UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A215 178

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:** Proprietary Software Systems, Inc., PSS Ada Compiler VAX/VMS, VAX/VMS 8350 (Host & Target), 890710I1.10120	5. TYPE OF REPORT & PERIOD COVERED 10 July 1989 to 10 July 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)

Proprietary Software Systems, Inc., PSS Ada Compiler VAX/VMS, Ottobrunn, West Germany, VAX 8350 under VMS, Version 4.7 (Host & Target), ACVC 1.10.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

89 11 30 056

Ada Compiler Validation Summary Report:

Compiler Name: PSS Ada Compiler VAX/VMS, Version TV-01.000

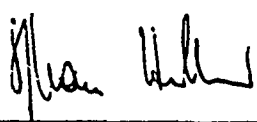
Certificate Number: #890710I1.10120

Host: VAX 8350 under VMS Version 4.7

Target: VAX 8350 under VMS Version 4.7

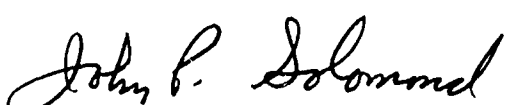
Testing Completed 10th July 1989 Using ACVC 1.10

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-032

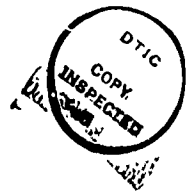
Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #890710I1.10120
Proprietary Software Systems, Inc.
PSS Ada Compiler VAX/VMS
VAX/VMS 8350 Host and Target

Completion of On-Site Testing:
10th July 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 10th July 1989 at IABG mbH, Ottobrunn.

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: PSS Ada Compiler VAX/VMS, Version TV-01.000

ACVC Version: 1.10

Certificate Number: #890710I1.10120

Host Computer:

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

Target Computer:

Machine:	VAX 8350
Operating System:	VMS Version 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 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 and two dimensional array types and no exception for one dimensional array and two 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 not supported for functions or 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 instatiate 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 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) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

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

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

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

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

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

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

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

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

12) Temporary text files are given names and not deleted when closed. (See test CE3112A.)

13) Only one internal file can be associated with each external file for sequential files. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

14) Only one internal file can be associated with each external file for direct files. (See tests CE2107F..H (3 tests), CE2110D and CE2111H.)

15) More than one internal file can be associated with each external file for text files when reading only. (See tests CE3111A..E (5 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 482 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 84 tests were required to successfully demonstrate the test objective. (See section 3.6.)

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

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	127	1131	1850	17	22	44	3191
Inapplicable	2	7	465	0	6	2	482
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	496	245	172	99	160	332	127	36	252	257	276	3191
N/A	20	102	184	3	0	0	6	0	10	0	0	112	45	482
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, 482 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 innapplicable because they require a value of MAX_MANTISSA greater than 32.

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

f. 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.

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

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

i. CA2009A, CA2009C, CA2009F and CA2009D are not applicable because this compiler creates dependancies between generic bodies, and units that instantiate them (see section 2.2i for rules and restrictions concerning generics).

j. LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F are not applicable because this implementation does not support pragma INLINE when applied across compilation units (See Appendix F of the Ada Standard in Appendix B of this report, and Section 2.2.h (1)).

k. 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.

l. CD2A61I is not applicable because this implementation imposes restrictions on 'SIZE length clauses for array types.

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

n. CD2A91A..E (5 tests) are not applicable because 'SIZE length clauses for task types are not supported.

o. CD2B11G is not applicable because 'STORAGE_SIZE representation clauses are not supported for access types where the designated type is a task type.

p. CD2B15B is not applicable because a collection size larger than the size specified was allocated.

q. The following 76 tests are not applicable because, for this implementation, address clauses are not implemented:

