AD-A245 663

ON PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 25 Apr 1991 to 01 Jun 1993

4. TITLE AND SUBTITLE
Intermetrics, Inc., AFCAS 1750A/XMEM Ada Compiler, Version 1.0, DEC VAXstation 3100 (Host) to Air Force RAID MIL-STD-1750A Simulator (Target), 910425W1.11143

5. FUNDING NUMBERS

6. AUTHOR(S)
Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB, Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-458.1191

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

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

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

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Intermetrics, Inc., AFCAS 1750A/XMEM Ada Compiler, Version 1.0, DEC VAXstation 3100 , VMS Version 5.3 (Host) to Air Force RAID MIL-STD-1750A Simulator running on VMS Version 5.3 (Target), ACVC 1.11.

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

AVF Control Number: AVF_VSR_458.1191
16-November-1991
90-11-16-INT

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910425W1.11143
Intermetrics, Inc.
AFCAS 1750A/XMEM Ada Compiler, Version 1.0
DEC VAXstation 3100 => Air Force RAID MIL-STD-1750A Simulator

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright Patterson AFB OH 45433-6503

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 25 April, 1991.

Compiler Name and Version: AFCAS 1750A/XMEM Ada Compiler, Version 1.0

Host Computer System: DEC VAXstation 3100, VMS Version 5.3

Target Computer System: Air Force RAID MIL-STD-1750A simulator running on VMS Version 5.3

Customer Agreement Number: 90-11-16-INT

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

As a result of this validation effort, Validation Certificate 910425W1.11143 is awarded to Intermetrics, Inc. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

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

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

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

DECLARATION OF CONFORMANCE

Customer: Intermetrics, Inc., Cambridge, MA

Ada Validation Facility: ASD/SCEL Wright-Patterson AFB, OH 45433-6503

ACVC Version: 1.11

Ada Implementation

Compiler Name and Version: AFCAS 1750A/XMEM Ada Compiler, Version 1.0
Host Computer System: DEC VAXstation 3100, VMS Version 5.3
Target Computer System: Air Force RAID MIL-STD-1750A simulator running on VMS

Customer's Declaration

I, the undersigned, representing Intermetrics, Inc., declare that Intermetrics, Inc. has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration. I declare that the U.S. Air Force is the owner of the above implementation and the certificates shall be awarded only in its name.

Date: 4/25/91

Dennis D. Struble,
Deputy General Manager, Development Systems Group,
Intermetrics, Inc.

++++++++++++++++++ CO_SIGNER SIGNATURE BLOCK ++++++++++++++++++++

Date: 4/25/91

Joseph P. Hall
Deputy Director, Projects Division
Deputy for Avionics Control
Aeronautical Systems Division - Acquisition Logistics Division
Wright-Patterson Air Force Base, OH

TABLE OF CONTENTS

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
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

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 SPPRT13, 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 SPPRT13 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 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 Program Office (AJPO)	The part of the certification body which provides policy and 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.
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 14 March 1991.

E28005C	B28006C	C34006D	C35508I	C35508J	C35508M
C35508N	C35702A	C35702B	B41308B	C43004A	C45114A
C45346A	C45612A	C45612B	C45612C	C45651A	C46022A
B49008A	A74006A	C74308A	B83022B	B83022H	B83025B
B83025D	C83026A	B83026B	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	CE3607[illegible]
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.

IMPLEMENTATION DEPENDENCIES

The following 285 tests 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)

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.

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.

C45423A, C45523A, and C45622A check that the proper exception is raised if MACHINE_OVERFLOWS is TRUE and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is FALSE.

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.

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

D55A03E..H (4 tests) use 31 levels of loop nesting which exceeds the capacity of the compiler.

D56001B uses 65 levels of block nesting which exceeds the capacity of the compiler.

D64005G uses 17 levels of recursive procedure calls nesting; this 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.

CA2009C and CA2009F check whether a generic unit can be instantiated before the separate compilation of its body (and any of its subunits); this implementation requires that the body and subunits of a generic unit be in the same compilation as the specification if instantiations precede them. (See section 2.3.)

CD1009C checks whether a length clause can specify a non-default size for a floating-point type. The representation which this implementation uses for floating point types is the smallest available; therefore, when this test attempts to use a representation of other than 32 or 64 bits, the length clause is rejected.

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.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

AE2101C uses instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

AE2101H uses 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 for sequential, text, and direct access 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 use an illegal file name in an attempt to create a file and expect NAME_ERROR to be raised; this implementation does not support external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 9 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.

BA1101C BC3205D

C32107A was graded passed by Evaluation Modification as directed by the AVO. The test was run using a task size of 16#400 and a stack size of 16#1200. The other tests were run using a task size of 16#400 and a stack size of 16#1000.

