REPOF

GE

Form Approved
OPM No. 0704-0188

Public reporting burden for this collection needed, and reviewing the collection of In Headquarters Service, Directorate for Inf Management and Budget, Washington, D

AD-A246 508

r reviewing instructions, searching existing data sources gathering and maintaining the data of this collection of information, including suggestions for reducing this burden, to Washington rlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENCY USE ONLY (Leav

3. REPORT TYPE AND DATES COVERED
Final: 28 Oct 1991 to 01 Jun 1993

2

4. TITLE AND SUBTITLE
TeleSoft, TeleGen2, Ada Development System, for VAX to 1750A, Version 3.25, MicroVAX 3800 under VAX/VMS Version V5.4 (Host) to MIL-STD-1750A ECSPO ITS RAID Simulator, Version 6.0, (Target), 911028I1.11229

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, Federal Republic of Germany

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

8. PERFORMING ORGANIZATION REPORT NUMBER
IABG-VSR 098

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

DTIC
ELECTE
FEB 28 1992
S D
D

11. SUPPLEMENTARY NOTES

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

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
TeleSoft, TeleGen2, Ada Development System, for VAX to 1750A, Version 3.25, Wright-Patterson AFB, MicroVAX 3800 under VAX/VMS Version V5.4 (Host) to MIL-STD-1750A ECSPO ITS RAID Simulator, Version 6.0, (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	

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-10-28.

Compiler Name and Version:	**TeleGen2™ Ada Development System for VAX to 1750A, Version 3.25**
Host Computer System:	**MicroVAX 3800 under VAX/VMS Version V5.4**
Target Computer System:	**MIL-STD-1750A ECSPO ITS RAID Simulator, Version 6.0**

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

As a result of this validation effort, Validation Certificate #911028I1.11229 is awarded to TeleSoft. This certificate expires on 01 June 1993.

This report has been reviewed and is approved.

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

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

AVF Control Number: IABG-VSR 098
31 October 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 911028I1.11229
TeleSoft
TeleGen2™ Ada Development System
for VAX to 1750A, Version 3.25
MicroVAX 3800 under VAX/VMS Version V5.4 =>
MIL-STD-1750A ECSPO ITS RAID Simulator,
Version 6.0

== based on TEMPLATE Version 91-05-08 ==

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

DECLARATION OF CONFORMANCE

Customer:	TeleSoft 5959 Cornerstone Court West San Diego CA USA 92121
Ada Validation Facility:	IABG, Dept. ITE W-8012 Ottobrunn Germany
ACVC Version:	1.11

Ada Implementation:

Ada Compiler Name and Version:	TeleGen2™ Ada Development System for VAX to 1750A,Version 3.25
Host Computer System:	MicroVAX 3800 (under VAX/VMS Version V5.4)
Target Computer System:	MIL-STD-1750A ECSPO ITS RAID Simulator, Version 6.0

Customer's Declaration

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

Date: 10/28/91

TELESOFT
Raymond A. Parra
Vice President
General Counsel

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
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 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 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 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				

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.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

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 ten; this implementation does not support such values for 'SMALL.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y checks for a predefined fixed-point type other than DURATION.

CA2009C and CA2009F check whether a generic unit can be instantiated before its body (and any of its subunits) is compiled; this implementation creates a dependence on generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3.)

LA3004B, EA3004D, and CA3004F check pragma INLINE for functions; this implementation does not support pragma INLINE for functions.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

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.

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.)

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)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 17 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.

B71001Q	BA1001A	BA2001C	BA2001E	BA3006A
BA3006B	BA3007B	BA3008A	BA3008B	BA3013A

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 compilation of that unit's body; as allowed by AI-00408 and AI-00506, the compilation of the generic unit bodies makes the compilation unit that contains the instantiations obsolete.

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).

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.

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:

TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-3731, USA
(619) 457-2700

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.

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	3463	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	63	
d) Non-Processed I/O Tests	264	
e) Non-Processed Floating-Point Precision Tests	285	
f) Total Number of Inapplicable Tests	612	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

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 magnetic tape were loaded onto the host computer via DECNET.

After the test files were loaded onto the host computer, the full set of tests except 264 I/O Tests and 285 Floating-Point Precision Tests were processed by the Ada implementation.

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

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 are given in the following section, which was supplied by the customer. The third page displays the contents of the additional linker options file, which is referenced through

```
options=bigacvc.opt
```

in the link command.

Compiler Option Information

B Tests and E Tests:

tsada/ 1750a/ ada/ monitor/ optimize/ enable/ list/ virtual_space=10000 <test_name>

option	description
tsada	invoke Ada compilation system
1750a	specify the 1750a target
ada	compile the test
monitor	verbose mode, output pass info during compilation
optimize	optimize code generated
enable	enable traceback for unexpected errors
list	generate compilation listing
virtual_space	memory allocation control
<test_name>	name of Ada source file to be compiled

All Execution Tests except E Tests:

tsada/ 1750a/ ada/ monitor/ optimize/ virtual_space=10000/ nolist <test_name>

option	description
tsada	invoke Ada compilation system
1750a	specify the 1750a target
ada	compile the test
monitor	verbose mode, output pass info during compilation
optimize	optimize code generated
virtual_space	memory allocation control
nolist	do not generate compilation listing
<test_name>	name of Ada source file to be compiled

BIND:

tsada/ 1750a/ lktools/ bind/ monitor/ virtual_space=10000 <main_unit>

option	description
tsada	invoke Ada compilation system
1750a	specify the 1750a target
bind	bind the main unit
monitor	verbose mode, output pass info during compilation
virtual_space	memory allocation control
<main_unit>	name of main compilation unit

LINK:

tsada/ 1750a/ lktools/ link/ monitor/ map/ options=bigacvc.opt/ enable/ -
virtual=10000 <main_unit>

option	description
tsada	invoke Ada compilation system
1750a	specify the 1750a target
link	link the test
monitor	verbose mode, output pass info during compilation
map	generate link map listing
options	specify linker options file
enable	enable traceback for unexpected errors
virtual_space	memory allocation control
<main_unit>	name of main compilation unit

```
--
-- Linker options file used for ACVCs requiring more than 64k.  The available
-- data space is split as follows:
--
--  4k Page 0 interrupt handlers, runtime
-- 28k Heap space (mostly used for task stacks)
-- 20k Stack space
-- 12k Data Region (literal and data control sections)
--
--
-- Memory size is 128k for this test.
--
MEMORY/SIZE=131072
--
-- Locate boot routine at page 0
--
LOCATE/CONTROL_SECTION=LTRL/COMPONENT_NAME=PSP0/OFM/AT=%X0000
LOCATE/CONTROL_SECTION=CODE/COMPONENT_NAME=PSP0/OFM/AFTER_SECTION=LTRL/-
AFTER_COMPONENT=PSP0
--
-- Define memory regions
--
REGION/LOW=%X0000/HIGH=%X0FFF PAGE0
REGION/LOW=%X1000/HIGH=%XCFFF MMRYREG
REGION/LOW=%XD000/HIGH=%XFFFF DATAREG
REGION/LOW=%X11000/HIGH=%X1FFFF CODEREG
--
-- Place the control sections in the appropriate regions.
--
LOCATE/CONTROL_SECTION=CODE/IN=CODEREG
LOCATE/CONTROL_SECTION=GATE/IN=CODEREG
LOCATE/CONTROL_SECTION=LTRL/IN=DATAREG
LOCATE/CONTROL_SECTION=DATA/IN=DATAREG
LOCATE/CONTROL_SECTION=DESC/IN=DATAREG
LOCATE/IN=CODEREG
--
-- Specify the required logical address constants.
--
DEFINE/XMEMGLHA=%X1000          -- heap base address
DEFINE/XMEMGLHS=%X6FFF          -- heap size
DEFINE/XMACBSAD=%X8000          -- stack base address
DEFINE/XMACBSSZ=%X4FFF          -- stack size
DEFINE/XMACDFIM=%XFDFF          -- default interrupt mask
DEFINE/XMACDFSW=%X0000          -- default status word
DEFINE/XMACDFSS=%X400           -- default task stack size
--
-- Define Page Register Contents
--
ADDRESS_STATE/INSTRUCTIONS/AS=0/REGION=PAGE0
ADDRESS_STATE/INSTRUCTIONS/AS=0/REGION=CODEREG
ADDRESS_STATE/OPERANDS/AS=0/REGION=PAGE0
ADDRESS_STATE/OPERANDS/AS=0/REGION=MMRYREG
ADDRESS_STATE/OPERANDS/AS=0/REGION=DATAREG
```