CD5003B..I (8 tests)		CD5011A	CD5011B	CD5011C
CD5011D	CD5001E	CD5011F	CD5011G	CD5011H
CD5011I	CD5011K	CD5011L	CD5011M	CD5011N
CD5011Q	CD5011R	CD5011S	CD5012A	CD5012B
CD5012C	CD5012D	CD5012E	CD5012F	CD5012G
CD5012H	CD5012I	CD5012J	CD5012L	CD5012M
CD5013A	CD5013B	CD5013C	CD5013D	CD5013E
CD5013F	CD5013G	CD5013H	CD5013I	CD5013K
CD5013L	CD5013M	CD5013N	CD5013O	CD5013R
CD5013S	CD5014A	CD5014B	CD5014C	CD5014D
CD5014E	CD5014F	CD5014G	CD5014H	CD5014I
CD5014J	CD5014K	CD5014L	CD5014M	CD5014N
CD5014O	CD5014R	CD5014S	CD5014T	CD5014U
CD5014V	CD5014W	CD5014X	CD5014Y	CD5014Z

r. 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.

s. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

aj. CE2107F..H (3 tests), CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file for direct files. The proper exception is raised when multiple access is attempted.

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

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

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

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

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

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

aq. CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 84 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	B41202A	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	B95032A	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 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 imformation 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 modified demonstrate the correct processing of pragma LIST.

f. Tests C45524A and C45524B contain a check at line 136 that may legitimately fail as repeated division may produce a quotient that lies within the smallest safe interval. This check was modified to include, after line 138, the text:

```
ELSIF VAL <= F'SAFE_SMALL THEN COMMENT ("UNDERFLOW SEEMS GRADUAL");
```

For this implementation, the required support package specification, SPPRT13SP, was rewritten to provide constant values for the function names.

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 PSS Ada Compiler VAX/VMS 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 PSS Ada Compiler VAX/VMS 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 computer:	VAX 8350
Host operating system:	VMS Version 4.7
Target computer:	VAX 8350
Target operating system:	VMS Version 4.7
Compiler:	PSS Ada Compiler VAX/VMS, Version TV-01.000

The original ACVC distribution tape was loaded to the host machine, where it was customized to remove all withdrawn tests and tests requiring unsup-

ported floating point precisions. Tests that make use of implementation specific values were also customized. Tests requiring modifications during the prevalidation testing were modified accordingly.

After the test files were loaded to disk, the full set of tests was compiled linked, and all executable tests were run on the VAX 8350. Results were evaluated and printed on the host machine.

The compiler was tested using command scripts provided by Proprietary Software Systems and reviewed by the validation team. The compiler was tested using no option qualifiers. All chapter B tests were compiled with the /LIST qualifier.

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 magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

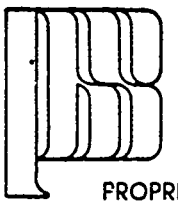
3.7.3 Test Site

Testing was conducted at IABG mbH, Ottobrunn and was completed on 10th July 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Proprietary Software Systems has submitted the following Declaration of Conformance concerning the PSS Ada Compiler VAX/VMS.



PROPRIETARY SOFTWARE SYSTEMS

DECLARATION OF CONFORMANCE
L001-1739

Compiler Implementor: PROPRIETARY SOFTWARE SYSTEMS, INC.
Ada Validation Facility: IABG mbH, Abt SZT.
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name:	PSS Ada Compiler VAX/VMS Version TV-01.000
Host Architecture:	VAX 8350
Host OS and Version:	VMS Version 4.7
Target Architecture:	VAX 8350
Target OS and Version:	VMS Version 4.7

Implementor's Declaration

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

Joel E. Fleiss Date: 14 September 1989

PROPRIETARY SOFTWARE SYSTEMS, INC.
Joel E. Fleiss

Owner's Declaration

I, the undersigned, representing PROPRIETARY SOFTWARE SYSTEMS, INC., take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compiler(s) listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Joel E. Fleiss Date: 14 September 1989

PROPRIETARY SOFTWARE SYSTEMS, INC.
Joel E. Fleiss

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 PSS Ada Compiler Version TV-01.000, as described in this Appendix, are provided by Proprietary Software Systems. 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 contained in Appendix F.