C34003A was graded passed by test modification as directed by the AVO. This test checks operations on derived floating-point types and uses a complex expression whose intermediate results overflow in this implementation. For this implementation, T'MACHINE_RADIX = 2, T'MACHINE_EMAX =127, and T'MACHINE_EMIN =-128. With these values, the two expressions at lines 219 and 226 equate to the calculation 2**126; but this calculation will overflow on the 1750A as a consequence of the way multiplication occurs (even though the result can be represented), and NUMERIC_ERROR is raised. Thus, lines 219 and 225 were modified as shown below:

```
[219]
   from:  (1.0 * T'MACHINE_RADIX) ** (T'MACHINE_EMAX -1) / 2 <

     to:  ( (1.0 * T'MACHINE_RADIX) ** (T'MACHINE_EMAX -2) +
            (1.0 * T'MACHINE_RADIX) ** (T'MACHINE_EMAX -2) ) / 2 <

[225]
   from:  (1.0 * T'MACHINE_RADIX) ** (T'MACHINE_EMIN + 2) *

     to:  ( (1.0 * T'MACHINE_RADIX) ** (T'MACHINE_EMIN + 3) *
            (1.0 * T'MACHINE_RADIX) ** (-1)                  ) *
```

CA2009C and CA2009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the separate compilation of that unit's body; as allowed by AI-257, this implementation requires that the bodies of a generic unit be in the same compilation if instantiations of that unit precede the bodies. The instantiations were rejected at compile time.

CE2103A, CE2103B, and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

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 a point of contact for sales or technical information about this Ada implementation system, see:

Mike Ryer
Intermetrics, Inc.
733 Concord Ave.
Cambridge MA 02138

Testing of this Ada implementation was conducted at the customer'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.

a) Total Number of Applicable Tests	3436
b) Total Number of Withdrawn Tests	93
c) Processed Inapplicable Tests	92
d) Non-Processed I/O Tests	264
e) Non-Processed Floating-Point Precision Tests	285
f) Total Number of Inapplicable Tests	641
g) Total Number of Tests for ACVC 1.11	4170

The above number of I/O tests were not processed because this implementation does not support a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the the magnetic tape were loaded directly onto the host computer.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were run on the host machine using the target simulator. Results were transferred via RSCS to a Sun 4 for printing.

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:

Option | Switch	Effect
PTR_FILE	Ada program library name

Test output, compiler and linker 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.

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	255
$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:"

MACRO PARAMETERS

```
$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	16
$ALIGNMENT	1
$COUNT_LAST	32767
$DEFAULT_MEM_SIZE	2097152
$DEFAULT_STOR_UNIT	16
$DEFAULT_SYS_NAME	MIL_STD_1750A
$DELTA_DOC	2.0**(-31)
$ENTRY_ADDRESS	SYSTEM.MAKE_ADDRESS(0,16#40#)
$ENTRY_ADDRESS1	SYSTEM.MAKE_ADDRESS(0,16#80#)
$ENTRY_ADDRESS2	SYSTEM.MAKE_ADDRESS(0,16#100#)
$FIELD_LAST	32767
$FILE_TERMINATOR	TEST_WITHDRAWN
$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	10_000_000.0
$GREATER_THAN_FLOAT_BASE_LAST	1.0E+63
$GREATER_THAN_FLOAT_SAFE_LARGE	16#0.FFFFFFFFFFFE1#E+63

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        16#0.FFFFF9#E+63

$HIGH_PRIORITY          127

$ILLEGAL_EXTERNAL_FILE_NAME1
                        NO_FILES_AT_ALL_1

$ILLEGAL_EXTERNAL_FILE_NAME2
                        NO_FILES_AT_ALL_2

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        "PRAGMA INCLUDE ("A28006D1.TST")"

$INCLUDE_PRAGMA2        "PRAGMA INCLUDE ("B28006F1.TST")"

$INTEGER_FIRST          -32768

$INTEGER_LAST           32767

$INTEGER_LAST_PLUS_1    32768

$INTERFACE_LANGUAGE     AIE_ASSEMBLER

$LESS_THAN_DURATION     -90_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -10_000_000.0

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           -127

$MACHINE_CODE_STATEMENT
                        NULL;

$MACHINE_CODE_TYPE      NO_SUCH_TYPE

$MANTISSA_DOC           31

$MAX_DIGITS             9

$MAX_INT                32767

$MAX_INT_PLUS_1         32768

$MIN_INT                -32768
```

MACRO PARAMETERS

$NAME	NO_SUCH_INTEGER_TYPE
$NAME_LIST	UTS,MVS,CMS,PRIME50,SPERRY1100, MIL_STD_1750A
$NAME_SPECIFICATION1	NO_FILES_1
$NAME_SPECIFICATION2	NO_FILES_2
$NAME_SPECIFICATION3	NO_FILES_3
$NEG_BASED_INT	16#FFFE#
$NEW_MEM_SIZE	TEST_WITHDRAWN
$NEW_STOR_UNIT	16
$NEW_SYS_NAME	TEST_WITHDRAWN
$PAGE_TERMINATOR	TEST_WITHDRAWN
$RECORD_DEFINITION	TEST_WITHDRAWN
$RECORD_NAME	TEST_WITHDRAWN
$TASK_SIZE	32
$TASK_STORAGE_SIZE	4096
$TICK	0.0001
$VARIABLE_ADDRESS	SYSTEM.MAKE_ADDRESS(16#0020#)
$VARIABLE_ADDRESS1	SYSTEM.MAKE_ADDRESS(16#0024#)
$VARIABLE_ADDRESS2	SYSTEM.MAKE_ADDRESS(16#0028#)
$YOUR_PRAGMA	TEST_WITHDRAWN

APPENDIX B

COMPILATION SYSTEM OPTIONS

COMPILER OPTIONS

The compiler 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.

ADA1750A

Parameters

input-file-spec

Ada source file(s) or a batching compiler script file specification. The user is responsible for the order of the compilation if multiple source files are specified. Only one script file may be specified. The default filetype for the source file is .ADA and for the batching compiler script is .SCR.

