

AD-A252 468



ITATION PAGE

Form Approved
OPM No.

average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE (Leave)	2. REPORT	3. REPORT TYPE AND DATES Final: 23 Apr 1992 to 01 Jun 1993
--------------------------	-----------	---

4. TITLE AND Validation Summary Report: Proprietary Software Systems, Inc., PSS VAX/ZR34325 Ada Compiler Vers. XB-01.000, VAX 8350/VMS (Host) to Zoran ZR34325/PSS AdaRAID(Target), 92042311.11250	5. FUNDING 2
---	---

6. IABG-AVF Ottobrunn, Federal Republic of Germany	
--	--

7. PERFORMING ORGANIZATION NAME(S) AND IABG-AVF, Industrieanlagen-Betriebsgesellschaft Dept. SZT/ Einsteinstrasse 20 D-8012 Ottobrunn FEDERAL REPUBLIC OF GERMANY	8. PERFORMING ORGANIZATION IABG-VSR
---	--

9. SPONSORING/MONITORING AGENCY NAME(S) AND Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081	10. SPONSORING/MONITORING AGENCY
---	----------------------------------

11. SUPPLEMENTARY	
-------------------	--

DTIC
S ELECTE D
JUL 01 1992
A

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

13. (Maximum 200) Proprietary Software Systems, Inc., PSS VAX/ZR34325 Ada Compiler Vers. XB-01.000, VAX 8350/VMS (Host) to Zoran ZR34325/PSS AdaRAID(Target), ACVC 1.11.

92

041

92-17194

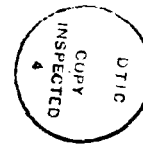


14. SUBJECT Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A,		15. NUMBER OF	
		16. PRICE	
17. SECURITY CLASSIFICATION UNCLASSIFIED	18. SECURITY UNCLASSIFIED	19. SECURITY CLASSIFICATION UNCLASSIFIED	20. LIMITATION OF

AVF Control Number: IABG-VSR 104
23 April, 1992

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 92042311.11250
Proprietary Software Systems, Inc.
PSS VAX/ZR34325 Ada Compiler Vers. XB-01.000
VAX 8350/VMS =>
ZORAN ZR34325/PSS AdaRAID
(bare machine simulation under VAX/VMS)

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



Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Certificate Information

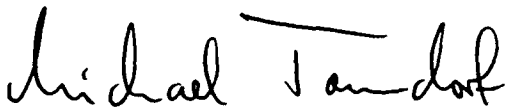
The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on April 23, 1992.

Compiler Name and Version: PSS VAX/ZR34325 Ada Compiler Version XB-01.000
Host Computer System: VAX 8350 under VMS Version 5.4
Target Computer System: Zoran ZR34325 Floating Point Digital Signal Processor/PSS AdaRAID Version XK-01.000 (bare machine simulation under VAX/VMS 5.4)

See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 920423I1.11250 is awarded to PSS, Inc. This certificate expires 24 months after ANSI approval of MIL-STD 1858B.

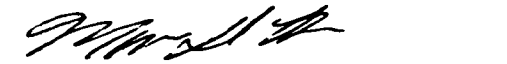
This report has been reviewed and is approved.



IAPG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany



for Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311



for Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

L001-2597

Customer: Proprietary Software Systems, Inc.

Certificate Awardee: Proprietary Software Systems, Inc.

Ada Validation Facility: IABG, mbH

ACVC Version: 1.11

Ada Implementation:

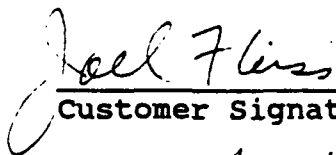
Ada Compiler Name and Version: PSS VAX/ZR34325 Ada
Compiler, Version XB-01.000

Host Computer System: VAX 8350/VMS 5.4

Target Computer System: PSS Zoran ZR34325 Digital
Signal Processor AdaRAID
Simulator Version XK-01.000

Declaration:

I, the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL/-STD-1815A ISO 8652-1987 in the implementation listed above.


Customer Signature

Date: 15 May 1992

Certificate Awardee Signature

Date:

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	USE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2	REFERENCES.	1-1
1.3	ACVC TEST CLASSES	1-2
1.4	DEFINITION OF TERMS	1-2
CHAPTER 2	IMPLEMENTATION DEPENDENCIES	
2.1	WITHDRAWN TESTS	2-1
2.2	INAPPLICABLE TESTS.	2-1
2.3	TEST MODIFICATIONS.	2-4
CHAPTER 3	PROCESSING INFORMATION	
3.1	TESTING ENVIRONMENT	3-1
3.2	SUMMARY OF TEST RESULTS	3-1
3.3	TEST EXECUTION.	3-2
APPENDIX A	MACRO PARAMETERS	
APPENDIX B	COMPILATION SYSTEM OPTIONS	
APPENDIX C	APPENDIX F OF THE Ada STANDARD	

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body 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 implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.
- [UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test 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. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT 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 package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. 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. If these units are not operating correctly, validation testing is discontinued.

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 all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint	The part of the certification body which provides policy and

Program Office (AJPO)	guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].

Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 02 August 1991.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 327 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113C..Y (23 tests)	C35705C..Y (23 tests)
C35706C..Y (23 tests)	C35707C..Y (23 tests)
C35708C..Y (23 tests)	C35802C..V (24 tests)
C45241C..Y (23 tests)	C45321C..Y (23 tests)
C45421C..Y (23 tests)	C45521C..Z (24 tests)
C45524C..Z (24 tests)	C45621C..Z (24 tests)
C45641C..Y (23 tests)	C46012C..Z (24 tests)

The following 21 tests check for the predefined type `SHORT_INTEGER`; for this implementation, there is no such type:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

The following 20 tests check for the predefined type `LONG_INTEGER`; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than `INTEGER`, `LONG_INTEGER`, or `SHORT_INTEGER`; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`; for this implementation, there is no such type.

C35713C, B86001U, and C86006G check for the predefined type `LONG_FLOAT`; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`; for this implementation, there is no such type.

A35801E checks that `FLOAT'FIRST..FLOAT'LAST` may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (See section 2.3.)

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater; for this implementation, `MAX_MANTISSA` is less than 47.

C45536A, C46013B, C46031B, C46033B, and C46034B contain length clauses that specify values for `'SMALL` that are not powers of two or ten; this implementation does not support such values for `'SMALL`.

C45624A..B (2 tests) check that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, `MACHINE_OVERFLOW` is `TRUE`.

D64005F..G use 17 levels of recursive procedure calls nesting; this level of nesting for procedure calls exceeds the capacity of the compiler.

B86001Y uses the name of a predefined fixed-point type other than type `DURATION`; for this implementation, there is no such type.

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C were graded inapplicable by Evaluation Modification as directed by the AVO. These tests instantiate generic units before those units' bodies are compiled; this implementation creates dependences as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete, and the objectives of these tests cannot be met.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE SMALL; this implementation does not support decimal SMALLs. (See section 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

AE2101C and EE2201D..E (2 tests) use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults; these instantiations are rejected by this compiler.