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	200 -- 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	"CCCCCCC10CCCCCCCC20CCCCCCCC30CCCCCCCC40 CCCCCCCC50CCCCCCCC60CCCCCCCC70CCCCCCCC80 CCCCCCCC90CCCCCCC100CCCCCCC110CCCCCCC120 CCCCCCC130CCCCCCC140CCCCCCC150CCCCCCC160 CCCCCCC170CCCCCCC180CCCCCCC190CCCCCC199"

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

Macro Parameter	Macro Value
$ACC_SIZE	16
$ALIGNMENT	4
$COUNT_LAST	2_147_483_646
$DEFAULT_MEM_SIZE	65536
$DEFAULT_STOR_UNIT	16
$DEFAULT_SYS_NAME	TELESOFT_ADA
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	INTERRUPT1
$ENTRY_ADDRESS1	INTERRUPT2
$ENTRY_ADDRESS2	INTERRUPT3
$FIELD_LAST	1000
$FILE_TERMINATOR	ASCII.EOT
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	3.9E+39
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E38

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        0.0

$HIGH_PRIORITY          31

$ILLEGAL_EXTERNAL_FILE_NAME1
                        BADCHAR*^/%

$ILLEGAL_EXTERNAL_FILE_NAME2
                        /NONAME/DIRECTORY

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.ADA")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006D1.ADA")

$INTEGER_FIRST          -32768

$INTEGER_LAST           32767

$INTEGER_LAST_PLUS_1    32768

$INTERFACE_LANGUAGE     ASSEMBLY

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -131_073.0

$LINE_TERMINATOR        ASCII.CR

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        mci'(mil_std,r_mode,andr,r0,r0)

$MACHINE_CODE_TYPE      mci

$MANTISSA_DOC           31

$MAX_DIGITS             9

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -2147483648

$NAME                   NO_SUCH_TYPE_AVAILABLE
```

$NAME_LIST	TELESOFT_ADA
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	65536
$NEW_SYS_NAME	TELESOFT_ADA
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	record null; end record;
$RECORD_NAME	no_such_machine_code_type
$TASK_SIZE	16
$TASK_STORAGE_SIZE	4096
$TICK	0.0001
$VARIABLE_ADDRESS	ADDRESS1
$VARIABLE_ADDRESS1	ADDRESS2
$VARIABLE_ADDRESS2	ADDRESS3

APPENDIX B

COMPILATION SYSTEM AND LINKER 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.

TELESOFT

TeleGen2 Ada Development System for VAX/VMS to Embedded MIL-STD-1750A Targets

Attachment H

Appendix F of the Ada Language Reference Manual

APF-1921N-V1.1(VAX.1750A) 27SEP91

Version 3.25

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at DFAR 252.227-7013, or FAR 52.227-14, ALT III and/or FAR 52.227-19 as set forth in the applicable Government Contract.

TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-9819
(619) 457-2700
(Contractor)

Table of Contents

Appendix F of the Ada LRM

1. Introduction

TeleGen2 for VAX/VMS to 1750A Targets, Version 3.25, compiles the full ANSI Ada language as defined by the *Reference Manual for the Ada Programming Language* (LRM), ANSI/MIL-STD-1815A. This appendix describes the characteristics of the compiler and run-time environment that are designated by the LRM as implementation dependent. These language-related issues are presented in the order in which they appear in the LRM Appendix F.

2. Implementation-defined pragmas

TeleGen2 for the VAX-1750A has six implementation-defined pragmas:

pragma Comment
pragma Linkname
pragma Interrupt
pragma State_Partition_Set
pragma Images
pragma Suppress_All

2.1. Pragma Comment

Pragma Comment is used for embedding a comment into the object code. The syntax of this pragma is

```
pragma Comment(<string_literal>);
```

<string_literal> represents the characters to be embedded in the object code.

Pragma Comment can appear at any location within the source code of a compilation unit, except within the generic formal part of a generic unit. Any number of comments can be entered into the object code by use of pragma Comment.

2.2. Pragma Linkname

Pragma Linkname is used in association with pragma Interface to provide access to any routine whose name can be specified by an Ada string literal. Pragma Linkname takes two arguments, the name of the subprogram specified in a pragma Interface and a string literal specifying the exact link name that the code generator is to use in emitting calls to the associated subprogram. In the example below, pragma Linkname associates the linkname "_access" with the procedure Dummy_Access used in a pragma Interface specification:

```
procedure Dummy_Access(Dummy_Arg: System.Address);
pragma Interface(Assembly, Dummy_Access);
pragma Linkname(Dummy_Access, "_access");
```

A pragma Linkname specification must immediately follow the pragma Interface for the associated subprogram or else a warning will be issued that the pragma Linkname has no effect.

2.3. Pragma Interrupt

Pragma Interrupt is used for function-mapped optimizations of interrupts according to the syntax:

```
pragma Interrupt (Function_Mapping);
```

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

2.4. Pragma State_Partition_Set

Pragma State_Partition_Set is used for grouping packages into a state partition set for applications that use multiple clusters. Pragma State_Partition_Set enables the linker apprentice to identify the application packages that are to be put into the same cluster. A state partition set is a group of packages that have only task entry points. A state partition set maps into a 1750A cluster. The pragma should appear in a package specification that contains only the specifications of the task entry points of the partition, as in the following example:

```
package Partition is
  pragma State_Partition_Set;

  task Partition_Task is
    entry Entry1(Param : Integer);
    entry Entry2(Param : Integer);
    entry Entry3(Param : Integer);
  end Partition_Task;

end Partition;
```

Following this rule, which the compiler does not enforce, prevents data visibility problems.

The State_Partition_Set pragma causes the compiler to attach a special attribute to the library entry for this package. The linker apprentice uses this attribute to identify the entry points of the state partition set. The apprentice will produce a linker options file that groups this package and the extended family of this package into the same cluster. The main procedure has an implicit State_Partition_Set associated with it.

2.5. Pragma Images

Pragma Images controls the creation and allocation of the image table for a specified enumeration type. The pragma may appear only within a compilation unit. The syntax of the pragma is

```
pragma Images(<enumeration_type>, Deferred);
or
pragma Images(<enumeration_type>, Immediate);
```

The default for VAX-1750A is Deferred, which saves space in the literal pool by not creating an image table for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If one of these attributes is used, an image table is generated in the literal pool of the compilation unit in which the attribute appears. If the attributes are used in more than one compilation unit, more than one image table is generated, eliminating the benefits of deferring the table. The image table is a string literal whose length is:

(sum of lengths of literals) + (3 * (number of literals + 1))

For a very large enumeration type, the length of the image table might be greater than Integer'Last and too large to fit into a string. For the 1750A target with a 16-bit integer, the upper bound on the length of the image table is 32767. Using the default Deferred value for all enumeration types allows the user to declare very large enumeration types. The attempt to use 'Image, 'Value, or 'Width for such a type, however, might cause the compiler to attempt to create an image table that exceeds the upper bound, resulting in an error message.

2.6. Pragma Suppress_All

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

3. Implementation-dependent attributes

3.1. 'Offset for machine code insertions

The implementation-dependent attribute 'Offset facilitates machine code insertions by allowing the user to access Ada objects with little data movement or setup. For a prefix P denoting a declared parameter object, P'Offset yields the statically known portion of the address of the first of the storage units allocated to P. The value is the object's offset relative to a base register and is of type Long_Integer.

3.2. Extended attributes for scalar types

The extended attributes extend the concept behind the Text_IO attributes 'Image, 'Value, and 'Width to give the user more power and flexibility when displaying values of scalars. Extended attributes differ in two respects from their predefined counterparts:

1. Extended attributes take more parameters and allow control of the format of the output string.

2. Extended attributes are defined for all scalar types, including fixed and floating point types.

Extended versions of predefined attributes are provided for integer, enumeration, floating point, and fixed point types:

```
Integer:        'Extended_Image, 'Extended_Value, 'Extended_Width
Enumeration:    'Extended_Image, 'Extended_Value, 'Extended_Width
Floating Point: 'Extended_Image, 'Extended_Value, 'Extended_Digits
Fixed Point:    'Extended_Image, 'Extended_Value, 'Extended_Fore, 'Extended_Aft
```

The extended attributes can be used without the overhead of including Text_IO in the linked program. Below is an example that illustrates the difference between instantiating Text_IO.Float_IO to convert a float value to a string and using Float'Extended_Image:

```
with Text_IO;
function Convert_To_String ( Fl : Float ) return String is

Temp_Str : String ( 1 .. 6 + Float'Digits );

package Flt_IO is new Text_IO.Float_IO (Float);

begin

  Flt_IO.Put ( Temp_Str, Fl );
  return Temp_Str;

end Convert_To_String;
```

```
function Convert_To_String_No_Text_Io( Fl : Float ) return String is
begin
  return Float'Extended_Image ( Fl );
end Convert_To_String_No_Text_Io;


with Text_IO, Convert_To_String, Convert_To_String_No_Text_IO;
procedure Show_Different_Conversions is
Value : Float := 10.03376;
begin
  Text_IO.Put_Line ( "Using the Convert_To_String, the value of the variabl
  is : " & Convert_To_String ( Value ) );
  Text_IO.Put_Line ( "Using the Convert_To_String_No_Text_Io, the value
  is : " & Convert_To_String_No_Text_Io ( Value ) );
end Show_Different_Conversions;
```

3.2.1. Integer attributes

'Extended_Image

```
X'Extended_Image(Item,Width,Base,Based,Space_If_Positive)
```

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

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

Parameters

`Item`	The item for which you want the image; it is passed to the function. Required.
`Width`	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. Optional.
`Base`	The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. Optional.

`Based`	An indication of whether you want the string returned to be in base notation or not. If no preference is specified, the default (false) is assumed. Optional.
`Space_If_Positive`	An indication of whether or not a positive integer should be prefixed with a space in the string returned. If no preference is specified, the default (false) is assumed. Optional.

Examples

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

Values yielded for selected parameters:

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

'Extended_Value

```
X'Extended_Value(Item)
```

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

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

Parameter

`Item` A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type X. Required.

Examples

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

Values yielded for selected parameters:

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

'Extended_Width

```
X'Extended_Width(Base,Based,Space_If_Positive)
```

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

Parameters

`Base`	The base for which the width will be calculated. If no base is specified, the default (10) is assumed. Optional.
`Based`	An indication of whether the subtype is stated in based notation. If no value for based is specified, the default (false) is assumed. Optional.
`Space_If_Positive`	An indication of whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified, the default (false) is assumed. Optional.

Examples

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

Values yielded for selected parameters:

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

3.2.2. Enumeration type attributes

'Extended_Image

```
X'Extended_Image(Item,Width,Uppercase)
```

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

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

Parameters

`Item` The item for which you want the image; it is passed to the function. Required.

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

`Uppercase` An indication of whether the returned string is in uppercase characters. In the case of an enumeration type where the enumeration literals are character literals, Uppercase is ignored and the case specified by the type definition is taken. If no preference is specified, the default (true) is assumed. Optional.

Examples

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

Values yielded for selected parameters:

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

'Extended_Value

```
X'Extended_Value(Item)
```

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

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

Parameter

`Item`	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of X. Required.

Examples

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

Values yielded for selected parameters:

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

'Extended_Width

X'Extended_Width

Returns the width for subtype of X.

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

Parameters

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

Examples

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

Values yielded:

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

3.2.3. Floating point attributes

'Extended_Image

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

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

Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameters

`Item` The item for which you want the image; it is passed to the function. Required.

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

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

`Exp` The minimum number of digits in the exponent. The exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. Optional.

`Base` The base that the image is to be displayed in. If no base is specified, the default (10) is assumed. Optional.

`Based` An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. Optional.

Examples

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

Values yielded for selected parameters:

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

'Extended_Value

```
X'Extended_Value(Item)
```

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

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

Parameter

`Item` A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of the input string. Required.

Examples

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

Values yielded for selected parameters:

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

'Extended_Digits

X'Extended_Digits(Base)

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

Parameter

Base The base that the subtype is defined in. If no base is specified, the default (10) is assumed. Optional.

Examples

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

Values yielded:

```
X'Extended_Digits     = 5
```

3.2.4. Fixed-point attributes

'Extended_Image

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

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

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

Parameters

`Item` The item for which you want the image; it is passed to the function. Required.

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

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

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

`Base` The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. Optional.

`Based` Return the string in based notation or not. If no preference is specified, the default is False. Optional.

Examples

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

Values yielded for selected parameters:

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

'Extended_Value

```
X'Extended_Value(Image)
```

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

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

Parameter

`Image`	Parameter of the predefined type string. The type of the returned value is the base type of the input string. Required.

Examples

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

Values yielded for selected parameters:

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

'Extended_Fore

```
X'Extended_Fore(Base,Based)
```

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

Parameters

`Base`	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. Optional.
`Based`	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. Optional.

Examples

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

Values yielded for selected parameters:

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

'Extended_Aft

```
X'Extended_Aft(Base,Based)
```

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

Parameters

`Base` The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. Optional.

`Based` An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. Optional.

Examples

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

Values yielded for selected parameters:

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

4. The package System

The current specification of package System is provided below. Note that the named number Tick is not used by any component of the Ada compiler or run-time system.

```
package System is

  type Address is private;
  Null_Address : constant Address;
  subtype Physical_Address is Long_Integer range 16#0#..16#FFFFF#;
  subtype Target_Logical_Address is Address;
  subtype Target_Address_State is Integer range 0..15;

  type Subprogram_Value is
    record
      Logical_Address : Target_Logical_Address;
      Address_State   : Target_Address_State;
      Parameter_Size  : Natural;
      Static_Base     : Target_Logical_Address;
    end record;

  type Name    is (Telesoft_Ada);

  System_Name    : constant Name := Telesoft_Ada;
  Storage_Unit   : constant := 16;
  Memory_Size    : constant := 65536;
  Min_Int        : constant := -(2 ** 31);
  Max_Int        : constant := (2 ** 31) - 1;
  Max_Digits     : constant := 9;
  Max_Mantissa   : constant := 31;
  Fine_Delta     : constant := 1.0 / (2 ** (Max_Mantissa));
  Tick           : constant := 0.0001;

  subtype Priority is Integer range 0..31;

  Max_Object_Size    : constant := Max_Int;
  Max_Record_Count   : constant := Max_Int;
  Max_Text_Io_Count  : constant := Max_Int-1;
  Max_Text_Io_Field  : constant := 1000;

private

  type Address is access Integer;
  Null_Address : constant Address := Null;

end System;
```

5. Representation clauses

Enumeration representation clauses are supported for character and enumeration types except for Booleans.

Record representation clauses are supported with the following constraints:

- Each component of the record must be specified with a component clause.
- Bits within a byte are referenced by number, with the most significant (leftmost) bit numbered 0 and the least significant (rightmost) bit numbered 7.

Change of representation is not supported for types with record representation clauses.

6. Implementation-generated names

There are no implementation-generated names to denote implementation-dependent components. Names generated by the compiler shall not interfere with programmer-defined names.

7. Address clause expression interpretation

The VAX-1750A compiler does not support address clauses for packages, task units, literal constants, or non-external Ada subprograms. For address clauses applied to objects, a simple expression of type Address is interpreted as a position within the linear address space of the 1750A. The expression is interpreted as the first storage unit of the object. Unchecked_Conversion to the private type System.Address must be used to specify address constants.

8. Unchecked conversion restrictions

Unchecked type conversions are allowed between types (or subtypes) T1 and T2 provided that they do not contain discriminants and that the destination (T2) is not an unconstrained record or array type.

9. Implementation-dependent characteristics of the I/O packages

The TD1750 target I/O modules support minimal input and output functions for use in the ECSPO simulator. Only console I/O (I/O to the default device) is supported.

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

Text_IO has the following implementation-defined characteristics:

- External files are not supported. All calls to the Text_IO file management procedures Create and Open will result in Use_Error.
- Type Count is defined as follows:

```
type Count is range 0..32767;
```

- Type Field is defined as follows:

```
subtype Field is integer range 0..1000;
```

ATTACHMENT F: PACKAGE STANDARD INFORMATION

For this target system the numeric types and their properties are as follows:

INTEGER:

```
type INTEGER is range -32768 .. +32767;
```

FLOAT:

```
type FLOAT is digits 6 range -1.0479E+38 .. +1.0479E+38;
```

DURATION:

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

```
-- Optionally supported numeric types
```

LONG_INTEGER:

```
type LONG_INTEGER is range -2147483648 .. +2147483647;
```

LONG_FLOAT:

```
type LONG_FLOAT is digits 9 range -1.6891629E+38 .. +1.6891629E+38;
```

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, which are not a part of Appendix F, are given on the following customer supplied page.

TELESOFT

TeleGen2 Ada Development System for VAX/VMS to Embedded MIL-STD-1750A Targets

Compiler Command Options

OPT-1920N-V1.1(VAX.1750A) 27SEP91

Version 3.25

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at DFAR 252.227-7013, or FAR 52.227-14, ALT III and/or FAR 52.227-19 as set forth in the applicable Government Contract.

TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-9819
(619) 457-2700
(Contractor)

Table of Contents

List of tables

Compiler Command Options

1. The compile command

The general command format for invoking the Ada compiler is:

```
$ TSADA/1750A/ADA<qualifier>[...] <file>[...]
```

<qualifier> Is one of the various compiler qualifiers. These qualifiers fall into four categories.

Search scope:	library list (/LIBFILE, /TEMPLIB)
Execution control:	abort after errors (/ABORT_COUNT) output progress messages (/MONITOR) process multiple files (/INPUT_LIST) update library for multiple files (/UPDATE)
Output control:	code optimization (/CG_OPTIMIZATION) suppress exception strings (/EXC_NAMES) suppress exception reasons (/EXC_REASONS) checking only (/NOOBJECT) enable debugging (/DEBUG) inline code (/OPTIMIZE) extended memory (/SEGMENTCALL) multi-cluster (/SPS) save library space (/SQUEEZE) suppress run-time checks (/SUPPRESS) control writing to stack (/WRITE_AHEAD)
Listing control:	listing file (/LIST) error context (/CONTEXT) page length (/LINES_PER_PAGE) page width (/PAGE_WIDTH)

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

1.1. Specifying a library

The user must have a library for compilation. As a default, the compiler uses the library file named LIBLST.ALB in the current working directory. An alternative library can be specified by using either the /LIBFILE or the /TEMPLIB qualifiers. Be sure to include the correct run-time sublibraries for your application.

1.2. Multiple compilations in the same directory

Temporary files are generated by the compiler and deleted when the compilation is complete. Users should not use the following file names in their compilation directory:

```
ADA.TMP
AINFO.
ERRORS.
TEMPLIB.ALB
TSADA_IEL.LIST    -- produced when /LIST is used
TSADA.OPTS        --     "       "    "    "  "
<unit>.SUM  --     "       "    "    "  "
```

Do not perform simultaneous compilations with the same default directory, as there will be confusion between temporary files generated by the compiler.

1.3. Using compilation command files

The compiler can be invoked from standard VMS command files. The result of each compilation is returned as a Boolean in the global symbol $STATUS. An odd value is returned if the compilation was successful; an even value is returned if the compilation failed. On systems with moderate to heavy user loads, it is recommended that compilations be performed from command files submitted to a batch queue.

1.4. Compiler command qualifiers

Table 1 summarizes the qualifiers for the /ADA command. Defaults are shown in boldface.

Table 1. Compiler command qualifiers

Qualifier Name	Action
/ABORT_COUNT=<n> (999)	Specify maximum errors/warnings.
/[NO]CG_OPTIMIZATION [=<string>] (abpst)	Optimize object code. Specify a,b,p,s,t optimizations.
/CONTEXT=<n> (1)	Request <value> context lines around each error in error listing.
/[NO]DEBUG	Generate information for the debugger.
/[NO]EXC_NAMES	Enable the output of exception names.
/[NO]EXC_REASONS	Enable the output of exception reasons.
/INPUT_LIST	Input file contains names of files to be compiled, not Ada source.
/LIBFILE=<file> (LIBLST.ALB)	Specify name of library file.
/LINES_PER_PAGE=<n> (60)	Specify page length for a listing. Used with /LIST qualifier.
/[NO]LIST[=<file>]	Create listing file. Default name is <source>.LIS, or <file>.LIS, if specified.
/[NO]MONITOR	/MONITOR requests progress messages.
/[NO]OBJECT	/NOOBJECT restricts compilation to syntactic and semantic analysis.
/[NO]OPTIMIZE [=<option>(,...)] /[NO]GRAPH [=<file>]	/OPTIMIZE causes the Optimizer to be run on unit(s) being compiled. Generates a call graph for a unit. Default output file is <unit>.GRF.
/PAGE_WIDTH=<n> (132)	Specify number of characters per line (80 or 132) for a listing.
/[NO]SEGMENTCALL	/SEGMENTCALL specifies that this unit is called across address state boundaries.
/[NO]SPS	/SPS generates code to support units in an application split across state partition sets.
/[NO]SQUEEZE	Deletes unneeded intermediate unit information after compilation.
/[NO]SUPPRESS[=<option>(,...)] (ALL_CHECKS)	Suppresses selected run-time checks in generated object code.
/TEMPLIB =<sublib>(,...)	Specify a temporary library ontaining listed sublibraries.
/[NO]UPDATE	Updates the sublibrary after each unit compiled from a list.
/[NO]WRITE_AHEAD	/NOWRITE_AHEAD prevents the compiler from writing ahead of the top of stack.

/ABORT_COUNT=<n>
/ABORT_COUNT=999 (default)

Specifies the number of errors or warnings after which the compilation is aborted. The minimum value is 1. The compiler maintains separate counts of all syntactic errors, semantic errors, and warning messages issued during a compilation. If any of these counts becomes too great, the user will generally want to abort the compilation. This qualifier allows the user to set the number of errors at which this action occurs.

/CG_OPTIMIZATION (default, =abpst)
/CG_OPTIMIZATION=<string>
/NOCG_OPTIMIZATION

Specifies the types of code optimizations to be performed by the 1750A Code Generator. Each type of optimization is represented by a character:

Character	Optimization
a	address-mode optimizations (e.g., attempt to collapse complex indexing expressions)
b	branch optimizations
p	peephole optimizations on generated code
s	subprogram entry/exit code optimizations
t	tracking of long-range register usage

The specified string is a sequence of 1-5 characters representing the selected types of optimizations. If no string is specified, all optimizations are performed, which is equivalent to /CG_OPTIMIZATION=abpst. /NOCG_OPTIMIZATION indicates that no code generator optimizations are to be performed, which speeds up compile time by about 20%.

/CONTEXT=<n>
/CONTEXT=1 (default)

Specifies the number of source lines to immediately precede and follow an error in an error listing. When an error message is output, it is helpful to include the lines of the source program that surround the line containing the error. These lines provide a context for the error in the source program and help to clarify the nature of the error. The qualifier affects the error messages output on SYS$OUTPUT or in a listing.

The specified value represents the total number of lines output for each error (including the line in error). The default value of 1 thus produces a listing that contains only the lines in error. When a greater value is specified, the extra context lines are distributed before and after the line in error. For example, consider the following source code fragment (taken from the ACVC test suite):

```
package P is
  type T1 is range 1..10;
  type T2 is digits 1;
  type Arr is array (1..2) of integer; type T3 is new Arr;     -- OK.

    package Inner is
      type In1 is new T1;      -- ERROR: T1 DERIVED.
      type In2 is new T2;      -- ERROR: T2 DERIVED.
      type In3 is new T3;      -- ERROR: T3 DERIVED.
      type Inarr is new Arr;   -- OK.
```

Compiling this program with /CONTEXT=2 produces a listing with the line in error and one extra line of context.