ADA1750A

Command_Qualifiers

/ASSEMBLER

The assembler listing is written to the listing file.

/LIST
/LIST[=list-file-spec]
/NOLIST

This qualifier controls the writing of source listing. If not specified an interspersed source listing is generated only if the compilation has error and/or warning message(s). Specify LIST to generates a source listing regardless of the compilation status. The listing file defaults to the source filename with character "L" prepended to the first two characters of the source filetype (e.g TEST.ADA will produce TEST.LAD). If specified the listing file will be placed in the file given by list-file-spec. NOLIST generates only the messages if any without any source listing.

/MAX_CONTINUE
/MAX_CONTINUE[=max-continue-spec]

The maximum return status allowable for continuing compilation. The valid max_continue-spec values are WARNING, ERROR, FATAL, or INTERNAL.
The default value is ERROR.

/OPTIMIZE
/OPTIMIZE
/OPTIMIZE[=optimization-value]

Sets the level of optimization for the code generated. An integer value 0 through 10 may be selected for the optimization-value. 0 request no optimization and 10 request the highest level of optimization. The default optimization-value is 10.

/OPTIONS_FILE
/OPTIONS_FILE=opt-file-spec

The file specification of the options file for the compiler phases. The options file must have the format:

```
option=value[,...]
        .
        .
        .
option=value[,...]
```

/PTR_FILE
/PTR_FILE=ptr-file-spec

The file specification of the pointer file for the Ada library. If this qualifier is not specified the compiler searches for the file definition of the pointer file in the logical name PTR_FILE. If the logical name PTR_FILE is not defined the pointer file defaults to the file ADALIB. in the current directory.

/SCRIPT

The qualifier specifies that the input-file-spec is a batching compiler script file specification.

/SOURCE_RECONSTRUCT

The qualifier specifies that sufficient information be retained in the program library to reconstruct the source and/or assembler listing.

/STARTWITH
/STARTWITH[=phase-spec]

The compilation will start at the specified phase. The valid values for the phase-spec are TB, SEM, GEN, STO, EXP, FLOW, or CG. The default value is TB.

/STATISTICS

Statistics such as the number of instructions generated will be written into the listing file.

/STOPAFTER
/STOPAFTER[=phase-spec]

The compilation will stop at the specified phase. The valid values for the phase-spec are TB, SEM, GEN, STO, EXP, FLOW, or CG. The default value is CG.

/TABLE_DIRECTORY
/TABLE_DIRECTORY=table-directory-spec

The qualifier specifies the directory which contains the code generation tables. If this qualifier is not specified the compiler searches for the table directory definition in the logical name TABLE_DIRECTORY.

/XMEM
/XMEM (default)
/NOXMEM

Enables or disables the expanded memory feature.

LINKER OPTIONS

The 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 linker documentation and not to this report.

LNK1750A

Parameters

unit-name

Unit name for which a load module is to be generated.

LNK1750A

Command_Qualifiers

/ALPHA

An alphabetically ordered link map is produced.

/COM_FILE
/COM_FILE=command-file-spec

The name of the linker input command file. See User's Manual for description of this and other linker input commands.

/DEBUG_FILE
/DEBUG_FILE=debug-file-spec

The name of the file to contain any user-requested listings. If /DEBUG_FILE is not specified the default file is unit-name.LBL in the current directory.

/PTR_FILE
/PTR_FILE=ptr-file-spec

The file specification of the pointer file for the Ada library. If this qualifier is not specified the pointer file defaults to ADALIB. in the current directory.

/MAP

An ascending address order link map is produced.

/OUTPUT
/OUTPUT=output-file-spec

The name of the output load module file. If /OUTPUT is not specified the default file is unit-name.SO in the current directory.

/XREF

The cross reference listing is produced.

Appendix F. Implementation Dependencies

This section constitutes Appendix F of the Ada LRM for this implementation. Appendix F from the LRM states:

> *The Ada language allows for certain machine-dependencies in a controlled manner. No machine-dependent syntax or semantic extensions or restrictions are allowed. The only allowed implementation-dependencies correspond to implementation-dependent pragmas and attributes, certain machine-dependent conventions as mentioned in Chapter 13, and certain allowed restrictions on representation clauses.*
>
> *The reference manual of each Ada implementation must include an appendix (called Appendix F) that describes all implementation-dependent characteristics. The Appendix F for a given implementation must list in particular:*
>
> 1. *The form, allowed places, and effect of every implementation-dependent pragma.*
> 2. *The name and the type of every implementation-dependent attribute.*
> 3. *The specification of the package SYSTEM (see 13.7).*
> 4. *The list of all restrictions on representation clauses (see 13.1).*
> 5. *The conventions used for any implementation-generated name denoting implementation-dependent components (see 13.4).*
> 6. *The interpretation of expressions that appear in address clauses, including those for interrupts (see 13.5).*
> 7. *Any restriction on unchecked conversions (see 13.10.2).*
> 8. *Any implementation-dependent characteristics of the input-output packages (see 14).*

In addition, the present section will describe the following topics:

9. Any implementation-dependent rules for termination of tasks dependent on library packages (see 9.4:13).
10. Other implementation dependencies.

F.1 Pragmas

This section describes the form, allowed places, and effect of every implementation-dependent pragma.