MIL-STD-1815A APPENDIX F

This section discusses how the PSS Ada Compiler handles MIL-STD-1815A issues that are left up to the implementor.

Supported Pragmas

The PSS Ada Compiler supports the following pragmas:

ELABORATE

This pragma controls elaboration order. It specifies that the named library unit must be elaborated before the following compilation unit. The pragma is allowed only immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument to the pragma must be the simple name of a library unit mentioned by the context clause.

An elaboration order that meets the rules of Ada may not satisfy the needs of some applications. In such cases, the user will specify the elaboration order via the pragma ELABORATE. In particular, a statement such as

```
PRAGMA ELABORATE (INITIALIZE);
```

may be used to cause an initialization package or procedure to be elaborated before all other units.

PRIORITY

This pragma specifies the priority of a task or the priority of a main program. It specifies the priority as a static expression of the predefined integer subtype PRIORITY which has a range of 10 to 200. The pragma is allowed within the specification of a task unit or immediately within the outermost declarative part of a main program.

Priority affects the order of task execution. The scheduler selects the task with the highest priority for execution. New tasks that are ready to execute are placed in a priority-ordered queue with tasks of equal priority being placed in time-arrival order within the same priority level. The following statement in a task sets its priority to 10.

```
PRAGMA PRIORITY (10);
```

The scheduler will select another waiting task to start execution if the executing task becomes blocked or when a higher priority task becomes ready. In other words, a higher priority task will preempt a lower priority task. All tasks which do not have a specified priority have a default priority of 10.

PACK

The pack pragma tells the compiler to use the minimum amount of storage for the named record or array type. This pragma will be ignored if a representation clause is specified for the type. The placement of the pragma is the same as that for representation clauses. Usually the pragma is placed immediately after the type declaration. The following statements declare a type named word to be a packed array of booleans. Since the array has a range of 0..31, the pack pragma has the effect of packing the array type into one 32 bit word for the VAX target.

```
TYPE WORD IS ARRAY (0..31) OF BOOLEAN;
PRAGMA PACK (WORD);
```

LIST

The list pragma controls the production of a compilation listing. The list (on) pragma turns on the listing and the list (off) pragma turns off the listing until a list (on) pragma is encountered. In the presence of compiler generated diagnostic messages, this pragma is ignored and the full listing is produced. An example of the list pragma follows.

```
PRAGMA LIST (OFF);
```

PAGE

The page pragma specifies that the source text that follows the page pragma is to start at the top of a new page in the compilation listing. In the presence of compiler generated diagnostic messages, this pragma is ignored. An example of the page pragma follows.

```
PRAGMA PAGE;
```

LINKAGE_NAME

This pragma effects a link-time binding between an Ada entity and an externally meaningful name. The format is:

**PRAGMA LINKAGE_NAME (Ada-simple-name,
string-constant);**

where the Ada-simple-name is the name of a subprogram, exception, or object. The Ada-simple-name must be declared in a package specification and the pragma must appear in the same package specification, after the declaration. The string-constant is a name that is not defined within a compilation unit in the user's library, but rather an external name to be supplied to the link editor.

The effect of pragma LINKAGE_NAME is to provide a specified external name for an Ada entity, allowing the link editor to associate the entity with a symbol (string-constant) that is known to the link editor but not to the user's library units. The PSS Ada Compiler will not check the string-constant supplied by the user as the external name; it is the user's responsibility to ensure that the string-constant is acceptable to the link editor and meaningful to the program.

FOREIGN_BODY

This pragma informs the PSS Ada Compiler that the body of a package, including all subprograms, objects, and exceptions, is implemented in an externally compiled module. The external module may be written in Ada (and compiled into another library) or in another language.