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

The following 264 tests check operations on sequential, text, and direct access files; this implementation does not support external files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A expect that NAME_ERROR is raised when an attempt is made to create a file with an illegal name; this implementation does not support the creation of external files and so raises USE_ERROR.

2.3 TEST MODIFICATIONS

Modifications (see Section 1.3) were required for 100 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24007A	B24009A	B25002B	B32201A	B33204A
B33205A	B35701A	B36171A	B36201A	B37101A	B37102A
B37201A	B37202A	B37203A	B37302A	B38003A	B38003B
B38008A	B38008B	B38009A	B38009B	B38103A	B38103B
B38103C	B38103D	B38103E	B43202C	B44002A	B48002A
B48002B	B48002D	B48002E	B48002G	B48003E	B49003A
B49005A	B49006A	B49006B	B49007A	B49007B	B49009A
B4A010C	B54A20A	B54A25A	B58002A	B58002B	B59001A
B59001C	B59001I	B62006C	B67001A	B67001B	B67001C
B67001D	B74103E	B74104A	B74307B	B83E01A	B85007C
B85008G	B85008H	B91004A	B91005A	B95003A	B95007B
B95031A	B95074E	BA1001A	BC1002A	BC1109A	BC1109C
BC1206A	BC2001E	BC3005B	BD2A06A	BD2B03A	BD2D03A
BD4003A	BD4006A	BD8003A			

E28002B was graded passed by Processing Modification as directed by the AVO. This test checks that pragmas may have unresolvable arguments, and it includes a check that pragma LIST has the required effect; but, for this implementation, pragma LIST has no effect if the compilation results in errors or warnings, which is the case when the test is processed without modification. This test was also processed with the pragmas at lines 46, 58, 70 and 71 commented out so that correct operation of pragma LIST was demonstrated.

C34003A was graded passed by Evaluation Modification as directed by the AVO. This test checks that required predefined operations are declared for floating-point types, and it includes a check for 'BASE'SIZE for a floating-point type. This implementation's target architecture uses an effective word size of 24 bits for all but floating-point objects, which occupy an additional 8 bits (for the sign and exponent) which are inaccessible to all but floating-point machine operations. To accommodate this unusual architecture with a normal storage-allocation scheme, both SYSTEM.STORAGE SIZE and FLOAT'BASE'SIZE are set to 24; hence, the check for 'BASE'SIZE at line 186 is failed (since the additional 8 bits are not counted in 'SIZE). The AVO accepted this deviation as justified by the target architecture in terms of AI-00325. Thus the test was graded passed given that the only Report.Failed output was (from line 187):

"INCORRECT 'BASE'SIZE"

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO. The compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. LRM 3.5.7:12).

C35902A was graded passed by Test Modification as directed by the AVO. This test checks that 'SMALL for fixed-point types has the correct value. This implementation's SYSTEM.MAX MANTISSA = 24, but the test declares a type at line 20 that requires 25 bits; the implementation rejects this declaration. This test was modified by changing the value of the named number at line 13 (which is used as delta for the type at line 20) from 2 ** -24 to 2 ** -23; i.e., line 13's declaration was changed to:

```
D_TINY : CONSTANT := 16#0.000002#;    -- (2 ** -23)
```

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package Report's body, and thus the packages' calls to function REPORT.IDENT INT at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

B83E01B was graded passed by Evaluation Modification as directed by the AVO. This test checks that a generic subprogram's formal parameter names (i.e. both generic and subprogram formal parameter names) must be distinct; the duplicated names within the generic declarations are marked as errors, whereas their recurrences in the subprogram bodies are marked as "optional" errors--except for the case at line 122, which is marked as an error. This implementation does not additionally flag the errors in the bodies and thus the expected error at line 122 is not flagged. The AVO ruled that the implementation's behavior was acceptable and that the test need not be split (such a split would simply duplicate the case in B83E01A at line 15).

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C were graded inapplicable by Evaluation Modification as directed by the AVO. These tests instantiate generic units before those units' bodies are compiled; this implementation creates dependences as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete, and the objectives of these tests cannot be met.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by re-compiling the obsolete units; all intended errors were then detected by the compiler.

BC3204D and BC3205C were graded passed by Test Modification as directed by the AVO. These tests are similar to BC3204C and BC3205D above, except that all compilation units are contained in a single compilation. For these two tests, a copy of the main procedure (which later units make obsolete) was appended to the tests; all expected errors were then detected.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-10 value as 'SMALL for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal 'SMALLs may be omitted.

AD9001B and AD9004A were graded passed by Processing Modification as directed by the AVO. These tests check that various subprograms may be interfaced to external routines (and hence have no Ada bodies). This implementation requires that a file specification exists for the foreign subprogram bodies. The following command was issued to the Librarian to inform it that the foreign bodies will be supplied at link time (as the bodies are not actually needed by the program, this command alone is sufficient):

```
zalib interface/system/library=lib_name ad9001b & ad9004a
```

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical and sales information about this Ada implementation, contact:

Proprietary Software Systems, Inc. (PSS)
Mr. Richard Gilinski
429 Santa Monica Blvd.
Santa Monica, California 90401
USA
Tel. (301) 394-5233
Fax (301) 393-3122

Testing of this Ada implementation was conducted at the AVF's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3388	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	96	
d) Non-Processed I/O Tests	264	
e) Non-Processed Floating-Point Precision Tests	327	
f) Total Number of Inapplicable Tests	687	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

With the customer's macro parameters a customised test suite was produced on the host computer system (VAX 8350) which was used for running the validation. Next the Ada implementation, the bare target simulation program and the command scripts, as supplied by the customer, were loaded and installed on the host. Then the full set of customised tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were executed by the bare target simulation program on the host computer system. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were for compiling

class-B- and class-E-tests

REPLACE	replace exisiting unit in library
LIST=SOURCE	produce a compilation listing with embedded error messages
NOSAVE SOURCE	don't keep the source listing in the library
LIBRARY=library_name	name of the library where the unit is compiled
WIDTH=79	width of compilation listing

all other test classes

REPLACE	replace exisiting unit in library
NOSAVE SOURCE	don't keep the source listing in the library
LIBRARY=library_name	name of the library where the unit is compiled

No explicit linker options were used except for tests AD9001B and AD9004A (see 2.3).

Test output, compiler and linker listings, and job logs were captured on a magnetic tape and archived at the AVF. The listings examined by the validation team were also archived.