F.1.1 Pragmas LIST, PAGE, PRIORITY, ELABORATE

Pragmas LIST, PAGE, PRIORITY and ELABORATE are supported exactly in the form, in the allowed places, and with the effect as described in the LRM.

F.1.2 Pragma SUPPRESS

Form: Pragma SUPPRESS (identifier)
where the identifier is that of the check that can be omitted. This is as specified in LRM B(14), except that suppression of checks for a particular name is not supported. The name clause (ON=>name), if given, causes the entire pragma to be ignored.

The compiler will support suppression of all checks. In particular, the suppression of the following run-time checks, which correspond to situations in which the exceptions CONSTRAINT_ERROR, STORAGE_ERROR, PROGRAM_ERROR, or NUMERIC_ERROR may be raised, are supported:

ACCESS_CHECK
DISCRIMINANT_CHECK
INDEX_CHECK
LENGTH_CHECK
RANGE_CHECK
STORAGE_CHECK
ELABORATION_CHECK
DIVISION_CHECK
OVERFLOW_CHECK

Allowed Places: As specified in LRM B(14) : SUPPRESS.

Effect: Permits the compiler not to emit code in the unit being compiled to perform various checking operations during program execution. The supported checks have the effect of suppressing the specified check as described in the LRM.

Use: To suppress Ada's builtin error-checking.

F.1.3 Pragma SUPPRESS_ALL

Form: As specified in LRM B(14) : SUPPRESS
except that the suppression of checks for a particular name is not supported so the name clause (ON=>name), if given, causes the entire pragma to be ignored.

Allowed Place: As specified in LRM B(14) : SUPPRESS

Effect: The implementation-defined pragma SUPPRESS_ALL has the effect of the specification of a pragma SUPPRESS for each of the supported checks.

Use: To suppress Ada's builtin error-checking.

F.1.4 Pragma INLINE

Form: Pragma INLINE (SubprogramNameCommaList)

Allowed Places: As specified in LRM B(4) : INLINE

Effect: If the subprogram body is available, and the subprogram is not recursive, the code is expanded in-line at every call site and is subject to all optimizations.

The stack-frame needed for the elaboration of the inline subprogram will be allocated as a temporary in the frame of the containing code.

Parameters will be passed properly, by value or by reference, as for non-inline subprograms. Register-saving and the like will be suppressed. Parameters may be stored in the local stack-frame or held in registers, as global code generation allows.

Exception-handlers for the INLINE subprogram will be handled as for block-statements.

Use: This pragma is used either when it is believed that the time required for a call to the specified routine will in general be excessive (e.g. for frequently called subprograms) or when the average expected size of expanded code is thought to be comparable to that of a call.

F.1.5 Pragma INTERFACE

Form: Pragma INTERFACE (language_name, subprogram_name)
where the language_name must be an enumeration value of the type SYSTEM.Supported_Language_Name (see Package SYSTEM below).

Allowed Place: As specified in LRM B(5) : INTERFACE.

Effect: Specifies that a subprogram will be provided outside the Ada program library and will be callable with a specified calling interface. Neither an Ada body nor an Ada body_stub may be provided for a subprogram for which INTERFACE has been specified.

Use: Use with a subprogram being provided via another programming language and for which no body will be given in any Ada program. See also the LINK_NAME pragma.

For a pragma INTERFACE (AIE_ASSEMBLER) subprogram, standard Ada calling conventions will be followed, but no Ada body is expected or allowed. The standard Ada calling conventions for the compiler are described in Appendix C.

F.1.6 Pragma LINK_NAME

Form: Pragma LINK_NAME (subprogram_name, link_name)

Allowed Places: As specified in LRM B(5) for pragma INTERFACE.

Effect: Associates with subprogram subprogram_name the name link_name

Syntax: The value of link_name must be a character string literal.

Use: To allow Ada programs, with help from INTERFACE pragma, to reference non-Ada subprograms. Also allows non-Ada programs to call specified Ada subprograms.

F.1.7 Pragma CONTROLLED

Form: Pragma CONTROLLED (AccessTypeName)

Allowed Places: As specified in LRM B(2) : CONTROLLED.

Effect: Ensures that heap objects are not automatically reclaimed. but are explicitly reclaimable by use of unchecked_deallocation.

Since no automatic garbage collection is ever performed. this pragma currently has no effect.

F.1.8 Pragma PACK

Form: Pragma PACK (type_simple_name)

Allowed Place: As specified in LRM 13.1(12)

Effect: Records or arrays are allowed their minimal number of storage units as provided for by their own representation and/or packing.

For arrays, elements are bit-packed with a size that is evenly divisible into the storage unit size. The elements are then packed into the minimal number of storage units.

For records, scalar components smaller than a word are bit-packed. The components are then packed into the minimal number of storage units.

Use: Pragma PACK is used to reduce storage size. This can allow records and arrays, in some cases, to be passed by value instead of by reference.

Size reduction usually implies an increased cost of accessing components. The decrease in storage size may be offset by increase in size of accessing code and by slowing of accessing operations.

F.1.9 Pragma REMOTE

Form: Pragma REMOTE or Pragma REMOTE (library_unit_name)

Allowed Places: Within library units which are package specifications or immediately following library units which are subprogram specifications. Note that this pragma is only supported when the RTS containing the expanded memory support is used. When the 64K RTS is used, this pragma is ignored.