The package containing pragma FOREIGN_BODY must be a non-generic top-level package specification. It may contain only the following: subprogram declarations, object declarations, number declarations, and pragmas. Object declarations must use an unconstrained type mark that is not a task type, and cannot use an initial value expression. The foreign body itself is responsible for initialization of all objects declared in the package, including objects that are normally initialized implicitly (such as access types and certain record types, as described in [LRM 3.2.1]). The FOREIGN_BODY pragma must appear before any declarations. The format is:

PRAGMA FOREIGN_BODY (language_name,
elaboration_routine_name);

where language_name is one of the following: JOVIAL, Modula-2, C, FORTRAN, Pascal, COBOL, or assembler. The language_name informs the PSS Ada Compiler which subroutine linkage will be used by the foreign module. The foreign module may include a routine for initialization, which is identified by the optional parameter elaboration_routine_name. It is the user's responsibility to ensure that foreign modules use data representations, calling conventions, and (optionally) initialization routines that are compatible with the PSS Ada Compiler and with the Ada language itself.

When using pragma FOREIGN_BODY, the user should include a LINKAGE_NAME pragma for each declaration in the package, including declarations in nested package specifications. This will give the user positive control over external names used by the foreign module and ensure that no naming conflicts occur at link time.

Appendix E gives a complete example of a program that uses the pragmas LINKAGE_NAME and FOREIGN_BODY in order to access several functions in the VAX Run-time library. LINKAGE_NAME associates the names of the VAX internal names and the Ada names. FOREIGN_BODY_body specifies the language. The ADALIB command:

```
$ ADALIB INSERT/OLB STARLET -
        SYS$ROOT:[SYSLIB]STARLET.OLB
```

will allow the user to link with procedures and functions in the VAX Run-time library.

Unsupported Pragmas The other predefined pragmas in the language currently have no effect. However the same functionality is provided for some pragmas by other means. These pragmas are the following:

CONTROLLED There is no automatic storage deallocation of access collections. If you want to deallocate, use the standard generic function for deallocation: UNCHECKED_DEALLOCATION.

INLINE The INLINE pragma is not supported in this version. It will be supported in subsequent versions.

INTERFACE Instead of the INTERFACE pragma, the PSS Ada Compiler uses the combination of the LINKAGE_NAME and FOREIGN_BODY pragmas that are implementation defined.

MEMORY_SIZE The MEMORY_SIZE pragma is not supported for the VAX target.

OPTIMIZE The OPTIMIZE option is used to control optimization rather than the OPTIMIZE pragma.

SHARED The SHARED pragma is not supported for the VAX target.

STORAGE_UNIT The STORAGE_UNIT pragma is not supported for the VAX target.

SUPPRESS The SUPPRESS option is used to control suppression of exceptions rather than the SUPPRESS pragma.

SYSTEM_NAME The SYSTEM_NAME pragma is not supported for the VAX target. Instead, the assigned name is VAX.

Attributes Aside from restrictions on certain representation specifications (see the Restrictions section that follows), the PSS Ada Compiler supports no implementation-dependent attributes.

Package "SYSTEM"

The predefined package called SYSTEM contains the definitions of certain implementation dependent characteristics. In accordance with Section 13.7 of the *Ada Language Reference Manual*, the package is defined as follows:

```
package SYSTEM is
--Required definitions:
   type ADDRESS is new integer;
   type NAME is (VAX);
   SYSTEM_NAME    : constant NAME    := VAX;
   STORAGE_UNIT   : constant         := 8;
   MEMORY_SIZE    : constant         := 1000000;
   MAX_INT        : constant         := 2**32-1;
   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'FIRST;
   RUNTIME_ERROR      : exception;
end SYSTEM;
```

Restrictions

Representation clauses are used to map Ada types onto the target machine. The PSS Ada Compiler implements all representation clauses defined in Chapter 13 of the LRM except for address clauses [LRM 13.5]. Support is provided for length clauses [LRM 13.2], enumeration representation clauses [LRM 13.3], and record representation clauses [LRM 13.4], with the following restrictions:

- Length clauses for size specifications (T'SIZE) are restricted to types and subtypes whose sizes are known at compile time.

- Length clauses for size specifications (T'SIZE) are meaningless for floating point types, access types, and task types. The user can assign the predefined sizes of these types. Other sizes will result in a diagnostic message.