APPENDIX A MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$MAX_IN_LEN	240 -- Value of V
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	"" & (1..V/2 => 'A') & ""
\$BIG_STRING2	"" & (1..V-1-V/2 => 'A') & '1' & ""
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	"" & (1..V-2 => 'A') & ""

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
\$ACC_SIZE	24
\$ALIGNMENT	8
\$COUNT_LAST	252
\$DEFAULT_MEM_SIZE	131072
\$DEFAULT_STOR_UNIT	24
\$DEFAULT_SYS_NAME	ZR34325
\$DELTA_DOC	2#1.0#E-23
\$ENTRY_ADDRESS	0
\$ENTRY_ADDRESS1	1
\$ENTRY_ADDRESS2	2
\$FIELD_LAST	252
\$FILE_TERMINATOR	ASCII.EOF
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
\$FORM_STRING	" "
\$FORM_STRING2	CANNOT_RESTRICT_FILE_CAPACITY
\$GREATER_THAN_DURATION	90_000.0
\$GREATER_THAN_DURATION BASE LAST	131_073.0
\$GREATER_THAN_FLOAT_BASE LAST	1.80141E+38
\$GREATER_THAN_FLOAT_SAFE LARGE	1.0E308
\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
\$HIGH_PRIORITY	200
\$ILLEGAL_EXTERNAL_FILE_NAME1	BAD&BAD
\$ILLEGAL_EXTERNAL_FILE_NAME2	BAD&BAD&BAD
\$INAPPROPRIATE_LINE_LENGTH	-1
\$INAPPROPRIATE_PAGE_LENGTH	-1
\$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.ADA")

MACRO PARAMETERS

\$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006F1.ADA")
\$INTEGER_FIRST	-8388608
\$INTEGER_LAST	8388607
\$INTEGER_LAST_PLUS_1	8388608
\$INTERFACE_LANGUAGE	CLINK
\$LESS_THAN_DURATION	-90_000.0
\$LESS_THAN_DURATION_BASE FIRST	-131_073.0
\$LINE_TERMINATOR	ASCII.CR
\$LOW_PRIORITY	10
\$MACHINE_CODE_STATEMENT	Z'(TRAN,(GEN,3,0,0));
\$MACHINE_CODE_TYPE	OP_CODE
\$MANTISSA_DOC	23
\$MAX_DIGITS	6
\$MAX_INT	8388607
\$MAX_INT_PLUS_1	8_338_608
\$MIN_INT	-8388608
\$NAME	NO_OTHER_INTEGER_TYPE
\$NAME_LIST	ZR34325
\$NEG_BASED_INT	16#FFFFFFE#
\$NEW_MEM_SIZE	131072
\$NEW_STOR_UNIT	24
\$NEW_SYS_NAME	ZR34325
\$PAGE_TERMINATOR	ASCII.FF
\$RECORD_DEFINITION	RECORD OPERATION:OP CODE; OPERANDS: OPERAND; END RECORD;
\$RECORD_NAME	Z
\$TASK_SIZE	24
\$TASK_STORAGE_SIZE	2048
\$TICK	0.02
\$VARIABLE_ADDRESS	0
\$VARIABLE_ADDRESS1	1
\$VARIABLE_ADDRESS2	2

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

4.4. ADA Command Line Qualifiers

4.4.1. **/[NO]CROSS_REFERENCE**

Specifying **/CROSS_REFERENCE** results in a cross reference listing being generated as part of the .MLS listing file.

The default is **/NOCROSS_REFERENCE**

4.4.2. **/[NO]DEBUG**

Specifying **/DEBUG** results in the inclusion of symbolic debug information in the object file. This information will be included at link time in the executable image file so that symbolic debugging may be performed. **/NODEBUG** results in no debug information in the object file.

The default is **/NODEBUG**.

4.4.3. **/[NO]JUMP_PADDING**

When an interrupt or a fault needs to result in the raising of an exception, the flow of control is changed in the interrupt handler so that control is passed to the exception raising program, instead of returning to the point of the interrupt or fault. However, if there is a control branch instruction waiting in the FIFO when the interrupt or fault occurs, then control may never reach the exception raising program. To avoid this problem, the compiler generates an extra conditional jump in front of every branch instruction (JMP, JMPC, CALL, RET). If this is not an issue for the system under development (for example, all interrupts are to be masked, or all interrupts will have control returned to the point of the interrupt), then these extra jumps may be omitted by using **/NOJUMP_PADDING**.

The default is **/JUMP_PADDING**.

4.4.4. **/LIBRARY=library-name**

Specifies the library into which the file is to be compiled.

4.4.5. /LENGTH=nn

The /LENGTH option allows you to specify the number of lines per page that are produced in the .LIS file. The default value is 60.

4.4.6. /[NO]LIST[=*listing-option-list*]

listing-option-list: Specifies the type of listing(s) to be generated. Allowable options are:

SOURCE
MACHINE_CODE
EMBEDDED_SOURCE
ALL

The default is /NOLIST. Any combination of SOURCE, MACHINE_CODE and EMBEDDED_SOURCE may be specified; however, EMBEDDED_SOURCE only takes effect when MACHINE_CODE is also specified. To specify multiple suboptions, enclose in parentheses and separate suboptions with commas. Example:

/LIST=(SOURCE,MACHINE_CODE)

/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=MACHINE produces a listing file of the machine code generated by the Ada Compiler 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_CODE,EMBEDDED_SOURCE)

/LIST is the same as:

/LIST=SOURCE

4.4.7. /NOENUMIMAGE

When an enumeration type is declared, constants are allocated for the 'IMAGE attribute of the type—that is, character constants containing the names of the elements of the enumeration constants are created. If the user does not intend to use 'IMAGE or 'VALUE for these enumeration types, then these constants can be omitted by using /NOENUMIMAGE.

There is no `/ENUMIMAGE` option; the default is that the character constants are allocated.

4.4.8. `/[NO]OPTIMIZE`

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

4.4.9. `/[NO]PHASES`

This option controls whether the compiler announces each phase of processing as it occurs. These phases indicate progress of the compilation. The default is `/NOPHASE`.

4.4.10. `/[NO]PROTECT_RAM`

It is assumed that the user will be making use of the Zoran RAM via MSPs or assembly code. In the instances where the compiler must use the RAM (e.g., multiplication, integer-to-float and float-to-integer conversions), if `/PROTECT_RAM` is selected, the entries of the RAM that are used by the compiler will be saved and restored so as not to interfere with values that the user has loaded. If the values in the RAM are expendable, the save/restore of these values may be omitted by using `/NOPROTECT_RAM`.

The default is `/PROTECT_RAM`.

4.4.11. `/[NO]REFINE`

Controls whether the compiler, when compiling a library unit, determines whether the unit is a refinement of its previous version and, if so, does not make dependent units obsolete. The default is `/NOREFINE`.

4.4.12. `/[NO]SAVE_SOURCE`

Controls the creation of a safety copy of the source code in the library directory. The default is `/SAVE_SOURCE`. You may save disk space by using `/NOSAVE_SOURCE`.