Effect: Pragma REMOTE without a parameter specifies that the library unit is to be located in a new group in the 1750a memory map. The library unit is then known as the *parent* of that group and may then be specified as the parameter to the pragma.

Pragma REMOTE with the "library_unit_name" parameter specifies that the unit which contains the pragma be placed into the same group as the specified "library_unit_name."

Use: The pragma is used to distribute the library units of a large program into the different address states of a 1750A containing support for the expanded memory option (via page registers). Pragma REMOTEs are used to group related units together so as to minimize remote references and calls.

The following simple example shows how two packages can be specified to be placed in the same group.

```
package A is
  pragma REMOTE;
end A;

with A;
package B is
  pragma REMOTE(A);
end B;
```

F.1.10 Pragma INTERRUPT

Form: Pragma INTERRUPT (Handler_Routine, Interrupt_Level)

Allowed Places: Library unit specifications.

Effect: Unmasks that level of 1750a interrupt and binds a parameterless procedure to the interrupt such that when the Interrupt_Level occurs, the Handler_Routine is invoked as quickly as possible. After the procedure completes, execution is resumed from where it was prior to the interrupt.

Use: Provides a faster interrupt mechanism than Ada interrupt entries.

```
with Interrupt_User_Support;
package Handlers is
   procedure User_0;     --| no parameters allowed
   pragma Interrupt(User_0, Interrupt_User_Support.Interrupt_Level_Type(2));
end Handlers;
```

F.1.11 Pragmas SYSTEM_NAME, STORAGE_UNIT, MEMORY_SIZE, SHARED

These pragmas are not supported and are ignored.

F.1.12 Pragma OPTIMIZE

Pragma OPTIMIZE is ignored; optimization is always enabled.

F.2 Implementation-dependent Attributes

This section describes the name and the type of every implementation-dependent attribute.

There are no implementation defined attributes. These are the values for certain language-defined, implementation-dependent attributes:

```
Type INTEGER.
     INTEGER'SIZE                 = 16 -- bits.
     INTEGER'FIRST                = - (2**15)   -- - 32,768
     INTEGER'LAST                 =  (2**15-1) --  32,767


Type FLOAT.
     FLOAT'SIZE                   = 32 -- bits.
     FLOAT'DIGITS                 = 6
     FLOAT'MANTISSA               = 21
     FLOAT'EMAX                       = 84
     FLOAT'EPSILON                    = 2.0**(-20)
     FLOAT'SMALL                      = 2.0**(-85)
     FLOAT'LARGE                      = (2.0**84)*(1.0-2.0**(-21))
     FLOAT'MACHINE_ROUNDS         =  false
     FLOAT'MACHINE_RADIX          =    2
     FLOAT'MACHINE_MANTISSA       =   24
     FLOAT'MACHINE_EMAX           =  127
     FLOAT'MACHINE_EMIN           =  -128
     FLOAT'MACHINE_OVERFLOWS      =  true
     FLOAT'SAFE_EMAX                  =  127
     FLOAT'SAFE_SMALL             = 2#0.1000000000000000000000#E-127
     FLOAT'SAFE_LARGE             = 2#0.1111111111111111111111#E127

Type LONG_FLOAT.
     LONG_FLOAT'SIZE     = 48 -- bits.
     LONG_FLOAT'DIGITS   = 9
     LONG_FLOAT'MANTISSA = 31
     LONG_FLOAT'EMAX     = 124
     LONG_FLOAT'EPSILON  = 2.0**(-30)
     LONG_FLOAT'SMALL    = 2.0**(-125)
     LONG_FLOAT'LARGE    = 2.0**(124)*(1.0-2.0**(-31))
     LONG_FLOAT'MACHINE_ROUNDS    = false
     LONG_FLOAT'MACHINE_RADIX     =    2
     LONG_FLOAT'MACHINE_MANTISSA  =   40
     LONG_FLOAT'MACHINE_EMAX      =  127
     LONG_FLOAT'MACHINE_EMIN      = -128
     LONG_FLOAT'MACHINE_OVERFLOWS = true
```

```
        LONG_FLOAT'SAFE_EMAX      = 127
        LONG_FLOAT'SAFE_SMALL     =
                2#0.11111111111111111111111111111111111111111#E-127
        LONG_FLOAT'SAFE_LARGE     =
                2#0.11111111111111111111111111111111111111111#E127

    Type DURATION.
        DURATION'DELTA            = 2.0**(-6) seconds
        DURATION'FIRST            = -86400.0 seconds
        DURATION'LAST             = 86400.0 seconds
        DURATION'SMALL            = 2.0**(-6)  - 15.6 ms Approx.

    Type PRIORITY.
        PRIORITY'FIRST            = -127
        PRIORITY'LAST             =  127
```

F.3 Package SYSTEM

There are two Run-time Systems which are available for the 1750A compiler. One is for the 1750A with 64K words of memory and the other is for a 1750A with the expanded memory option. See the User's Manual for details. Each RTS has its own package System.