- Length clauses for size specifications (T'SIZE) for discrete types can not exceed the largest size of any predefined discrete type.

- Length clauses cannot be used for composite types to force a smaller size on components than is established by length clauses for the componen types or by the default types of the components.

- Length clauses for the attributes T'STORAGE_SIZE and T'SMALL are restricted only as specified in the [LRM 13.2]. Note that the PSS Ada Compiler will include a small amount of extra storage for administrative purposes in storage sizes for tasks and access types. If a length clause results in a T'STORAGE_SIZE of 0, the exception STORAGE_ERROR will be raised upon allocation of an object of that type T. A length clause for T'STORAGE_SIZE is not available for access types that designate a task type.

- The value of T'SMALL in a fixed point length clause must be a power of 2 and must be available in a predefined type.

- For enumeration representation clauses, the integer codes given in the aggregate must be in the range INTEGER'FIRST..INTEGER'LAST.

- Record representation clauses may be used only on types whose components' sizes are known at compile time. Care must be taken to ensure that record representation clauses map to predefined type boundaries, otherwise severe run-time penalties may be observed.

- If representation clauses are given for some (but not all) components of a record, the PSS Ada Compiler will allocate the unspecified components as it sees fit.

- Address clauses are not permitted.

- The PSS Ada Compiler represents integer and fixed point types and subtypes in VAX native form; that is, as two's complement numbers. As a consequence, it is an error to specify a length clause of 1 bit for the integer range 100..101. Even though 1 bit is sufficient to represent these two values, the PSS Ada Compiler will allocate 7 bits because 2**6 < 101 < 2**7 and reject a length clause which specifies fewer than 7 bits for such a range.

Names

The PSS Ada Compiler generates implementation-dependent components for arrays with bounds dependent on record discriminants. These components, which are used for book-keeping by the PSS Ada Compiler cannot be accessed by the user.

Address Clauses

The PSS Ada Compiler does not support address clauses.Ada semantics of address clauses allow for the association of numbered interrupts with task entries. The PSS Ada Compiler implements this association by interpreting the simple_expression in the clause as the vector number of a VAX exception or interrupt. These exceptions and interrupts are described in the VAX Programmer's Reference Manual.

Unchecked Conversions

The generic function UNCHECKED_CONVERSION can be instantiated to effect an unchecked type conversion. The only restriction imposed by the PSS Ada Compiler is that the sizes of the source and target types must be known at compile time. Unchecked conversions between types of unequal sizes will result in truncation or zero-padding, as appropriate. Unconstrained arrays and unconstrained record types without defaulted discriminants are not allowed as target types of unchecked conversions.

Input/Output Packages

Predefined packages for input and output are provided with the PSS Ada Compiler. These packages include SEQUENTIAL_IO, DIRECT_IO, TEXT_IO, IO_EXCEPTIONS, and LOW_LEVEL_IO, as described in Chapter 14 of the Ada Language Reference Manual. All input and output operations are supported except for SEQUENTIAL_IO and DIRECT_IO operations on unconstrained array types.

Additional Information

Generics

The PSS Ada Compiler allows a generic declaration to be compiled separately from its corresponding proper body. It also permits separate compilation of subunits of a generic unit [LRM 10.3]. The PSS Ada Compiler enforces the requirement that a generic body must be compiled prior to an instantiation of the generic unit. When recompiling the body of a generic unit, the PSS Ada Librarian will mark as obsolete all units that instantiated the generic.

Every instantiation of a user-defined generic will result in the generation of in-line code for the generic unit. Thus, multiple instantiations of a given generic will produce duplications of code. Instantiations of the predefined generics UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION are implemented as calls to runtime support routines.

Main Programs

When linking an Ada program, one of the library units must be designated as the main program . The main program must be a subprogram library unit with no parameters [LRM 10.1].