4.4.13. `/[NO]SHORT_INT_COMPARE`

Selecting `/SHORT_INT_COMPARE` results in simpler code for integer compares. Only a subtract is performed to set the condition codes. However, if numbers being compared have values such that a subtraction causes an overflow (such as $(2^{**}23)-1$ and -1), then incorrect results will be obtained. If such comparisons may occur, use `/NOSHORT_INT_COMPARE`. If it may be guaranteed that such comparisons will not occur, then use `/SHORT_INT_COMPARE` to obtain more efficient code.

The default is `/NOSHORT_INT_COMPARE`.

4.4.14. `/[NO]SUPPRESS[=suppress-option]`

suppress-option: Specifies the various Ada checks that are to be suppressed. Multiple suppress-options can be supplied by separating them with commas and enclosing the list in parentheses.

`/SUPPRESS=CONSTRAINT_CHECKS` causes the 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 Ada Compiler to eliminate all checks performed during elaboration. This compiler option is used where elaboration order is specified by the user or higher execution performance is necessary.

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

`/SUPPRESS=ALL` has the same effect as combining every suppress option.

4.4.15. `/SYNTAX_ONLY`

Parses a unit and reports syntax errors, then stops compilation without entering a unit in the library.

4.4.16. `/WIDTH=nn`

The `/WIDTH` option allows you to specify the width of the .LIS listing file that is produced. The default value is 124 characters.

VISIBLE	only visible imports can overwrite a local unit.
HIDDEN	both visible and hidden imports can overwrite local units.
NONE	no imported units (visible or hidden) may overwrite local units.

/[NO]CONFIRM: Requests that the user be given the opportunity to confirm before each compilation unit is imported. NOCONFIRM is the default.

/[NO]LOG: Causes a message to be written to the standard output device when a compilation unit is imported. NOLOG is the default.

3.3.16. LINK [/qualifier...] comp-unit-name

This command performs the following actions:

- checks that the unit within the library specified by the user has the legal form for a main unit
- checks the consistency of the unit's link closure
- finds all required object files
- links the main program with its link closure producing an executable image in the default directory.

comp-unit-name: Specifies the unit to be made the main program. An error message is issued if the compilation unit does not exist. All units in the link closure must exist and must be consistent.

The following command qualifiers may be used:

/[NO]KEEP: controls whether temporary files created by the librarian are deleted. /NOKEEP is the default.

/LIBRARY=*library-name*: Specifies the project and name of the library containing the compilation unit to be made a main program. An error message is issued if the library is a specification library. An error message is issued if the project does not exist. The project need not be specified if there is a default project. An error message is issued if it is not specified and there is no default project. An error message is issued if the library does not exist. The library need not be specified if there is a default library. An error message is issued if a library is not specified and there is not a default library.

APPENDIX C

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 this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD can be found on page 12 of the following documentation.

MIL-STD-1815A Appendix F

This section discusses the MIL-STD-1815A Appendix F issues that are left up to the implementor.

Predefined Pragmas

The LACE Ada Compiler supports the following predefined pragmas:

ELABORATE, INLINE, LIST, PACK, PAGE, PRIORITY, and SUPPRESS

Pragmas LIST and PAGE have effect only if there are no compilation errors or warnings in the units which contain them.

Pragma PACK causes the densest possible representation to be used. If a length clause is applied to the array type, then the pragma is ignored. If a length clause is applied to the component type of an array, then the array is packed as tightly as possible, using the specified size of the component. Note that pragma PACK does not change the allocation of string types or other arrays of characters. Characters are allocated one per referable unit. For records, the compiler chooses the densest possible allocation that conforms to all size and alignment constraints for all components of the record. Pragma PACK has an effect only if some of the component types have size specifications and are non-referable. Otherwise, each component is allocated a referable amount of space. Note that boolean types, and other types that require exactly one bit, are packed in records only if there is more than one such component within the record. Otherwise, a single, one-bit item will occupy a referable unit. Pragma PACK does not change the allocation of any component placed by a component clause. Also, pragma PACK does not make use of gaps left within the record as a result of such specifications. See the descriptions of size specifications for arrays and size specifications for records, later in this section, for additional information about array and record allocation.

The compiler accepts pragmas MEMORY_SIZE, STORAGE_UNIT and SYSTEM_NAME, but only the predefined values defined in package SYSTEM are allowed.

The compiler accepts pragma INTERFACE, but only CLINK is allowed as the language name. The linkage conventions used for CLINK are the same as the standard Ada linkage conventions. The compiler may rename entities that have

pragma INTERFACE applied to them, thereby requiring the user to examine the generated name in order to provide an implementation routine with the same spelling. The implementation-defined pragma LINKAGE_NAME (see below) may be used to force the compiler to use a specific name for an entity. Furthermore, implementation-defined pragma FOREIGN_BODY, in conjunction with pragma LINKAGE_NAME, may be used to achieve an effect similar to that of INTERFACE.

The LACE Ada Compiler does not support the following predefined pragmas:

- CONTROLLED: This is not supported because automatic storage reclamation does not occur.
- OPTIMIZE: Optimization is controlled via command line option.
- SHARED: This pragma is not supported. No warning is generated if SHARED is used.

Implementation-defined Pragmas

The LACE Ada Compiler also supports the following implementation-defined pragmas:

Pragma LINKAGE_NAME

Pragma LINKAGE_NAME is used to specify an exact external name to a given linkable entity (such as a function or a variable). The syntax is:

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

Ada-simple-name must be the name of an Ada entity declared in a package specification, or the name of a library-level subprogram or function. This entity must be something that exists at runtime, such as a subprogram, an exception, or an object. It cannot be a named number or string constant. The pragma is placed after the declaration of the entity.

LINKAGE_NAME causes the string-constant to be used in the generated object code for references to the Ada-simple-name. The user must assure that the name specified by string-constant is unique for the program under construction, and that the form of the name is legal.

An example of pragma LINKAGE_NAME usage:

```
procedure SPECIAL_CASE ( x, y : integer) ;  
pragma LINKAGE_NAME (SPECIAL_CASE, "xx_Special_Case") ;
```

Here, the procedure `SPECIAL_CASE` is provided with the linkage name `xx_Special_Case`. As a result, all calls to `SPECIAL_CASE` will result in a reference in the object code to the external name `xx_Special_Case`.

Pragma `FOREIGN_BODY`

Pragma `FOREIGN_BODY` provides a method to access subprograms and data written in other languages--or even other Ada programs. Pragma `FOREIGN_BODY` expands upon the utility of pragma `INTERFACE` by providing the capability of accessing data as well as subprograms written in other languages. The syntax is:

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

Pragma `FOREIGN_BODY` must appear in a non-generic library package. It must appear before any declarations. A package that includes pragma `FOREIGN_BODY` may not include any type declarations. All objects, subprograms, functions, and exceptions declared in such a package must be supplied by a foreign object module.

Language_name is a string. Any string may be used; the LACE compiler will follow its standard linkage conventions for all subprogram and function calls. The user must assure that these conventions are adhered to in the actual foreign object module.

