than one parameter. The parameter Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameter Descriptions:

Item – The user specifies the item that he wants the image of and passes it into the function. This parameter is required.

Fore – The user may specify the minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, then leading spaces are output first to make up the difference. If no Fore is specified then the default (2) value is assumed.

Aft – The user may specify the minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1 then Aft is one. If no Aft is specified then the default (X'Digits-1) is assumed. If based notation is specified the trailing '#' is included in aft.

Exp – The user may specify the minimum number of digits in the exponent; the exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified then the default (3) is assumed. If Exp is 0 then no exponent is used.

Base – The user may specify the base that the image is to be displayed in. If no base is specified then the default (10) is assumed.

Based – The user may specify whether he wants the string returned to be in based notation or not. If no preference is specified then the default (false) is assumed.

ATTACHMENT B

APPENDIX F, Cont.

Examples:

Suppose the following type was declared:

```
Type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true:

```
X'Extended_Image(5.0)              = " 5.00E+00"
X'Extended_Image(5.0,1)            = "5.00E+00"
X'Extended_Image(-5.0,1)           = "-5.00E+00"
X'Extended_Image(5.0,2,0)          = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)        = " 5.0"
X'Extended_Image(5.0,2,0,0,2)      = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)   = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)   = "2#1.1#E+02"
```

'Extended_Value Attribute

Usage: X'Extended_Value(Image)

Returns the value associated with Item as per the Text_Io definition. The Text_Io definition states that it skips any leading zeros, then reads a plus or minus sign if present then read the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a CONSTRAINT_ERROR is raised.

Parameter Descriptions:

Image -- The user passes to the function a parameter of the predefined type string. The type of the returned value is the base type of the input string.

ATTACHMENT B

APPENDIX F, Cont.

Examples:

Suppose the following type was declared:

```
Type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true:

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Fore Attribute

Usage: X'Extended_Fore(Base,Based)

Returns the minimum number of characters required for the integer part of the based representation of X.

Parameter Descriptions:

Base -- The user may specify the base that the subtype would be displayed in. If no base is specified then the default (10) is assumed.

Based -- The user may specify whether he wants the string returned to be in based notation or not. If no preference is specified then the default (false) is assumed.

ATTACHMENT B

APPENDIX F, Cont.

Examples:

Suppose the following type was declared:

```
Type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true:

```
X'Extended_Fore          = 3  -- "-10"
X'Extended_Fore(2)       = 6  -- " 10001"
```

'Extended_Aft Attribute

Usage: X'Extended_Aft(Base,Based)

Returns the minimum number of characters required for the fractional part of the based representation of X.

Parameter Descriptions:

Base -- The user may specify the base that the subtype would be displayed in. If no base is specified then the default (10) is assumed.

Based -- The user may specify whether he wants the string returned to be in based notation or not. If no preference is specified then the default (false) is assumed.

Examples:

Suppose the following type was declared:

```
Type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true:

```
X'Extended_Aft           = 1  -- "1" from 0.1
X'Extended_Aft(2)        = 4  -- "0001" from 2#0.0001#
```

ATTACHMENT B

APPENDIX F, Cont.

3. Specification of Package SYSTEM

```
PACKAGE System IS

TYPE Address is Access Integer;
TYPE Subprogram_Value is PRIVATE;

TYPE Name    IS (TELEGEN2);

System_Name  : CONSTANT name := TELEGEN2;

Storage_Unit : CONSTANT := 8;
Memory_Size  : CONSTANT := (2 ** 31) - 1;

-- System-Dependent Named Numbers:

Min_Int      : CONSTANT := -(2 ** 31);
Max_Int      : CONSTANT := (2 ** 31) - 1;
Max_Digits   : CONSTANT := 15;
Max_Mantissa : CONSTANT := 31;
Fine_Delta   : CONSTANT := 1.0 / (2 ** Max_Mantissa);
Tick         : CONSTANT := 10.0E-3;


-- Other System-Dependent Declarations

SUBTYPE Priority IS Integer RANGE 0 .. 63;

PRIVATE
  TYPE Subprogram_Value IS
    RECORD
      Proc_addr    : Address;
      Static_link  : Address;
    END RECORD;

END System;
```

ATTACHMENT B

APPENDIX F, Cont.

4. Restrictions on Representation Clauses

The Compiler supports the following representation clauses:

Length Clauses: for enumeration and derived integer types 'SIZE attribute (LRM 13.2(a))
Length Clauses: for access types 'STORAGE_SIZE attritube (LRM13.2(b))
Length Clauses: for tasks types 'STORAGE_SIZE attribute (LRM 13.2(c))
Length Clauses: for fixed point types 'SMALL attribute (LRM13.2(d))
Enumeration Clauses: for character and enumeration types other than boolean (LRM 13.3)
Record representation Clauses (LRM 13.4) with following constraints:
- Each component of the record must be specified with a component clause.
- The alignment of the record is restricted to mod 4, word (32 bit)aligned.
- Bits are ordered right to left within a byte.