F.3.1 Expanded Memory RTS

```
package System is

   type Address is private;    -- "=", "/=" defined implicitly;
   type Name is (UTS, MVS, CMS, Prime50, Sperry1100, MIL_STD_1750A);

   System_Name : constant Name := MIL_STD_1750A;

   Storage_Unit : constant := 16;
   Memory_Size : constant := 2 * 16 * 2**16;
    -- In storage units     I+D   groups   64K.

   -- System-Dependent Named Numbers:

   Min_Int : constant := Integer'POS(Integer'FIRST);
   Max_Int : constant := Integer'POS(Integer'LAST);
   Max_Digits : constant := 9;
   Max_Mantissa : constant := 31;
   Fine_Delta   : constant := 2.0**(-31);
   Tick : constant := 0.0001;

   -- Other System-Dependent Declarations

   subtype Priority is Integer range -127..127;

-- ..................................................................  --
Implementation-dependent  additions  to  package  System  --  ..........
-- ..................................................

   Null_Address : constant Address;
    -- Same bit pattern as "null" access value
    -- This is the value of 'ADDRESS for named numbers.
    -- The 'ADDRESS of any object which occupies storage
    --  is NOT equal to this value.

   Address_Size : constant := 32;
    -- Number of bits in Address objects, = Address'SIZE, but static.

   subtype Group_Number is integer range -1 .. 15;
    -- -1 means global area.  Other values are address states.

   Global_Group_Number : constant Group_Number := -1;

   subtype Address_Segment is Group_Number;

   Address_Segment_Size : constant := 16;
    -- Number of bits in address segment, =Address_Segment'SIZE,
    -- but static

   type Address_Offset is new Integer;  -- Used for address arithmetic

   subtype Logical_Address is Address_Offset;
```

```
   Null_Logical_Address : constant Logical_Address := 0;
      -- This is the value of a null access type.

   -- Interrupt level for the 1750a, 0 is the highest level Type
   Interrupt_Level_Type is range 0 .. 15; for Interrupt_Level_Type'Size use
   16;


      type Supported_Language_Name is ( -- Target dependent
         -- The following are "foreign" languages:
         AIE_ASSEMBLER, -- NOT a "foreign" language - uses AIE RTS
         UNSPECIFIED_LANGUAGE
      );

      -- Most/least accurate built-in integer and float types

      subtype Longest_Integer is Standard.Integer;
      subtype Shortest_Integer is Standard.Integer;

      subtype Longest_Float is Standard.Long_Float;
      subtype Shortest_Float is Standard.Float;


      function Offset_Of (              --| Offset portion of an Address.
            Addr        : in Address
            ) return      Address_Offset;
         --| Returns the offset portion of an Address.
      pragma Inline(Offset_Of);

      function Segment_Of (         --| Segment portion of an Address.
            Addr        : in Address
            ) return      Address_Segment;
         --| Return segment portion (group number) of an Address.
      pragma Inline(Segment_Of);

      function Make_Address (    --| Compose an Address from parts.
            Segment     : in Address_Segment;
            Offset      : in Address_Offset
            ) return      Address;
         --| Form an Address from a segment (group number) and an offset.
      pragma Inline(Make_Address);

   private

      MAX_FIX : constant := 2.0**Max_Mantissa -1.0;
      MIN_FIX : constant := (Min_Int + Max_Int) * 1.0 - MAX_FIX;

      type Address is delta 1.0 range MIN_FIX .. MAX_FIX;
      -- Also see type Composite_Address.
      -- This is an address state and a logical (simple) address.  We use
      -- fixed point (two words) so that an address is not treated as a
      -- composite object by the compiler (for example it will be passed
      -- by value rather than by reference).  We specify maximum range to
      -- avoid constraint checks.  We sacrifice some checking in order to
      -- get speed (in-line code for address arithmetic).  Thus the user
      -- will be responsible for ensuring that address calculations have
      -- the appropriate operands.

      -- One mechanism by which the user may enforce more rigorous type
      -- checking is to write another package through which all address
      -- calculations must pass which provides the appropriate checks.
      -- The current implementation has been chosen because it provides
      -- efficiency and does not preclude stricter type checking (i.e.
      -- the above mentioned package).  The type checking which is not
      -- provided is noted below

      -- Adding two addresses will produce unpredictable results
```

```
    -- Subtracting two addresses which are in different groups will
    -- produce unpredictable results, and may raise Constraint_Error.

    Null_Address : constant Address := 0.0;

    type Composite_Address is record
        -- Used solely for accessing Address segent and offset.
        Segment : Address_Segment;
        Offset  : Address_Offset;
    end record;

end System;
```