The optional elaboration routine_name is a string which, if supplied, results in a call to a procedure of that name during program elaboration. This elaboration routine must be provided in the foreign object module supplied by the user. The user is responsible for initialization of all objects within such a package, whether or not an elaboration routine is specified.

A package that uses pragma `FOREIGN_BODY` may contain only subprogram declarations, object declarations that use unconstrained type marks, number declarations, and other pragmas. No object whose type mark is a task type, or includes a component whose type mark is a task type, may be included in such a package. If any of the restrictions for pragma `FOREIGN_BODY` are violated, then the pragma is ignored, and a warning message is issued.

Pragma `LINKAGE_NAME` should be used for all declarations in a package that uses pragma `FOREIGN_BODY`. When `LINKAGE_NAME` is not used for a given entity, the compiler may rename that

entity, and use this new spelling for all references to the entity. Thus, when the foreign object module is supplied, the references from other Ada units will not resolve at link time.

The user specifies the foreign object module to the librarian via the FOREIGN_BODY command. For example, for a package named STUFF that includes pragma FOREIGN_BODY, one would issue the following librarian command to provide a foreign object module named STUFF_BODY.OBJ for package STUFF:

```
ZALIB/LIBRARY=project:library FOREIGN_BODY STUFF STUFF_BODY.OBJ
```

See the section on the librarian commands for more information on the FOREIGN_BODY command.

An example of pragma FOREIGN_BODY usage:

```
package STUFF is
  pragma FOREIGN_BODY("assembly", "xx_stuff_elab");

  xyz : integer;
  pragma LINKAGE_NAME (xyz, "xx_xyz");

  procedure SPECIAL_CASE ( x, y : integer) ;
  pragma LINKAGE_NAME (SPECIAL_CASE, "xx_Special_Case") ;

end STUFF;
```

Compilation of Package STUFF will generate no object code. Instead, the user must supply an object module for STUFF via the FOREIGN_BODY command to the librarian, as described above. In the object module for STUFF there should be at least three externally referable labels: xx_stuff_elab, xx_xyz, and xx_Special_Case. Xx_stuff_elab should be the entry point of a parameterless procedure; this procedure will be called to perform the elaboration for package STUFF. Xx_xyz should be a label for an integer data object; references to STUFF.XYZ will resolve to references to xx_xyz. Xx_Special_Case should be the entry point for the implementation of procedure SPECIAL_CASE; calls to SPECIAL_CASE will result in calls to xx_Special_Case.

Implementation-dependent Attributes

The LACE Ada Compiler supports no implementation-dependent attributes.

Package SYSTEM

The predefined package SYSTEM contains the definitions of certain implementation defined characteristics. SYSTEM is

defined for the LACE Ada Compiler as follows:

```
package system is
-- Package System for PSS Ada Zoran Target
subtype address is new integer;
type name is (ZR34325);
system_name : constant name := ZR34325;
storage_unit : constant := 24 ; -- 24 bits per storage unit
memory_size : constant := 131072; -- 128K units
max_int : constant := 8388607; -- 24 bit integer
min_int : constant := -max_int - 1;
max_digits : constant := 6; -- 6 digits for floating point
max_mantissa : constant := 23; -- fraction for fixed point
fine_delta : constant := 2#1.0#e-23; -- for fixed point
tick : constant := 0.02; -- delta of tick
subtype priority is integer range 10 .. 200;
default_priority : constant priority := priority'first;
runtime_error : exception;
end system;
```

Restrictions on Representation Clauses

The LACE compiler has a basic restriction on all representation clauses:

- o Representation clauses may be given only for types declared in terms of a type definition, excluding generic_type_definitions (LRM 12.1) and private_type_definitions (LRM 7.4)..

Furthermore, some types have minimal alignment requirements that must be adhered to when specifying representations. Any representation that does not meet these requirements is diagnosed by the compiler and ignored. The general rule for minimal alignment of types is summarized:

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

Any representation clause in violation of the above rules is diagnosed, and is not obeyed. The following sections describe further restrictions. A violation of any of these restrictions results in a diagnostic message from the compiler.

Restrictions on Size Specifications

The following fundamental rules apply to all classes of size specifications:

- o The size is specified in bits and must be given by a static expression.

- o The given size is taken as a request to store objects of the type in that number of bits when feasible--use of a smaller size is never attempted, even if possible. The following describe what is feasible:
 - o An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine. In other words, multiple individual objects are never packed into a single address unit. If such packing is desired, combine the individual objects into a component structure, and specify the components.
 - o Formal parameters of a type with a specified size are allocated as is required by parameter passing conventions. Any necessary size conversions are performed during parameter passing and are transparent to the user.
 - o For a component within a composite type, if adjacent bits are not otherwise occupied, they may be affected by stores to the component; the compiler will allocate unused bits to non-referable components to make them referable.
- o A size specification for a type specifies the size for objects of that type and for any subtypes of the type. Even if a subtype would allow for a smaller representation, no attempt is made to use the smaller size.
- o A size specification for an access type must match the default size used by the compiler for the type.
- o Size specifications are not supported for floating point types or task types.

The following sections describe restrictions that are unique to each class of size specification.

Restrictions on Size Specifications for Scalar Types

Any size must be large enough to accommodate all possible values of the type including the value 0, even if 0 is not in the range of the values of the type. For numeric types with negative values, the size must include the sign bit.

A size specification for a real type does not affect the accuracy of operations for that type.

A size specification may not specify a size larger than the largest size supported by the target architecture for the representation of objects of that class of type. For the LACE compiler, the largest sizes are:

enumeration:	24 bits
integer:	24 bits
fixed:	24 bits
float:	24 bits

Restrictions on Size Specifications for Array Types

A size specification for an array type must be large enough to accommodate all components of the array under the densest packing strategy. Alignment constraints on all components must be obeyed--such constraints are described in a later section.

Arrays with component sizes less than or equal to 24 bits are densely packed; no pad or unused bits exist between components. Arrays with component sizes greater than 24 bits are padded up to the next 24-bit (i.e., referable unit) boundary. The size of the component type is not influenced by the size clause for the array. However, the representation of components declared via a subtype may be reduced to the minimum number of bits required for the range of values of the subtype (but not of the parent type), unless the parent type has a size specification.

If there is a size specification for the component type, but not for the array type, then the component size is rounded up to the nearest referable size, unless pragma PACK is used. This even applies to boolean types or other types that require only a single bit.

Restrictions on Size Specifications for Record Types

A size specification for a record type does not change the default layout of the record type. Any size given for a record type must be at least as large as the number of bits

determined by the default layout. Changing of the layout of a record type must be done by use of record representation clauses or by pragma PACK. As a consequence, size specifications for record types can only be used to increase the size of the record. Any decrease in the size of a record type must be achieved by using pragma PACK or by using representation clauses.

Neither the size nor representation of component types of a record are altered by a length clause for a record.

If a record contains an array component whose bounds depend on discriminants of the record or contains components of dynamic size, then implementation-dependent dope information components are allocated within the record. These dope components cannot be named or sized by the user.