Address Clauses: for objects, entries, and external subprograms (LRM 13.5(a)(c))

This compiler does NOT support the following representation clauses:

Enumeration Clauses: for boolean (LRM 13.3)

Address Clauses for packages, task units, or non-external Ada subprograms (LRM 13.5(b))

5. Implementation dependent naming conventions

There are no implementation-generated names denoting implementation dependent components.

6. Interpretation of Expressions in Address Clause

Expressions that appear in address specifications are interpreted as the first storage unit of the object.

ATTACHMENT B

APPENDIX F, Cont.

7. Restrictions on Unchecked Conversions

Unchecked conversions are allowed between any types or subtypes unless the target type is an unconstrained record or array type.

8. I/O Package Characteristics

The implementation provides limited support for text_io, direct_io and sequential_io. The user can perform operations only on standard input and standard output. Any attempt to handle permanent files will cause an exception to be raised .The name of the exception is dependent on the kind of the operation performed.
The only attempt to open or create a file must be done with the file name "console:" for outfile and "keyboard:" for infile and these correspond to standard output and standard input respectively.
Any attempt to use other file names will cause the exception NAME_ERROR to be raised.

Sequential_IO and Direct_IO cannot be instantiated for unconstrained array types or unconstrained types with discriminants without default values.

In TEXT_IO the type COUNT is defined as follows:

```
type COUNT is range 0 .. 2_147_483_645;
```

In TEXT_IO the subtype FIELD is defined as follows:

```
subtype FIELD is INTEGER range 0..1000;
```

According to the latest interpretation of the LRM, during a TEXT_IO.Get_Line call, if the buffer passed in has been filled, the call is completed and any succeeding characters and/or terminators (e.g., line, page, or end-of-file) will not be read. The first Get_Line call will read the line up to but not including the end-of-line mark, and the second Get_Line will read and skip the end-of-line mark left by the first read.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	199 * '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.	199 * '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.	100 * 'A' & '3' & 99 * '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.	100 * 'A' & '4' & 99 * '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.	197 * '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.	195 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 100 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 99 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	180 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_646
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2147483647
$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.	TELEGEN2
$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.	1000
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	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.	63
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BADCHAR * `/%
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/NONAME/DIRECTORY

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32768
$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.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	200
$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
$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.	"2:" & 195 * '0' & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 193 * '0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 198 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE_AVAILABLE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	TELEGEN2
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

Name and Meaning	Value
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2147483647
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	TELEGEN2
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

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

a. E28005C This test expects that the string "-- TOP OF PAGE. -- 62" 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 unitended 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

References and page numbers in this appendix are consistent with compiler documentation and not with this report.

A.1. Compiler Qualifiers

The compiler options are available as VMS command line qualifiers. The general command format of the Ada compiler is:

$ ADA386/COMPILE[{<qualifier>}] <file_spec>{,<file_spec>}

where:

<qualifier> is one of the qualifiers available for the compiler.

<file_spec> is one in a possible series of file specifications, separated by commas, indicating the unit(s) to be compiled. If /INPUT_LIST is used, <file_spec> is interpreted as a file containing a list of files to be compiled. The default source file extension is ".ADA", and the default list file extension is ".LIS." A file name may be qualified with a device and/or directory name in standard VMS format. A file may reside on any directory in the system.

The default qualifier settings should be appropriate for most applications. The following table presents an alphabetical list of the qualifiers, their actions, and their defaults.

Qualifier Name	Action	Default
/ABORT_COUNT=<value>	Specify maximum errors/warnings.	999
/[NO]BIND [=<main_unit>]	/BIND runs the Binder on unit being compiled or on unit specified.	/NOBIND
/CONTEXT =<value>	Request <value> context lines around each error in error listing.	1
/[NO]DEBUG	/DEBUG enables generation of information for the Source Level Debugger.	/NODEBUG
/INPUT_LIST	Input file contains names of files to be compiled, not Ada source.	File contains Ada source.
/LIBFILE=<file_spec>	Specify name of library file.	LIBLST.ALB
/[NO]LIST [=<file_spec>]	/LIST creates listing file. Default name is <source_file_name>.LIS, or <file_spec>.LIS, if specified.	/NOLIST
/[NO]MACHINE_CODE [=<file_spec>]	/MACHINE_CODE requests macro assembly listing, which is sent to <comp_unit>.ASM or to <file_spec>, if specified.	/NOMACHINE_CODE
/[NO]MIXED	/NOMIXED with /LIST causes errors to be output following source listing.	/MIXED, errors and source intermixed.
/[NO]MONITOR	/MONITOR requests progress messages.	/NOMONITOR
/[NO]OBJECT	/NOOBJECT restricts compilation to syntactic and semantic analysis.	/OBJECT
/[NO]OFFSET	/OFFSET includes hex offsets in an assembly listing.	/NOOFFSET
/[NO]OPTIMIZE [=(<option>{,<option>})| <qualifier>]	/OPTIMIZE causes Optimizer to be run on unit(s) being compiled.	/NOOPTIMIZE
/[NO]PROCEED	/NOPROCEED causes prompts to be issued after each error.	/PROCEED (batch) /NOPROCEED (interactive)
/[NO]SQUEEZE	/SQUEEZE deletes unneeded intermediate unit information after compilation.	/NOSQUEEZE if /DEBUG or /NOOBJECT; /SQUEEZE otherwise.
/[NO]SUPPRESS [=<option>{,<option>}]	/SUPPRESS suppresses selected run-time checks and/or source line references in generated object code.	/NOSUPPRESS
/TEMPLIB =(<sublib>{,<sublib>})	Specify a temporary library containing listed sublibraries.	None.