F.3.2 64K RTS

```
package System is

    type Address is private;    -- "=", "/=" defined implicitly;
    type Name is (UTS, MVS, CMS, Prime50, Sperry1100, MIL_STD_1750A);

    System_Name : constant Name := MIL_STD_1750A;

    Storage_Unit : constant := 16;
    Memory_Size : constant := 2**16;
    -- In storage units.

    -- System-Dependent Named Numbers:

    Min_Int : constant := Integer'POS(Integer'FIRST);
    Max_Int : constant := Integer'POS(Integer'LAST);
    Max_Digits : constant := 9;
    Max_Mantissa : constant := 31;
    Fine_Delta : constant := 2.0**(-31);
    Tick : constant := 0.0001;

    -- Other System-Dependent Declarations

    subtype Priority is Integer range -127..127;

--------------------------------------------------------------------
-- Implementation-dependent additions to package System --
--------------------------------------------------------------------

    Null_Address : constant Address;
    -- Same bit pattern as "null" access value
    -- This is the value of 'ADDRESS for named numbers.
    -- The 'ADDRESS of any object which occupies storage
    --   is NOT equal to this value.

    Address_Size : constant := 16;
    -- Number of bits in Address objects, = Address'SIZE, but static.

    subtype Group_Number is integer range -1 .. 15;
    -- -1 means global area.  Other values are address states.

    Global_Group_Number : constant Group_Number := -1;

    subtype Address_Segment is Group_Number;

    Address_Segment_Size : constant := 16;
    -- Number of bits in address segment, =Address_Segment'SIZE,
    -- but static.
```

```
   type Address_Offset is new Integer;  -- Used for address arithmetic

   subtype Logical_Address is Address_Offset;
   Null_Logical_Address : constant Logical_Address := 0;
   -- This is the value of a null access type.

   type Interrupt_Level_Type is range 0..15;
   for Interrupt_Level_Type'size use 16;

   type Supported_Language_Name is (  -- Target dependent
      -- The following are "foreign" languages:
      AIE_ASSEMBLER, -- NOT a "foreign" language - uses AIE RTS
      UNSPECIFIED_LANGUAGE
   );

   -- Most/least accurate built-in integer and float types

   subtype Longest_Integer is Standard.Integer;
   subtype Shortest_Integer is Standard.Integer;

   subtype Longest_Float is Standard.Long_Float;
   subtype Shortest_Float is Standard.Float;


   subtype Normalized_Address_Offset is
     Address_Offset;  --bal range 0 .. Address_Segment_Size - 1;
      -- Range of address offsets returned by Offset_Of

   function Offset_Of (              --| Offset portion of an Address.
         Addr       : in Address
         ) return     Address_Offset;
      --| Returns the offset portion of an Address.
   pragma inline(Offset_Of);


   function Segment_Of (         --| Segment portion of an Address.
         Addr       : in Address
         ) return     Address_Segment;
      --| Return segment portion (group number) of an Address.
   pragma inline(Segment_Of);


   function Make_Address (    --| Compose an Address from parts.
         Segment    : in Address_Segment;
         Offset     : in Address_Offset
         ) return     Address;
      --| Form an Address from a segment (group number) and an offset.
   pragma inline(Make_Address);



function "+"(addr : Address; offset : Address_Offset) return Address;
function "+"(offset : Address_Offset; addr : Address) return Address;
   -- Provide addition between addresses and
   --  offsets.  May cross segment boundaries on targets where
   --  objects may span segments.
   -- On other targets, CONSTRAINT_ERROR will be raised when
   --  Offset_Of(addr) + offset not in Normalized_Address_Offset.

function "-"(left, right : Address) return Address_Offset;
   -- May exceed Segment_Size on targets where objects may
   --  span segments.
   -- On other targets, Constraint_Error
   --  will be raised if Segment_Of(left) /= Segment_Of(right)

function "-"(addr : Address, offset   Address_Offset) return
  Address;
```

```
        -- Provide subtraction of addresses and offsets.
        -- May cross segment boundaries on targets where
        --   objects may span segments.
        -- On other targets, CONSTRAINT_ERROR will be raised when
        --   Offset_Of(addr) - offset not in Normalized_Address_Offset.

    private

        Max_Fix : constant := 2.0**Max_Mantissa - 1.0;
        Min_Fix : constant := (Min_Int + Max_Int) * 1.0 - Max_Fix;

        type Address is access Integer;

            -- Note: The designated type here (Integer) is irrelevant.
            --       Address is made an access type simply to guarantee it has
            --       the same size as access values, which are single addresses.
            --       Allocators of type Address are NOT meaningful.


        Null_Address : constant Address := null;

    end System;
```

F.4 Representation Clauses

This section describes the list of all restrictions on representation clauses.

"NOTE: An implementation may limit its acceptance of representation clauses to those that can be handled simply by the underlying hardware.... If a program contains a representation clause that is not accepted [by the compiler], then the program is illegal." (LRM 13.1(10)).

Those restrictions which are defined by the LRM are not listed. A description of the effect of the representation clause is also included where appropriate.

a. Length clauses:

- Size specification: T'SIZE.

The size specification may be applied to a type T or first-named subtype T which is an access type, a scalar type, an array type or a record type.

AI-00536/07 has altered the meaning of a size specification. In particular, the statement from the LRM 13.2.a that the expression in the length clause specifies an upper bound for the number of bits to be allocated to objects of the type is incorrect. Instead, the expression specifies the exact size for the type. Objects of the type may be larger than the specified size for padding. Note that the specified size is not used when the type is used as a component of a record type and a component clause specifying a different size is given.

If the length clause can not be satisfied by the type, an error message will be generated.

The supported values of the size expression are explained for the types as follows. If the value of the size expression is not supported, an error message will be generated.

access type: the only size supported is 16.

integer or enumeration type: minimum size supported is 1, the maximum size that is supported is 16 the size of the largest predefined integer type. Biased representation is not supported.

fixed point: minimum size supported is 1, the maximum size that is supported is 32. Biased representation is not supported.

floating point type: the sizes supported are 32 and 48. Note that the size must satisfy the DIGITS requirement. No support is provided for shortened mantissa and/or exponent lengths.

record type: if the size of the unpacked type is greater than the specified size of the length clause, an implicit pragma pack will be assumed on the record type. If the size of the implicit pragma packed record type is still greater than the specified size of the length clause, an error will be generated. (See also Pragma Pack F.1.8 and Record Representation Clauses F.4.c).

array type: if the size of the unpacked array type is greater than the size clause expression, an implicit pragma pack will be assumed on the array type. If the size of an implicit pragma packed array type is still greater than the size expression clause, an error will be generated.