A size specification may not be applied to a record type with dynamically sized components.

Restrictions on Size Specifications for Collections

The specification of a collection size causes the collection to be allocated with at least the specified size. The size is in storage units, and need not be static. If a static size of zero is specified, then no collection is allocated; however, if a non-static size of zero is specified, then the default collection size is allocated.

Any attempt to allocate more objects than the collection can hold causes `STORAGE_ERROR` to be raised. Dynamically sized records and arrays carry administrative overheads and must be accounted for when choosing a collection size. Furthermore, for each object allocated from a collection, there is an overhead of three words.

If no collection size is specified, then the default collection size is used.

Restrictions on Size Specifications for Task Activation

The specification of a task activation size causes the task activation to be allocated with the specified size, expressed in storage units. Any attempt to exceed the specified size results in the raising of `STORAGE_ERROR`. If no size is given, then the default task activation size is used.

Restrictions on Size Specification of 'SMALL

Only powers of two are allowed for 'SMALL. The length of the representation of a type may be effected by the specification

of 'SMALL. If a size specification is also given for the type, the size specification takes precedence; the size of 'SMALL must be accommodatable within the specified size.

Restrictions on Enumeration Representation Clauses

Internal codes specified for enumeration literals must lie within the range INTEGER'FIRST..INTEGER'LAST. Note that specification of the internal codes or enumeration literals results in significant runtime overhead--even if the specified codes match the default values. (The default values start with zero, and increase by one for each literal in the list.) Thus, it is not recommended to specify the internal codes for enumeration literals unless absolutely necessary.

Restrictions on Record Representation Clauses

Alignment clauses for record representation clauses are observed; however, only power of two values are allowed. The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record, as determined by component allocation and the minimum alignment requirements of those components is more stringent than the specified alignment.

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

The size specified for each component must be sufficient to allocate all possible values of the component subtype. The location specified must be compatible with any alignment constraints of the component type. Alignment constraints on a component type may cause an implicit alignment constraint upon the containing record type.

If all components and/or discriminants are not specified by component clauses, then the unspecified components and discriminants are allocated after those that are specified. No attempt is made to make use of any gaps left by the user-specified allocation.

Restrictions on Address Clauses

Address clauses are supported with the following restrictions:

- o The address expression must be a static expression.

- o Address clauses may only be applied to objects declared within library-level packages, and the address clause must be included in the package where the object is declared. For any other objects, address clauses are ignored.
- o When applied to an object, references to the object result in direct references to the exact location.

Implementation Generated Components in Records

The only implementation-generated components allocated within records are those generated for dope information for arrays whose bounds are determined by discriminants of the record. These dope information components may not be named by the user.

Restrictions on Unchecked Conversions

The sizes of both the source and target types for unchecked conversions must be known at compile time, but they need not be the same size. Conversion to a smaller size results in truncation, while conversion to a larger size results in zero-extension. Calls on instantiations of `unchecked_conversion` are automatically made inline.

Implementation-dependent Aspects of Input-Output Packages

The predefined packages `DIRECT_IO`, `SEQUENTIAL_IO`, and `TEXT_IO`, as defined in LRM chapter 14, are provided in this implementation; package `LOW_LEVEL_IO` is not. Because the ZR34325 is used in embedded applications, and because this processor has no input/output capability, the functionality of these packages is limited.

`DIRECT_IO` and `SEQUENTIAL_IO` raise `USE_ERROR` or `NAME_ERROR` if a file open or file access is attempted. `SEQUENTIAL_IO` and `DIRECT_IO` may not be instantiated for unconstrained array types, nor for record types with discriminants without default values.

`TEXT_IO` supports reading from `Standard_Input` and writing to `Standard_Output`. Any routine that takes a file name raises `USE_ERROR`, unless one of the two aforementioned standard files is used; however, `MODE_ERROR` is raised if the wrong one is used.

Package `MICRO_IO` is also provided to allow a simple text output capability. `MICRO_IO` allows output of strings. It is a small subset of the functions available in `Text_IO`.

Following is the source listing of the package specification for MICRO_IO.

```
-- *****
-- *****
-- ***                                     ***
-- ***   package specification MICRO_IO   ***
-- ***                                     ***
-- *****
--
-- Outline: This package contains a subset of the specification
--          for Ada Text Input-Output as described in LRM chapter 14.
--
-- *****
--
with PSS_IO_Types; Use PSS_IO_Types;
package MICRO_IO is

    subtype count is pss_io_Types.count;

    procedure Put_Line (L: String);

    procedure Set_Col (I: in Positive_Count);

end MICRO_IO;
```

Definition of the Main Program

Any library-level subprogram that has no parameters may be specified as the main program. Tasks initiated in library units follow the normal rules for termination; also, these tasks do not terminate simply because the main program has terminated. Thus, it is strongly recommended that terminate alternatives be provided in selective wait statements in library-level tasks.

Implementation of Generics

All instantiations of generics, except for the predefined generics UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION are implemented by code duplication.

The body of a generic must be compiled before the unit may be instantiated. However, the specification and body of a generic need not be included in the same compilation unit. A recompilation of the body of a generic will make obsolete any units which instantiate the generic.

Implementation-defined Characteristics from Package STANDARD

The following are the implementation-defined characteristics for the Zoran ZR34325 from Package STANDARD (LRM Appendix C):

```
type INTEGER is range -8388608 .. 8388607;
type FLOAT is digits 6 range -2#0.1111_1111_1111_1111_1111#E125
.. 2#0.1111_1111_1111_1111_1111#E125;
type DURATION is delta 0.02 range -86400.0 .. 86400.0;
```

Attributes of Type DURATION

The type DURATION has the following attributes for the ZR34325 target:

DURATION'delta:	0.015625 seconds
DURATION'small:	0.015625 seconds
DURATION'first:	-86400.0 seconds
DURATION'last:	86400.0 seconds

Attributes of Type INTEGER

The ZR34325 architecture supports only one predefined integer type, integer. Its attributes are:

integer'first:	-(2**23) or -8388608
integer'last:	(2*23)-1 or 8388607
integer'size:	24

Following are the bounds of types declared in TEXT_IO:

```
COUNT'first:      0
COUNT'last:     252

POSITIVE_COUNT'first:  1
POSITIVE_COUNT'last:  252

FIELD'first:      0
FIELD'last:      252
```

Following are the bounds of types declared in DIRECT_IO:

```
COUNT'first:      0
COUNT'last:     256

POSITIVE_COUNT'first:  1
POSITIVE_COUNT'last:  256
```