Predefined Types

The PSS Ada Compiler has the predefined numeric types of INTEGER, FLOAT, and LONG_FLOAT. The predefined types for COUNT, POSITIVE_COUNT and FIELD are found in the input/output packages TEXT_IO and DIRECT_IO. The predefined type DURATION is found in the package CALENDAR. The attributes for each of these types are listed in the following table.

Fixed Point Types

The maximum fixed point accuracy on the VAX is 2**-31.

Attributes of Predefined Types

TYPE	ATTRIBUTE	VALUE
INTEGER	FIRST	-2**31
INTEGER	LAST	2**31 - 1
FLOAT	DIGITS	6
FLOAT	MANTISSA	21
FLOAT	EMAX	84
FLOAT	EPSILON	16#0.1000_00#E-4
	approximately	9.53674E-07
FLOAT	SMALL	16#0.8000_000#E-21
	approximately	2.54894E-26
FLOAT	LARGE	16#0.FFFF_F80#E+21
	approximately	1.93428E+25
FLOAT	FIRST	-16#0.7FFF_FF8#E+32
	approximately	-1.70141E+38
FLOAT	LAST	16#0.7FFF_FF8#E+32
	approximately	1.70141E+38
FLOAT	SAFE_EMAX	127
FLOAT	SAFE_SMALL	16#0.1000_000#E-31
	approximately	2.93874E-39
FLOAT	SAFE_LARGE	16#0.7FFF_FC0#E+32
	approximately	1.70141E+38
FLOAT	MACHINE_RADIX	2
FLOAT	MACHINE_MANTISSA	24
FLOAT	MACHINE_EMAX	127
FLOAT	MACHINE_EMIN	-127
FLOAT	MACHINE_ROUNDS	TRUE
FLOAT	MACHINE_OVERFLOWS	TRUE

Attributes of Predefined Types

TYPE	ATTRIBUTE	VALUE
LONG_FLOAT	DIGITS	9
LONG_FLOAT	MANTISSA	31
LONG_FLOAT	EMAX	124
LONG_FLOAT	EPSILON	16#0.4000_0000_0000_000#E-7
	approximately	9.3132257461548E-31
LONG_FLOAT	SMALL	16#0.8000_0000_0000_000#E-31
	approximately	2.3509887416446E-38
LONG_FLOAT	LARGE	16#0.FFFF_FFFF_0000_000#E+31
	approximately	2.1267647922655E+37
LONG_FLOAT	FIRST	-16#0.7FFF_FFFF_FFFF_FF8#E+32
	approximately	-1.7014118346047E+38
LONG_FLOAT	LAST	16#0.7FFF_FF8#E+32
	approximately	1.7014118346047E+38
LONG_FLOAT	SAFE_EMAX	127
LONG_FLOAT	SAFE_SMALL	16#0.1000_0000_0000_000#E-31
	approximately	2.9387358770557E-39
LONG_FLOAT	SAFE_LARGE	16#0.7FFF_FFFF_0000_000#E+32
	approximately	1.7014118338124E+38
LONG_FLOAT	MACHINE_RADIX	2
LONG_FLOAT	MACHINE_MANTISSA	56
LONG_FLOAT	MACHINE_EMAX	127
LONG_FLOAT	MACHINE_EMIN	-127
LONG_FLOAT	MACHINE_ROUNDS	TRUE
LONG_FLOAT	MACHINE_OVERFLOWS	TRUE
DURATION	DELTA	0.0001
DURATION	SMALL	2#1.0#E-14
		0.6103515625E-4
DURATION	FIRST	-86400.0
DURATION	LAST	86400.0

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 preceeding 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.	2_147_483_646
$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_FLOAT_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.	90_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	33_554_433.0
$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.	BAD.BAD.BAD
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	ANOTHER.BAD.BAD

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.	-90_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-33_554_433.0
$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
$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:" & 235 * '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:" & 233 * '0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 238 * '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_INTEGER_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	VAX
$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.	1_000_000
$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.	VAX
$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.	0.01667

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 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, 85 & 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 99)

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

This appendix contains information of the compiler and linker options used in this validation.