- Specification of collection size: T'STORAGE_SIZE.

  The effect of the specification of collection size is that a contiguous area of the required size will be allocated for the collection. If an attempt to allocate an object within the collection requires more space than currently exists in the collection, STORAGE_ERROR will be raised. Note that this space includes the header information.

- Specification of storage for a task activation: T'STORAGE_SIZE.

  The value specified by the length clause will be the total size of the stacks allocated for the task, rounded up to a multiple of the storage page or "chunk" size (which is 512 words). The default stack size for each task is 1028 words.

- Specification of small for a fixed point type : T'SMALL

  The value of T'SMALL is restricted to composite powers of 2 and 5 (e.g. 2, 5, 10).

b. Enumeration Representation Clauses:

Values must be in the range of MIN_INT .. MAX_INT.

c. Record-representation-clause:

An alignment clause forces each record of the given type to be allocated at a starting address that is a multiple of the value of the given expression. Allowed alignment value is 1 (SU aligned).

The range of bits specified has the following restrictions: if the starting bit is 0, there is no limit on the value for the ending bit; if the starting bit is greater than 0, then the ending bit must be less than or equal to 15.

Record components, including those generated implicitly by the compiler,

whose locations are not given by the representation-clause, are laid out by the compiler following all the components whose locations are given by the representation-clause. Such components of the invariant part of the record are allocated to follow the user-specified components of the invariant part, and such components in any given variant part are allocated to follow the user-specified components of that variant part.

The actual size of the record object (including its use as a component of a record or array type) will always be a multiple of storage units (e.g 16,32, etc. bits) with padding added to the end of the record, if necessary. User-specified ranges must contain at least the minimal number of bits required to represent a (bit-packed) object of the corresponding type; e.g. to represent an integer type with a range of 0..15, at least 4 bits must be specified in the record representation specification range.

d. Address clauses:

Address clauses are allowed for objects (variable or constant) and for sub-programs to which a pragma INTERFACE applies. Address clauses are not allowed for packages or tasks. The interpretation of the value of an address clause is described in F.6.

F.5 Implementation-dependent Components

This section describes the conventions used for any implementation-generated name denoting implementation-dependent components.

There are no implementation-generated names denoting implementation-dependent (record) components, although there are, indeed, such components. Hence, there is no convention (or possibility) of naming them and, therefore, no way to offer a representation clause for such components.

Note: Records containing dynamic-sized components will contain (generally) unnamed offset components which will "point" to the dynamic-sized components stored later in the record. There is no way to specify the representation of such components.

F.6 Address Clauses

This section describes the interpretation of and restrictions upon expressions that appear in address clauses.

The address specified by the simple_expression of an address clause,

```
for simple_name use at simple_expression ;
```

should be a call on System.Make_Address, for example,

```
for ABC use at System.Make_Address( Group, Offest ) ;
```

System.Address is a private type. It is recommended that a user utilize the routines provided in package System to manipulate Address values. These routines are "inlined" to make the use of them faster so that Unchecked_Conversion between addresses and access types is unnecessary. In fact, for the Expanded Memory RTS, addresses are represented by 32 (rather than 16) bits and Unchecked_Conversion is not available to convert between a 32 bit System.Address and a 16 bit access value.

An address clause may be applied to an object (variable or constant), to a subprogram, or to a task entry. It may not be applied to a package or task.

When an address clause is applied to an object, the address value will be interpreted as an address in memory which will be used for all reads and updates of the object. No storage space will be allocated for the object by the compiler (or pre-linker).

When an address clause is applied to a subprogram, there must be a pragma INTERFACE associated with the subprogram to provide the body. No storage space is allocated by the compiler and the PLACE command of the linker should be used to locate the interfaced unit where the address clause specified it to be.

For interrupt entries, the addresses of the 1750A interrupt table are used to specify the interrupt level which will call the entry.

F.7 Unchecked Conversions

This section describes any restrictions on unchecked conversions.

The source and target must both be of a staticly sized type (other than a discriminated record type) and both types must have the same static size.

F.8 Input-Output

This section describes implementation-dependent characteristics of the input-output packages.

The 1750A is assumed to operate without an operating system and without external I/O devices other than the console device which supports Text_IO for STANDARD_INPUT and STANDARD_OUTPUT. The predefined exception USE_ERROR will be raised if an attempt is made to open any external file or use the console for other than Text_IO.

F.9 Tasking

This section describes implementation-dependent characteristics of the tasking run-time packages.

Even though a main program completes and terminates (its dependent tasks, if any, having terminated), the elaboration of the program as a whole continues until each task dependent upon a library unit package has either terminated or reached an open terminate alternative. See LRM 9.4(13).

Ada "delay statements" are implemented for the 1750A target using the optional timer referred to as Timer B. Timer B is also used to support the implementation of Package Calendar.

F.10 Other Matters

This section describes other implementation-dependent characteristics of the system.

a. Package Machine_Code
   Will not be provided.

b. Order of compilation of generic bodies and subunits (LRM 10.3:9):
   Body and subunits of generic must be in the same compilation as the specification if instantiations precede them (see AI-00257/02).