Attributes of Type FLOAT

The ZR34325 supports only one predefined floating point type, FLOAT. Notice that float'size and float'mantissa are both 24. Floating point operands in the ZR34325 are actually 32 bits; however, because only the least significant 24 bits of any ZR34325 word may be directly manipulated, STORAGE_UNIT is defined as 24 bits. If float'size were defined as 32 bits, two referable units would be allocated for each floating point object. In order to avoid this, float'size of 24 is used. All floating point objects, including floating point literals, are allocated exactly one 32-bit target word each. All floating point operations assume native ZR34325 floating point types. The attribute 'SIZE for floating point types does not affect storage allocation or code generation, both of which are optimized for the machine's characteristics.

```
float'size:      24
float'digits:    6
float'mantissa:  21
float'emax:      84
float'epsilon:   2#1.0#E-20
float'small:     2#1.0#E-85
float'large:     2#1.1111_1111_1111_1111#E83
float'safe_emax: 125
float'safe_small: 2#1.0#E-126
float'safe_large: 2#1.1111_1111_1111_1111#E124
float'first:     -2#1.1111_1111_1111_1111_1111_1111#E124
float'last:      2#1.1111_1111_1111_1111_1111_1111#E124
float'machine_radix: 2
float'machine_mantissa: 24
float'machine_emax: 125
float'machine_emin: -125
float'machine_rounds: TRUE
float'machine_overflows: TRUE
```

Package MACHINE_CODE

This implementation includes an implementation of package MACHINE_CODE. The user should refer to the listing of package MACHINE_CODE included in an appendix for further details.

The fundamental instruction record type is defined as follows:

```
type Z is
  record
    Operation : Op_Code;
    Operands  : Operand;
  end record;
```

Type Op_Code is defined in package MACHINE_CODE, and includes all of the ZR34325 opcodes. Type Operand is a variant record structure which is used to specify all other fields within the instruction. It is defined as follows:

```
type Operand(ot:operand_type:=GEN) is
  record
    case ot is
      when Addr => -- Use when you want to set EMA field with an
                  -- Address...(Op_4)
        -- Word 0
        Op_1 : ten_bits;           -- bits 16..25
        Op_2 : sixteen_bits;      -- bits 0..15
        -- Word 1
        Op_3 : thirteen_bits;     -- bits 20..31
        Op_4 : Zaddr;             -- bits 0..19
```

```

when Ilit => -- Use when you want to include an integer literal
             -- (Op_8)
             -- Word 0
Op_5 : ten_bits;           -- bits 16..25
Op_6 : sixteen_bits;      -- bits 0..15
             -- Word 1
Op_7 : eight_bits;        -- bits 24..31
Op_8 : INTEGER;           -- bits 0..23

when Flit => -- Use when you want to include a FLOAT literal
            -- (Op_11)
            -- Word 0
Op_9 : ten_bits;          -- bits 16..25
Op_10 : sixteen_bits;     -- bits 0..15
            -- Word 1
Op_11 : FLOAT;            -- bits 0..31

when Gen => -- Use for Gen purpose
           -- Word 0
Op_12 : ten_bits;         -- bits 16..25
Op_13 : sixteen_bits;     -- bits 0..15
           -- Word 1
Op_14 : sixteen_bits;     -- bits 16..31
Op_15 : sixteen_bits;     -- bits 0..15

end case;

end record;

```

All the fields are set via the actual bit values to be used; in other words, there is no symbolic references possible. This is a very simple interface, and requires that the user be very careful when specifying instructions.

Machine-specific Procedures (MSPs)

The LACE compiler includes two packages, ZR34325_MACHINE and ZR34325_MACHINE_DATA, that implement machine-specific procedures (MSPs) for the ZR34325 processor. These are provided as a set of procedure specifications that the user may call, which will result in the generation of one or more ZR34325 instructions. To make use of the Zoran MSPs you must import both of these packages via WITH clauses. ZR34325_MACHINE_DATA contains type and constant declarations, and ZR34325_MACHINE contains the actual MSP procedure specifications.

It is suggested that the prospective MSP user studies both of these packages carefully; source listings of both are included as an appendix. In order to make most effective use of the MSPs, use the constants defined in ZR34325_MACHINE_DATA, instead of literal integers, for the corresponding MSP parameters. These constants are defined for specific instruction fields, which map onto MSP

parameters, and, in many cases, their values have been chosen to provide the correct value to insert in those fields. As an example, the type regReference is defined to cover the range of values that may be specified for all the register bit fields of the register-based instructions (ALO, LDR, STR). The definition of regReference, and the constants defined for it, are:

```

subtype regReference is integer range 2..262142;
-- sum any of the following:
rA      : constant := 128;
rACC_I  : constant := 8192;
rACC_R  : constant := 4096;
rB      : constant := 64;
rIF     : constant := 512;
rIM     : constant := 32768;
rIP     : constant := 16384;
rPR     : constant := 2;
rLC     : constant := 8;
rMBS_MSS : constant := 65536;
rMNMXI  : constant := 2048;
rMNMXV  : constant := 1024;
rMODE   : constant := 131072;
rSAR    : constant := 32;
rSP     : constant := 16;
rSTATUS : constant := 256;
rX      : constant := 4;

```

Notice that all the values are powers of two. Each constant is equal to the value of the entire register bit mask area if only the register for that constant is set. Thus, the user can specify multiple registers simply by adding the register constants needed, such as rX+rA+rB+rLC.

Sample Program Using MSPs

The following is an example of a very simple program which uses some MSPs. This program is not intended to make any sense; its purpose is simply to show an example of MSP usage.

Source:

```
with system;
with MartinGlobal;    -- This contains ALPHA, GAMMA and BETA
use MartinGlobal;

with ZR34325_Machine_Data;
use ZR34325_Machine_Data;

with ZR34325_machine;
use ZR34325_machine;

procedure MartinTest is
begin

  ORR_LI(16,rMODE);
  ORR_LI(1,rMODE);
  LDR_LA(GAMMA(0)'address,rSAR);
  LD_RZ(64,ALPHA(0),C0);

  MULT_ER(64, C0, BETA(0), none, 0, ACC_use=>replace);
  MULT_ER(64, C0, BETA(0), none, 1, ACC_use=>replace);
  MULT_ER(64, C0, BETA(0), none, 2, ACC_use=>replace);
  MULT_ER(64, C0, BETA(0), none, 3, ACC_use=>replace);

  MAX_EX(64,GAMMA(0),repeat=>2);
  SHRR(1,rMNMXI);

end MartinTest;
```