The PSS Ada Compiler

The PSS Ada Compiler translates a single source file and places the compilation results (object modules, date and time stamps, symbol information) in the Ada program library. Invocation of the PSS Ada Compiler must be from the directory where the PSS Ada library resides. However, the source file that is used as input to the PSS Ada Compiler may reside in a different directory.

Ada Compiler Files

The PSS Ada Compiler creates and uses several files during the course of developing and maintaining Ada programs. Each of these files has the name of the compilation unit with an extension indicating the purpose of the file. The PSS Ada Compiler generates files with the following extensions:

.BOD Contains the representation of the body of a generic and the visibility information available to subunits. The PSS Ada Compiler reads this file when compiling a unit that instantiates a generic or is a subunit of another unit.

.DI Contains the representation of a unit specification. The PSS Ada Compiler reads this file when compiling a unit that includes a "with" clause.

.LIS Contains the listing of the source interspersed with any error and warning messages produced by the PSS Ada Compiler.

.MLS Contains the machine language listing of the generated code.

.OBJ Contains the object code for an Ada unit body.

.SOBJ Contains object code for an Ada unit specification.

The PSS Ada Librarian controls all of these files except the .LIS and .MLS files. The user should not use these extensions for any other purpose, nor should any of these files be deleted by the user while the corresponding unit is still active in the Program Library. To avoid unpredictable and erroneous results,

do not delete, edit, rename or otherwise modify these files via VMS commands. Instead use the PSS Ada Librarian commands as described in the PSS Ada Library section to perform operations on library files.

Several temporary files are created during compilation. Some of these files have unique names composed of hexadecimal numbers concatenated with the extensions listed above. Other temporary files have extensions .TER, .TEB, .TSB, and .TSE. Any of these temporary files that appear in the user's directory as a result of an abnormally terminated compilation (or a system failure) should be deleted by the user.

Invoking the Compiler

The PSS Ada Compiler is an executable program under VMS. To use the PSS Ada Compiler you give an invocation command that specifies the name of an Ada source code file. Normally the PSS Ada Compiler invocation command is available system-wide. This is done at installation time by the System Manager, in the same manner that commonly used commands are usually defined in a VAX development environment. The syntax of the PSS Ada Compiler invocation command is:

$ ADA/qualifier(s) file_specification/qualifier(s)

where

/qualifier(s) specifies the compilation options to be used

file_specification specifies the name of the file containing the Ada source code to be compiled. This may be any legal VMS file specification, including a logical name.

The Ada command is not order or column dependent. You may use spaces between any of the parts of the command and you may place the qualifiers after the Ada command or after the file name.

Following are examples of equivalent invocation commands to compile a module named "source_file" with options to suppress constraint checks and to produce a machine code listing:

```
$ ADA/SUPPRESS=CONSTRAINT_CHECKS/LIST=MACHINE source_file

$ ADA source_file/SUPPRESS=CONSTRAINT_CHECKS/LIST=MACHINE

$ ADA source_file /SUPPRESS=CONSTRAINT_CHECKS/LIST=MACHINE

$ ADA/SUPPRESS=CONSTRAINT_CHECKS source_file/LIST=MACHINE

$ ADA /SUPPRESS=CONSTRAINT_CHECKS source_file /LIST=MACHINE
```

You may also omit much of the text in commands. As long as a command option is unique it can be truncated. For example, the above commands could be simplified to:

```
$ ADA /SUP=C/LIS=M source_file
```

Also, if you use the qualifier .ADA for our Ada source file name, this qualifier may be omitted when entering the source file name. For example

```
$ ADA/SUP=C/LIS=M HELLO
```

will compile the source file HELLO.ADA.

The PSS Ada Compiler accepts a maximum command string length of 240 characters. Command strings may continue on multiple lines by using the continuation character, a hyphen "-", as the last element on each line that is to be continued. Command line continuation can be useful when entering a very long command line, or when placing an Ada compilation command in a command procedure file.