/SUPPRESS Qualifier Options.

Option	Action
ALL	Suppress source line information and all run-time checks listed below.
NONE	Equivalent to /NOSUPPRESS
SOURCE_INFO	Suppress source line information in the object code.
ALL_CHECKS	Suppress all access checks, discriminant checks, division checks, elaboration checks, index checks, length checks, overflow checks, range checks, and storage checks.
ELABORATION_CHECK	Suppress all elaboration checks.
OVERFLOW_CHECK	Suppress all overflow checks.
STORAGE_CHECK	Suppress all storage checks.

A.3. Binder Qualifiers

The general command format of the bind step is:

$ ADA386/BIND{<qualifier>} <main_unit_name>

where:

<qualifier> can be none or more Binder qualifiers.

<main_unit_name> indicates the name of the unit to be used as the main program.

The Binder has the following qualifiers:

Qualifier Name	Action	Default
/LIBFILE =<file_spec>	Specify name of library file.	LIBLST.ALB
/NO/MONITOR	/MONITOR requests progress messages.	/NOMONITOR
/TEMPLIB =(<sublib>{,<sublib>})	Temporary list of sublibraries.	None.
/TRACEBACK =<#_levels>	Set the depth of exception traceback report.	5 levels.

A.4. Linker Qualifiers and Options

The VMS command line for the Ada Linker is:

$ ADA386/LINK{<qualifier_name>} {<compilation_unit_name>}

where:

<qualifier_name> is none or more of the command line qualifiers listed in the following table:

Linker Command Line Qualifiers

Qualifier	Action	Default
/BASE =<address>	Specify start location.	0
/NO[DEBUG]	Output debug information.	/NODEBUG
/EXECUTE_FORM	Choose EF load module format.	/EXECUTE_FORM
/HEX86	Choose Intel 80386 hexadecimal load module format.	None.
/LIBFILE =<file_spec>	Specify name of library file.	LIBLST.ALB
/LOAD_MODULE =<file_spec>	Specify load module output.	/LOAD_MODULE
/NO[MAP] =<file_spec> /[NO]EXCLUDED /[NO]IMAGE /LINES_PER_PAGE =<value> (>10) /[NO]LOCALS /WIDTH=<132 80>	Control output of a link map.	/NOMAP /NOEXCLUDED /NOIMAGE 50 /NOLOCALS 132
OBJECT_FORM =<library_component_name>	Choose linked OF module output.	None.
/NO[OPTIONS] =<file_spec>	Designate options file.	/NOOPTIONS
TEMPLIB =(<sublib>{,<sublib>})	Temporary list of sublibraries.	None.

If the /EXECUTE_FORM or /HEX86 qualifier is used, it must be specified immediately after the /LINK command specification.

The /OPTIONS command line qualifier is used to specify that additional Linker options are to be taken from a file. The available Linker options are summarized in the following table. Defaults are in italics.

Linker Options File Options

DEFINE /<symbol_name>=<value> /ADDRESS	-- Specify link-time values for symbols.
EXIT	-- Terminate options list.
INPUT /MAIN	/*SPEC*	/BODY	/OFM /EXPORT_DEFINITIONS /PHANTOM	/WORKING_SUBLIB /NOSEARCH	<library_component_name>	-- Identify object modules to be linked and -- specify the search path.
LOCATE /CONTROL_SECTION=CODE	DATA	CONSTANT /COMPONENT_NAME=<library_component_name> /*SPEC*	/BODY	/OFM /AT=<address> /IN=<region_name> /AFTER=<control_section_name	library_component_name> /ALIGNMENT=<value>	-- Specify addresses for control sections.
MAP /*NO*IMAGE /*NO*LOCALS /*NO*EXCLUDED /WIDTH=<*132*	80> /LINES_PER_PAGE=<value>	(*50*)	<file_spec>	-- Control link map generation.
OUTPUT /*COMPLETE*	/INCOMPLETE /*LOAD_MODULE*=<file_spec> /OBJECT_FORM=<library_component_name>	-- Specify complete or incomplete output -- and its format.
QUIT	-- Abandon link operation.
REGION /LOW_BOUND=<address> /HIGH_BOUND=<address> /UNUSED	<region_name>	-- Define and name memory regions.