Generated code:

```

                                AOAMRTNTST0000 EQU      $
                                ORIGIN HEX(6)
000006 2 31920000 00000010    ANDR #0x10, [$MODE] ;
                                * LINE NUMBER 10.
000008 2 31920000 00000001    ANDR #0x1, [$MODE] ;
                                * LINE NUMBER 11.
00000A 2 21E00020 00000000 X   LDR &AOAMRTNLBLO000#GAMMA#00 => [$SAR] ;
                                * LINE NUMBER 12.
00000C 2 10030040 80100000 X   LD_C:(64) AOAMRTNLBLO000#ALPHA#00:(1,1) => $C0 ;
                                * LINE NUMBER 14.
00000E 2 80028080 00000000 X   MULT_(R,R):(64) $C0, AOAMRTNLBLO000#BETA#00:(0,1) => $NULL, $ACC ;
                                * LINE NUMBER 15.
000010 2 80028080 10000000 X   MULT_(R,R):(64) $C0, AOAMRTNLBLO000#BETA#00:(0,2) => $NULL, $ACC ;
                                * LINE NUMBER 16.
000012 2 80028080 20000000 X   MULT_(R,R):(64) $C0, AOAMRTNLBLO000#BETA#00:(0,4) => $NULL, $ACC ;
                                * LINE NUMBER 17.
000014 2 80028080 30000000 X   MULT_(R,R):(64) $CC, AOAMRTNLBLO000#BETA#00:(0,8) => $NULL, $ACC ;
                                * LINE NUMBER 19.
000016 2 44028002 80100000 X   MAX_R:(64,2) AOAMRTNLBLO000#GAMMA#00:(1,1) => $MMMX ;
                                * LINE NUMBER 20.
000018 2 31E00800 80100000    SHRR:[SHIFT:1] [$MMMXI] ;
                                RETURN EQU      $
                                LB..001A EQU      $
00001A 2 38000001 06000001    RET ;
                                ORIGIN HEX(0)
000000 2 31600010 70000002    SUBR:[TC:1,TR:1] [$SP], #0x2 => $X ;
000002 2 21E00020 00000000    LDR #0x0 => [$SAR] ;
000004 2 24000034 06000003    STR [$SAR,$SP,$X] => $SP+(3) ;
                                ORIGIN HEX(1C)

```

Interrupt Processing Support Routines

The LACE compiler includes the following package specifications as part of the standard, pre-compile packages:

```

package DWELL_EXCEPTION is
DWELL_EXPIRED : exception;
pragma LINKAGE_NAME(DWELL_EXPIRED, "DWELL_EXPIRED");
end DWELL_EXCEPTION;

with SYSTEM;
package PSS_INTERRUPT_SERVICES is
pragma FOREIGN_BODY("ASSEMBLY");

procedure PSS_RECORD_INTERRUPTS (INT_MASK : integer;
ROUTINE : SYSTEM.ADDRESS) ;
pragma LINKAGE_NAME(PSS_RECORD_INTERRUPTS, "PSS_RECORD_INTERRUPTS");

procedure PSS_EXCEPTION_INTERRUPTS (INT_MASK : integer) ;
pragma LINKAGE_NAME(PSS_EXCEPTION_INTERRUPTS,
"PSS_EXCEPTION_INTERRUPTS");

end PSS_INTERRUPT_SERVICES;

```

The ZR34325 computer has one external interrupt that may be either an outgoing interrupt (i.e., to signal another processor) or an incoming interrupt (i.e., so that another processor may signal the ZR34325). The environment in which the LACE compiler is to be used treats the external interrupt as an incoming interrupt that signals that a processing dwell cycle has expired. When the LACE runtime interrupt handler processes the external interrupt, its normal action is to raise the exception `DWELL_EXPIRED`, defined in package `DWELL_EXCEPTION`, as shown above. The user may import `DWELL_EXCEPTION` and place handlers where needed to handle this exception.

The user may also import `PSS_INTERRUPT_SERVICES` and make calls to the two procedures defined therein, `PSS_RECORD_INTERRUPTS` and `PSS_EXCEPTION_INTERRUPTS`. These procedures allow the user to determine how certain interrupts are to be processed by the LACE runtime interrupt handler. Each of these procedures includes an integer input parameter called `INT_MASK`. `INT_MASK` is a mask in which the user specifies which interrupts are to be affected. For bit positions zero through twelve (with zero being the LSB), each bit in `INT_MASK` specifies an interrupt that may be affected; a one in that bit position means that the interrupt is to be affected, and a zero means that it is not. Thus, the actual range of meaningful values for `INT_MASK` is 1..8191. Any other bit positions that are set in `INT_MASK` are ignored. The following is the mapping of bit positions to ZR34325 interrupts:

Bit	Interrupt

0	ISZ -- Occurs during SPLIT if DV=1 and one of the elements is zero.
1	ISI -- Invalid adder or subtractor operation.
2	ISX -- Inexact subtractor result.
3	ISU -- Underflow in subtractor.
4	ISO -- Overflow in subtractor.
5	IAX -- Overflow in adder.
6	IAX -- Inexact adder result.
7	IDO -- Overflow in multiplier.
8	IDU -- Underflow in multiplier.
9	IDX -- Inexact result in multiplier.
10	IDI -- Invalid multiplier operation.
11	IRO -- Overflow during register operation.
12	IEI -- External interrupt.

Thus, a value of 3 for `INT_MASK` will affect the ISZ and ISI interrupts, while a value of 4096 (16#1000#) will affect the IEI interrupt.

`PSS_RECORD_INTERRUPTS` is called to instruct the interrupt handler to call the procedure specified by the second parameter, `ROUTINE`, when any of the interrupts identified in `INT_MASK` occur. The interrupt handler will do this, and when

the procedure returns to the interrupt handler, control will pass back to the point of the interrupt, as if no fault had occurred. The intention is to have the procedure record that the interrupt occurred for later evaluation. The second parameter to `PSS_RECORD_INTERRUPTS`, `ROUTINE` is of type `SYSTEM.ADDRESS`; the address of the fault-recording procedure should be passed as this second parameter. The fault-recording procedure must have two parameters, the first of type `SYSTEM.ADDRESS`, and the second of type integer. When the fault-recording procedure is called, the first parameter will contain the address to which the interrupt handler will return execution (and thus, very near the faulting instruction), and the second parameter will contain the value of the interrupt register at the time of the interrupt.

***** Important notes *****

Fault-recording procedures should not declare much (if any) local data, and should not call any other subprograms. This is because they are called using the interrupt handler's stack space, which is very limited. Furthermore, processing should be kept to a minimum. This is because the fault-recording procedures will execute in interrupt mode, which is much slower than master mode. Finally, if more than one bit is set in the interrupt flags register, the one to be processed is selected as follows: First, all interrupts other than the thirteen shown above are masked out. If the external interrupt has occurred, it alone is processed. Otherwise, the signaled interrupt with the highest bit number from the above list is used as the sole interrupt to be processed.

`PSS_EXCEPTION_INTERRUPTS` specifies that the interrupt handler should raise an exception for the interrupts specified by `INT_MASK`. The external interrupt, `IEI`, will raise the `DWELL_EXPIRED` interrupt. All other interrupts will raise `NUMERIC_ERROR`.

As an example, if the user wanted to specify that the `ISO` interrupt be recorded by procedure `RECORD_ISO` instead of being treated as an exception, then the following would be used:

```
procedure RECORD_ISO (FAULT_ADDRESS : SYSTEM.ADDRESS;
                     INTERRUPT_FLAGS : integer);

PSS_RECORD_INTERRUPTS(16, RECORD_ISO'address);
```