Compilation Options

The PSS Ada Compiler has a variety of options that may be chosen by using qualifiers in the compiler command line. The text that follows describes each qualifier in detail. All qualifiers but the /LINES qualifier make use of the prefix "NO" to effect the negative form of the option. For example, /NOCONSTRAINT_CHECKS suppresses the generation of constraint checks.

Some qualifiers are incompatible with certain other qualifiers. For example, /LINES=20 is incompatible with /NOLIST. The PSS Ada Compiler will handle incompatible qualifiers by accepting the first valid qualifier and ignoring later, inconsistent qualifiers.

Since the compilation command line is limited to 240 characters, abbreviation of qualifiers can help fit a compilation command onto one line. When command length is not a factor, it is better to spell out the qualifier names in full to yield more readable commands, particularly in command procedures.

The following table summarizes the PSS Ada Compiler command qualifiers, including applicable qualifier values, defaults, and incompatible qualifiers.

Ada Command Line Qualifiers

Qualifer <qualifer values>	Default Qualifiers	Incompatible Qualifiers
/<NO>LIST=(SOURCE|ERRORS|MACHINE|ALL)	/LIST=ERRORS	/NOLIST when /LIST
/<NO>OPTIMIZATION	/OPTIMIZATION	
/<NO>SUPPRESS=(CONSTRAINT_CHECKS| STACK_CHECKS| ELABORATION_CHECKS|ALL)	/NOSUPPRESS	

The following paragraphs show the positive and negative forms of each qualifier for the PSS Ada Compiler command. The default forms are indicated by "(D)".

/LIST=SOURCE
/LIST=ERRORS (D)
/LIST=MACHINE
/LIST=ALL

/LIST=SOURCE produces a listing of the source text with line numbers prefixed to each source line. The list file produced has the same name as the source file but with a file type of .LIS.

/LIST=ERRORS produces a listing of the source text as described for **/LIST=SOURCE** but only in the event that some error is detected by the PSS Ada Compiler. Since **/LIST=ERRORS** is the default condition, a compilation that has errors will generate a listing, but an error-free compilation will not generate a listing unless a listing has been specifically requested.

/LIST=MACHINE produces a source file of the machine code generated by the PSS Ada Compiler in a format similar to the listing output of the VAX VMS Macro Assembler including both the generated assembly code and the hexadecimal representation. The listing file that is produced has the same name as the source file but with a file type .MLS.

/LIST=ALL is the same as **/LIST=(SOURCE,MACHINE)**.

Listing options may be combined. Examples of some listing options are:

/LIST
/LIST=SOURCE
/LIST=ERRORS
/LIST=MACHINE
/LIST=(SOURCE,MACHINE)
/LIST=(ERRORS,MACHINE)
/NOLIST

/OPTIMIZE (D)
/NOOPTIMIZE

/OPTIMIZE causes the PSS Ada Compiler to produce optimized code. This takes the place of the pragma for optimization. The PSS Ada Compiler produces code that has been optimized for both time and space.

/SUPPRESS=CONSTRAINT_CHECKS
/SUPPRESS=ELABORATION_CHECKS
/SUPPRESS=STACK_CHECKS
/SUPPRESS=ALL
/NOSUPPRESS (D)

/SUPPRESS=CONSTRAINT_CHECKS causes the PSS Ada Compiler to eliminate all checks performed to test for constraint errors. This compiler option is used where higher execution performance is necessary.

/SUPPRESS=ELABORATION_CHECKS causes the PSS Ada Compiler to eliminate all checks performed during elaboration. This compiler option is used where elaboration order is specified by the user.

/SUPPRESS=STACK_CHECKS causes the PSS Ada Compiler to eliminate all checks on the run-time stack. This compiler option is used where higher exection performance is necessary.

/SUPPRESS=ALL has the same effect as combining every suppress option. It has the same effect as the expression:

/SUPPRESS=(CONSTRAINT_CHECKS, ELABORATION_CHECKS, STACK_CHECKS)