```
20:       package Inner is
21:         type In1 is new T1;    -- ERROR: T1 DERIVED.
                                   ------
>>>     Illegal parent type for derivation <3.4:15,7.4.1:4>

        Type 'A' or <esc> to abort, ' ' or <cr> to continue:
```

/DEBUG
/NODEBUG (default)

Causes the generation of Debugger Information (DI) to support debugging the compiled unit(s). /DEBUG automatically sets the /NOSQUEEZE qualifier to ensure that the High Form, Low Form, and Debugger Information for secondary units are not deleted. While the compilation time overhead generated by use of /DEBUG is minimal, retaining Debugger Information increases the space overhead in the library. To check if a compilation unit has been compiled using /DEBUG, use the TSADA/1750A/SHOW/EXTENDED command for the unit.

/EXC_NAMES
/NOEXC_NAMES (default)

Enables run-time display of the full textual name of an exception defined in the Ada unit if the exception is raised and not handled. Identifying the exception facilitates debugging during development. /NOEXC_NAMES eliminates exception strings, displaying "name unavailable" instead, to minimize object code size for debugged production code.

/EXC_NAMES controls information about an exception only if the /EXCEPTION_INFO qualifier was specified when binding the program. /EXCEPTION_INFO enables information about an unhandled exception to be displayed before the exception is propagated out of the main subprogram and the program halts.

/EXC_REASONS
/NOEXC_REASONS (default)

Causes the code generator to generate and propagate a more detailed exception reason code for the standard predefined exceptions. For example, the Ada exception Storage_Error will have associated with it one of two reasons, Stack_Overflow or Allocator_Failure. /NOEXC_REASONS eliminates this reason information and displays "NO_STATED_REASON" in describing an unhandled exception.

Using /EXC_REASONS when compiling any unit in a program causes an 80-word reason table in the CGS assembly module XEXC0 to be referenced and included in the program at link time. Using /NOEXC_REASONS on *all* units in a program eliminates references to the reason table so that it is not linked in with the program. Use /EXC_REASONS for debugging during development and then use /NOEXC_REASONS for all units in the program to optimize the size of debugged production code.

/EXC_REASONS controls information about an exception only if the /EXCEPTION_INFO qualifier was specified when binding the program. /EXCEPTION_INFO enables information about an unhandled exception to be displayed before the exception is propagated out of the main subprogram and the program halts.

/INPUT_LIST

Indicates that the file specified on the /ADA command line contains a list of source files to be compiled instead of the Ada unit(s) directly. The input file should contain the source file specifications, one per line. The default file type is ".LIS".

For example, to compile source files CALC_ARITH.ADA and CALC_MEM.ADA in the current default directory and file CALC_IO.ADA in directory [CIOSRC], the user could first prepare a file CALC_COMPILE.LIS containing the following text:

```
CALC_ARITH
CALC_MEM
[CIOSRC]CALC_IO
```

The following command would then cause these three files to be compiled in sequence.

```
$ TSADA/1750A/ADA/INPUT_LIST  CALC_COMPILE.LIS
```

Functionally, each file in the input list is treated as a separate compilation. However, the overall compilation rate will be higher than the corresponding separate compilations, as the compiler is only initialized once, and the Ada library is only opened once.

When the /INPUT_LIST qualifier is used in conjunction with the default /UPDATE qualifier, the working sublibrary is updated after each unit that

successfully compiles. If a unit in the list fails to compile because of errors, the sublibrary is still updated for all other successfully compiled units before and after it in the list. If /NOUPDATE is used, the sublibrary is not updated for any unit if one unit fails to compile.

In addition to the names of the source files, the input list can contain comments in Ada syntax. All text on a line including and following the comment marker "--" will be ignored. Thus, an equivalent of the example above is a file with the contents:

```
-- New baseline components for Calculator Project
CALC_ARITH
CALC_MEM
[CIOSRC]CALC_IO  -- Jim's implementation of display blink
```

/LIBFILE=<file>
/LIBFILE=LIBLST.ALB (default)

Specifies the name of the library file to be used. The file contains a list of sublibraries. If <file> does not include a file extension, the compiler uses a default extension of ".ALB". The default library file is LIBLST.ALB in the default directory.

/LINES_PER_PAGE=<n>
/LINES_PER_PAGE=60 (default)

Specifies the number of lines of output per page in a listing, including the header for each page. The value can be any positive integer.

/LIST (default)
/LIST=<file>
/NOLIST

Produces a file containing 1) a source listing with numbered lines interspersed with any error messages, 2) a summary of processing times, and 3) a summary of information about the compilation unit if no errors occurred.

The /NOLIST qualifier suppresses the listing and error output to a file. The compiler still outputs error messages to the device specified by SYS$OUTPUT.

If /LIST is used without specifying an output file, the error output is sent to the file named "<source>.LIS", where <source> is the name of the source file being compiled. If a file is specified, the output is sent to that file instead. If the specified file does not include a file extension, the system appends a file extension of ".LIS". If a file name is specified and /INPUT_LIST is used, the listing for each file in the input list is sent to a separate version of the file.

The number of surrounding source lines output with an error message is determined by the /CONTEXT qualifier. The dimensions of a listing are controlled by the /LINES_PER_PAGE and /PAGE_WIDTH qualifiers.

The listing file contains a series of sections. A section is identified by the header output on each page. The header contains the following information:

Section title
Source file specification
Compiler version banner
Date and time of the file creation
Page number of the listing

The header is output on either one or two lines depending on the specified page width.

Three sections are output in a listing:

1. ERROR LISTING
2. TIME SUMMARY
3. UNIT SUMMARY

1. The ERROR LISTING section contains the TSADA/1750A/ADA command line used to compile the source file and the Ada source lines numbered with any error messages interspersed with the source. The last line of the section summarizes the number of errors and warnings encountered in the lines compiled.

2. The TIME SUMMARY section consists of a table showing the elapsed time and the CPU time in milliseconds used by each phase of compilation and by the compilation as a whole. The section also includes the number of lines compiled per CPU minute.

3. The UNIT SUMMARY section contains information produced by the code generator for each secondary unit in the Ada source file compiled. The unit is characterized by its library component name, and its code size, static data size, and read-only data size. Each subprogram in the unit is then summarized by providing its code offset, frame size, and the subprograms or routines it calls at listed source lines. If /NOOBJECT is used, or if no code was generated due to syntactic or semantic errors, no information will be available for this section. The message:

```
***   UNIT SUMMARY WAS NOT PRODUCED   ***
```

will then be the only output for this section.

/MONITOR
/NOMONITOR (default)

Causes the compiler to display progress messages that allow the user to monitor the compilation process. Normally, the only visible output produced by the compiler during operation is the error/listing output, if the default /LIST qualifier is used.

When the /MONITOR qualifier is used, the output is sent to the file specified by the logical name SYS$OUTPUT. First, the compiler produces a banner message announcing the release number. Next, the front end lists the name of the file it is compiling and the current date and time. If no errors are detected, the middle pass, optimizer (if specified), and code generator phases of the compiler process each compilation unit in the source file. As they begin execution, the middle pass, optimizer, and code generator each list the name of the unit they are processing and its type (e.g., specification or body). When all compilation units have been processed, a final status line is produced. This line indicates the number of syntax errors and semantic errors encountered, the number of warning messages issued, and the number of lines compiled.

/OBJECT (default)
/NOOBJECT

Controls the generation of object code. The /NOOBJECT qualifier instructs the compiler to perform syntactic and semantic analysis of the source program without generating object code. /OBJECT allows the generation of object code.

/NOOBJECT is frequently used in conjunction with the optimizer when a unit is being compiled in preparation for collective optimization only and object code is not required. To support this use, /NOOBJECT sets a default of /NOSQUEEZE to ensure that the High Form and Low Form for secondary units, required for collective optimization, are not deleted. Another use of the /NOOBJECT qualifier is early in program development when syntactic and semantic checking is all that is required and maximum compilation rate is of interest. In this case, specifying /SQUEEZE explicitly limits the size of stored code.

/OPTIMIZE[=<option>(,...)][<qualifier>]
/NOOPTIMIZE (default)

Causes the compiler to invoke the optimizer to optimize the Low Form generated by the middle pass for the unit(s) being compiled. The code generator takes the optimized Low Form as input and produces more efficient object code.

<option> is one of the optimizer options listed in Table 2. Defaults are shown in boldface. If no options are specified, SAFE is the default.

<qualifier> can be either /NOGRAPH or /GRAPH[=<file>]. /NOGRAPH is the default value. /GRAPH generates a call graph for the unit being compiled. The default output file is <unit>.GRF when no file is given. If multiple units are being compiled and optimized and call graphs are desired, use /GRAPH without specifying a file name. The graph for each unit will be located in a separate file named after the unit. If a file is given, separate versions of the file will be created for each unit.

Table 2. /OPTIMIZE options

Option	Action
ALL	Enables all optimization options. See Section "Noptim_optquals before using this. ALL is equivalent to (AUTOINLINE,INLINE, NOPARALLEL,NORECURSE).
NONE	Disables all optimization options. NONE is equivalent to (NOAUTOINLINE,NOINLINE,PARALLEL,RECURSE).
SAFE	Enables optimizations that are safe to do on all programs. SAFE is equivalent to (AUTOINLINE,INLINE, PARALLEL,RECURSE).
[NO]AUTOINLINE	AUTOINLINE enables automatic inlining of subprograms called from one place, including those not marked by the user with an Inline pragma. INLINE and NOAUTOINLINE only enable inlining of subprograms marked by the user and subprograms generated by the middle pass and called from one place. NOINLINE has precedence over AUTOINLINE.
[NO]INLINE [:<list_file>]	INLINE enables inline expansion of subprograms marked with an Inline pragma and compiler-generated subprograms. NOINLINE suppresses all inlining. The <list_file> contains two lists of subprograms separated by a semicolon. Each subprogram is qualified by the name of the unit containing its declaration. Subprograms in the first list will be inlined; those in the second will not.
[NO]PARALLEL	PARALLEL indicates that subprograms may be called from parallel tasks.
[NO]RECURSE	RECURSE indicates that subprograms may be called recursively.

/PAGE_WIDTH=<n>
/PAGE_WIDTH=132 (default)

Specifies the number of characters per line to be output in a listing. The specified value must be either 80 or 132. If a line exceeds the specified page width, the line is broken with an "&" appended, and the remainder of the line is continued on the next line. If the page width is 132, the header for each page is on one line. If the page width is 80, the header is on two lines.

/SEGMENTCALL
/NOSEGMENTCALL (default)

Causes the compiler to generate code to handle calling a unit from a unit located in a different address state. This qualifier is used for extended memory applications that are divided among multiple address states. It causes the compiler to generate both a GATE and a DESC control section. The run-time system uses these extra control sections to transfer control to the called routine across address state boundaries.

If an application is split across multiple address states, compile all of its units with /SEGMENTCALL. You can attempt to save space by not compiling a unit with /SEGMENTCALL if none of the unit's subprograms are called from another address state. This attempt might not work, however, if the unit has compiler-generated subprograms, such as elaboration and record initialization subprograms, that are called from another address state.

/SPS
/NOSPS (default)

Causes the compiler to generate different code for the Ada tasking constructs that are used to communicate between different state partition sets used for an application. This qualifier is only used to compile a unit that will be part of an application that is split across state partition sets using pragma State_Partition_Set. /SPS also causes the generation of GATE and DESC control sections for use by the run-time segment call mechanism.

When using state partition sets, compile *all* of the compilation units of the application with the /SPS qualifier. In addition, in the library list, replace the standard TS1750RTL sublibrary with the TS1750RTL_SPS sublibrary.

/SQUEEZE (default)
/NOSQUEEZE

Causes the compiler to delete the High Form and Low Form intermediate representations for the compiled unit in the working sublibrary. The deletion of this information can cause significant decreases in the size of sublibraries, typically 50% to 70% for multi-unit programs. Some information that is required for any subsequent use of the unit is saved, however. For example, the information about the contents of the specification of a package must be saved for compilation of the corresponding body. Other information that is only used by the standalone optimizer, the cross-referencers, or the debugger can be deleted if these utilities are not going to be used on the unit.

Use /NOSQUEEZE if the compiled unit is to be bound, optimized as part of a collection, or cross-referenced. The /DEBUG qualifier automatically sets the /NOSQUEEZE qualifier.

/SUPPRESS[=<option>(,...)]
/NOSUPPRESS (default)

Allows the user to suppress selected run-time checks in generated object code. <option> is one of the features to be suppressed. The default option is ALL_CHECKS. The options and their actions are presented in Table 3. The default is in boldface.

The Ada language requires, as a default, a wide variety of run-time checks to ensure the validity of operations. For example, arithmetic overflow checks are required on all numeric operations, and range checks are

required on all assignment statements that could result in an illegal value being assigned to a variable. While these checks are vital during development and an important asset of the language, they introduce a substantial overhead. This overhead may be prohibitive in time-critical applications. Thus, the Ada language provides a way to selectively suppress classes of checks via the Suppress pragma. However, use of the pragma requires modifications to the Ada source. The /SUPPRESS qualifier provides an alternative mechanism that allows the user to specify functionality equivalent to the Suppress pragma in the compiler invocation command. The Suppress pragma is valid in any declarative region of a package and affects all nested regions. The /SUPPRESS qualifier is equivalent to adding pragma Suppress to the beginning of the declarative part of each compilation unit in a file, except for the ELABORATION_CHECK option, which differs from pragma Suppress(Elaboration_Check).
/SUPPRESS=(ELABORATION_CHECK) suppresses elaboration checks made by other units on this unit. The pragma suppresses elaboration checks made on other units from this unit.

Table 3. /SUPPRESS options

Option	Action
ALL_CHECKS	Suppress all access checks, discriminant checks, division checks, elaboration checks, index checks, length checks, overflow checks, range checks, and storage checks.
ELABORATION_CHECK	Suppress all elaboration checks.
OVERFLOW_CHECK	Suppress all overflow checks.

The names of the options can be abbreviated as long as they remain unique within the set of options. The ALL_CHECKS option functions as if a Suppress pragma were present in the Ada source for each of the checks, except for ELABORATION_CHECK, as described above.

As an example of usage, the following command suppresses all overflow and elaboration checks in the object code of the units in file MY_FILE.

```
$ TSADA/1750A/ADA/SUPPRESS=(OVERFLOW,ELAB)  MY_FILE
```

/TEMPLIB=<sublib>(,...)

Specifies a list of sublibraries to be used for a single compilation. A <sublib> can be prefixed with the specification of the VMS directory in which it resides. If neither /LIBFILE nor /TEMPLIB is provided, the compiler assumes that the library is specified by the library file named LIBLST.ALB in the current working directory.

/UPDATE (default)
/NOUPDATE

Causes the compiler to update the working sublibrary after each unit in an input list successfully compiles. This qualifier is only meaningful when the compilation uses /INPUT_LIST and an input list.

If a unit in the input list fails to compile because of errors, the sublibrary is still updated for all other successfully compiled units before and after it in the list. If later units depend on the failed unit, they must be recompiled. This is not often the case, however, because users most often compile package bodies that have no such dependencies. If a source file contains more than one unit and one unit fails to compile, the remaining units in the source are compiled for syntax errors only (i.e., no code is generated).

If /NOUPDATE is used and one unit fails to compile, the sublibrary is not updated at all, for any unit. All units and sources after the one that failed are compiled for syntax errors only. /NOUPDATE is useful if the user knows that all compilations will be successful and wants to save the time involved in updating the sublibrary for each source file.

/WRITE_AHEAD (default)
/NOWRITE_AHEAD

Controls when the compiler is allowed to write ahead in the stack. The compiler is normally the only user of the run-time stack, and it assumes that interrupts are handled on a stack separate from the run-time stack. For users implementing non-tasking interrupt handling, this assumption may not be safe. The /NOWRITE_AHEAD qualifier allows users to turn off the default behavior. This feature is useful when compiling subprograms that need to coexist with interrupt handling code.

/WRITE_AHEAD causes the code generator to emit entry and exit code sequences that may write ahead of the top of stack. /NOWRITE_AHEAD causes the code generator to emit reordered entry and exit code sequences that do not write ahead of the run-time stack pointer. The code sequences generated when using /NOWRITE_AHEAD contain an extra single-word instruction.

/NOWRITE_AHEAD is ignored under certain conditions:

1. When used with /CG_OPTIMIZE="s", a warning message is issued and the /CG_OPTIMIZE="s" is ignored.

2. This qualifier cannot be used for subprograms calling locally declared subprograms. If this occurs, a warning message is issued and the qualifier is ignored.

3. If the cumulative size of the parameters passed to a subprogram exceeds 10 words (the number of registers available for parameter passing), a

warning message is issued and the qualifier is ignored. When the qualifier is ignored, the entry/exit code sequences will write ahead on the stack.

2. The binder command

The binder can be invoked at the same time as the linker, as described in the linker command section, or by itself using a command of the format:

```
$ TSADA/1750A/LKTOOLS/[NO]BIND[<qualifier>[...]]  <main>
```

`/BIND` Enables binding.

`/NOBIND` Is the default for use with prebound programs and the linker.

`<qualifier>` Is one of various qualifiers for the binder. These qualifiers fall into two categories.

Search scope: library list (/LIBFILE, /TEMPLIB)

Output control: suppress exception information (/NOEXCEPTION_INFO)
bind for expanded memory (/EXPANDED)

`<main>` Is the Ada name of the subprogram that is the main program unit (not the name of the source file).

The binder can be invoked from standard VMS command files. On systems with moderate to heavy user loading, it is recommended that binds be performed from command files submitted as batch jobs.

Table 4 summarizes the qualifiers for the /BIND command. Defaults are shown in boldface.

Table 4. Binder command line qualifiers

Qualifier	Action
/[NO]EXCEPTION_INFO	Include debug information in the code.
/[NO]EXPANDED	Bind for expanded memory.
/LIBFILE=<file> (LIBLST.ALB)	Specify name of library file.
/[NO]MONITOR	Show progress messages.
/TEMPLIB=(<sublib>[,...])	Specify temporary list of sublibraries.

/EXCEPTION_INFO (default)
/NOEXCEPTION_INFO

This qualifier applies only to bind operations. The binder generates the exception handling tables for exceptions that are propagated out of the elaboration code or the main program. If /EXCEPTION_INFO is specified or applies by default, the handler for those exceptions is defined in the CGS module XDBG and provides information about the exception before halting the program. This is the only reference to XDBG, which defines a large number of strings used for displaying the information. If /NOEXCEPTION_INFO is specified, XDBG is not referenced and is not included in the program image. Exceptions that are not handled within the program will cause the program to halt with no information at all displayed. /EXCEPTION_INFO is the default for use during code development. Use /NOEXCEPTION_INFO to eliminate the overhead of exception information in debugged production code.

/EXPANDED
/NOEXPANDED (default)

This qualifier informs the binder that the application program is being linked for expanded memory (multiple address states). This qualifier causes the binder to build the Global Exception Map using logical 32-bit references (address state and logical address) rather than 16-bit references. Use the /EXPANDED qualifier whenever you are linking an application that has state partition sets or multiple address states. The default is /NOEXPANDED, for use with non-expanded memory applications.

/LIBFILE=<file>
/LIBFILE=LIBLST.ALB (default)

This qualifier specifies the name of the library file to be used. The qualifier has the same effect as the library manager qualifier of the same name. The file contains a list of sublibraries. If <file> does not include a file extension, the linker uses a default extension of ".ALB". The default library file is LIBLST.ALB in the default directory.

/TEMPLIB=<sublib>(,...)

This qualifier specifies a list of sublibraries to be used for a single run of the linker. The qualifier has the same effect as the library manager qualifier of the same name. A <sublib> can be prefixed with the specification of the VMS directory in which it resides. If neither /LIBFILE nor /TEMPLIB is provided, the linker assumes that the library is specified by the library file named LIBLST.ALB in the current working directory.

Obtaining lists of units to be bound

The complete list of units required by the binder, in correct elaboration order, can be obtained by using the library listing command:

```
$ TSADA/1750A/SHOW/ELABORATION/NAME=<main>
```

<main> is the Ada name of the main program unit.

This report also specifies any missing or obsolete units and reports any elaboration circularities that would prevent the binder operation from succeeding. (Note: if the library file is not LIBLST.ALB, the library must also be specified in this command with the /LIBFILE or /TEMPLIB qualifier.)

Similarly, the compilation status of these units (i.e., the need to be recompiled) can be obtained by using the library listing command:

```
TSADA/1750A/SHOW/EXECUTABLE/BRIEF/NAME=<main>
```

3. The linker command

The Ada linker can be invoked by itself or with the bind operation. The syntax for invoking the Ada linker and binder is:

```
$ TSADA/1750A/LKTOOLS/[NO]BIND/[NO]LINK[<qualifier>][...] <main>
```

`/[NO]BIND` /BIND is required if the main program has not been bound already. /NOBIND is the default.

`/[NO]LINK` /LINK enables linking. /NOLINK is the default, which prevents linking after a bind operation.

`<main>` Is the name of the Ada compilation unit to be linked as a main program. Do not use this parameter if the name of the unit is specified in the linker options file with INPUT/MAIN for a link-only operation.

`<qualifier>` Is one of the various command line qualifiers. These qualifiers fall into four categories.

Search scope:	library list (/LIBFILE, /TEMPLIB)
Execution control:	use options file (/OPTIONS) display progress messages (/MONITOR) allot virtual space (/VIRARS_SIZE)
Output control:	enable debugging (/DEBUG) specify output (/ECSPO, /IEEE, /LOAD_MODULE, /OBJECT_FORM) produce link map (/MAP, /IMAGE, /LINES_PER_PAGE, /PAGE_WIDTH)

3.1. Linker command line qualifiers

The linker can be directed by means of VMS command line qualifiers or, alternatively, by using options entered via an options file or SYS$INPUT. Command line qualifiers are useful for controlling options that a user is likely to change often. The default qualifier settings are designed to allow for the simplest and most convenient use of the linker.

Table 5 summarizes the qualifiers for the /BIND and /LINK commands. Defaults are shown in boldface.

Table 5. Linker command line qualifiers

Qualifier	Action
/[NO]BIND /[NO]EXCEPTION_INFO /[NO]EXPANDED	Elaborate the main program. Include debug information in the code. Bind for expanded memory.
/LIBFILE=<file> (LIBLST.ALB)	Specify name of library file.
/LINES_PER_PAGE=<n> (60)	Specify number of lines per page (> 10) for every listing produced.
/[NO]LINK /[NO]DEBUG /ECSPO /IEEE /LOAD_MODULE [=<file>] /[NO]MAP[=<file>] /[NO]IMAGE /OBJECT_FORM [=<lib_comp>] /VIRARS_SIZE=<n> (2)	Create a load module. Produce a debug link map. Output ECSPO format. Output IEEE-695 format. Produce a load module. Specify file name of load module. Request and control link map. /IMAGE includes a memory image. Choose linked OF module output. Set number of virtual arrays (1 .. 10) for linker operations.
/[NO]MONITOR	Show progress messages.
/[NO]OPTIONS[=<file>] (SYS$INPUT)	Specify options file for the linker. Default file is SYS$INPUT. Default file extension is .OPT for the linker.
/PAGE_WIDTH=<n> (132)	Specify number of characters per line (132 or 80) for every listing produced.
/TEMPLIB=(<sublib>[,...])	Specify temporary list of sublibraries.

/DEBUG
/NODEBUG (default)

/DEBUG causes the linker to produce a debug link map and put it in the Ada library. Use this qualifier when linking main programs to be debugged with the source level debugger.

The default is /NODEBUG.

/ECSPO (default)

This qualifier specifies that the linker should output ECSPO load module format, which is the default output format. If used, /ECSPO must immediately follow the /LINK qualifier. The load module produced has the file type ".SO."

/IEEE

This qualifier specifies that the linker should output standard IEEE-695 load module format. If used, this qualifier must immediately follow the /LINK qualifier. The load module produced has the file type ".I3E." The default output format is ECSPO.

/LIBFILE=<file>
/LIBFILE=LIBLST.ALB (default)

This qualifier specifies the name of the library file to be used. The file contains a list of sublibraries. If <file> does not include a file extension, the linker uses a default extension of ".ALB". The default library file is LIBLST.ALB in the default directory.

/LINES_PER_PAGE=<n>
/LINES_PER_PAGE=60 (default)

/LINES_PER_PAGE specifies the number of lines per page for the link map listing. The value is a positive integer greater than 10.

/LOAD_MODULE[=<file>]

This qualifier is used primarily to specify the VMS file name for the load module created by the linker. <file> is the optional VMS file specification for the output. If <file> does not include a file type, the linker will append a .SO extension for the default ECSPO load module format, or a .I3E extension for IEEE-695 format if /IEEE has been specified. If no output file is specified, the linker will write the linked output to <unit>.SO or <unit>.I3E, as appropriate, where <unit> is the Ada name of the main program unit (if present), the name specified as the command line parameter, or the name specified as the first INPUT option, modified as necessary to form a valid VMS file specification.

The /LOAD_MODULE qualifier can be used with the /OPTIONS qualifier. Any output file specification present in the options file is superceded by the specification on the command line. Both formats are produced by specifying /LOAD_MODULE and /OBJECT_FORM.

/NOMAP (default)
/MAP[=<file>]
[/IMAGE]
[/NOIMAGE] (default)

This qualifier is used to request and control a link map listing. <file> is the optional VMS file specification for the link map output. If the user does not specify a file extension, the linker uses a default extension of ".MAP". If the user does not specify an output file, the linker writes the listing to <unit>.MAP, where <unit> is the name of the main program unit (if present), the name specified as the command line parameter, or the name specified as the first INPUT option, modified as necessary to form a valid VMS file specification.

/IMAGE generates a memory image listing in addition to the map listing. The linker writes the image listing to the same file as the link map listing. This is the only optional section of the listing. The default is /NOIMAGE.

A command line /MAP qualifier supercedes any MAP options in an options file. /NOMAP can be used on the command line to suppress MAP options specified in an options file.

/MONITOR
/NOMONITOR (default)

The /MONITOR qualifier causes the linker tools invoked to put out progress messages that allow the user to monitor the current process. When /MONITOR is used, the output is sent to the file specified by the logical name SYS$OUTPUT. If the job is run in batch mode, the output can be captured in the log file.

/MONITOR causes the linker tools driver to display a banner and then identify each of the tools as they are initialized and finished. The binder and linker each function as a unitary operation and hence mainly display error messages.

/OBJECT_FORM[=<lib_comp>]

This qualifier specifies that one output of the linker is to be linked OF. The linked OF is put into the library as an object form module ("ofm") component. <lib_comp> is any name desired by the user. If no library component name is specified, the output of the link is put into the library as the object form module <unit>, where <unit> is the Ada name of the main program unit (if present), the name specified as the command line parameter, or the name specified as the first INPUT option. The object form module of <unit> is a library component separate from that of the specification or body of the unit.

If an object form module library component with the specified name already exists in the current working sublibrary, the existing component is deleted and replaced by the new output.

The /OBJECT_FORM qualifier can be used with the /OPTIONS qualifier. Any format or name present in the options file is superceded by the format and name specified on the command line. The user can request /OBJECT_FORM instead of /ECSPO or /IEEE or in addition to /LOAD_MODULE.

/NOOPTIONS (default)
/OPTIONS
/OPTIONS=<file>

The /OPTIONS qualifier specifies that the linker is to process additional options obtained interactively or from an options file. The default file specification is SYS$INPUT, for entering options interactively after the command line is processed. For interactive use, no prompts are given, and an EXIT command is used to mark the end of input. Options cannot be entered both interactively and in an options file. <file> is a valid VMS file specification and represents a file containing linker options.

/PAGE_WIDTH=<n>
/PAGE_WIDTH=132 (default)

/PAGE_WIDTH specifies the number of characters per line for every listing produced. The value of <n> is either 80 or 132. The default value is 132 characters.

/TEMPLIB=<sublib>(,...)

This qualifier specifies a list of sublibraries to be used for a single run of the linker. A <sublib> can be prefixed with the specification of the VMS directory in which it resides. If neither /LIBFILE nor /TEMPLIB is provided, the linker assumes that the library is specified by the library file named LIBLST.ALB in the current working directory.

/VIRARS_SIZE=<n>
/VIRARS_SIZE=2 (default)

This qualifier is used to set the number of virtual arrays (virars) available to the linker to perform its operations. The linker has its own virtual space manager, which this qualifier affects, while the library's virtual space manager, used for other functions, is not affected. If the process performing the link does not have enough virtual pages for allocation, this qualifier will not increase the linker's virtual working size. The value for <n> is an integer in the range 1..10. The default is 2.

3.2. Linker options file

The /OPTIONS command line qualifier is used to specify that additional options are to be read from an options file or from SYS$INPUT. When the linker apprentice is not used, /OPTIONS can specify a text file with options for the linker. The options file provided is named L1750.OPT. The default file specification is SYS$INPUT, whereby the linker can read commands from a command file or from user input from a terminal immediately after the linker command line. For interactive use, no prompts are given, and an EXIT command is used to mark the end of input. Options cannot be entered both interactively and in an options file.

Options file format

Each option in an options file must start on a separate line but can be continued on the next line using the standard VMS continuation character (-). The file can contain comments prefaced either by the VMS comment character (!) or the Ada comment indicator (--). Commands and qualifiers can be abbreviated as long as they remain unique.

The format for a command line in an options file is as follows:

```
[command_name(<qualifier>) [parameter]] [!comment | --comment]
```

Mandatory options file

As delivered, the TeleGen2 system requires that an options file be used to define certain user environment values. The user can modify the TD1750 run-time support modules to define these values and thereby eliminate the need to include them in an options file. The run-time definitions can always be overridden for individual applications by redefining the values in an options file. When using the options file, the user must specify values to be assigned to various symbols recognized by TD1750 modules and defined using linker DEFINE commands. The small- and medium-size memory models represented by the standard run-time sublibraries TS1750RTL and TS1750RTL_EXP require one set of DEFINE commands, while the extended memory model represented by the run-time sublibrary TS1750RTL_SPS requires another set. Each command has the form:

```
DEFINE/<symbol>=<value>
```

where <value> is specified by the user. Tables 6 and 7 below present the two sets of symbols along with the type of value appropriate to each symbol.

Table 6. Linker DEFINE symbols - standard memory model

XMACBSAD	-- Base address of the stack. This is the lowest -- address the stack can ever reach.
XMACBSSZ	-- Size of stack in words. XMACBSAD + XMACBSSZ -- defines the starting location of the stack.
XMEMGLHA	-- Base address of Ada heap.
XMEMGLHS	-- Size of Ada heap in words.
XMACDFSW	-- Default (initial) status word.
XMACDFSS	-- Default stack size for tasks whose 'Storage_Size attribute -- has not been specified *or* whose 'Storage_Size attribute is -- greater than 3000.
XMACDFIM	-- Default interrupt mask.

Table 7. Linker DEFINE symbols - extended memory model

```
XMACBSAD    -- Base address of main program stack. This is the
            -- lowest address the main program stack can reach.

XMACBSSZ    -- Size of main program stack in words. XMACBSAD +
            -- XMACBSSZ defines the starting location of the stack.

XMACESSZ    -- Size of elaboration stack. The elaboration stack
            -- allocated from the global heap and used for the
            -- duration of elaboration. It is returned to the
            -- heap at the conclusion of elaboration.

XMACCSSZ    -- Size of task RSP cluster stacks. Each time a task is
            -- activated, a stack is created in the RSP cluster. This stack
            -- is used while a task is executing code inside the RSP.

XMACDFSS    -- Default stack size for tasks whose 'Storage_Size attribute
            -- has not been specified or whose 'Storage_Size attribute is
            -- greater than 3000.

XMACDFIM    -- Default interrupt mask.

XMEMGLHA    -- Global Ada heap base address. This is the address
            -- of the heap that is addressable from all clusters.

XMEMGLHS    -- Global Ada heap size.

XMEMC0HA  )
XMEMC1HA  |  -- Cluster local heap base addresses. For each
XMEMC2HA  |  -- cluster there exists a "local" heap that is only
    ..    |  -- addressable while executing code in that cluster.
          |  -- XMEMCnHA defines the base address of the local
XMEMCFHA  )  -- heap for cluster n. If cluster n is not being used,
             -- then its base address should be set to zero.

XMEMC0HS  )
XMEMC1HS  |
XMEMC2HS  |  -- Cluster local heap size in words.
    ..    |
XMEMCFHS  )

XMEMAS0C  )  -- Address state to cluster mapping. These values
XMEMAS1C  |  -- represent the mapping of address states into
XMEMAS2C  |  -- clusters. The value assigned to XMEMASxC should
    ..    |  -- be the cluster n that the address state x belongs
          |  -- to. Any address state that does not map to a
XMEMASFC  )  -- cluster should have a value of -1 (%XFFFF).
```

Low memory is reserved for the PSP0 platform-specific run-time module, which the user must place at page 0 using a LOCATE command. Once this module has been modified and its size is known, other symbols can be defined so as to not overlap the space in which PSP0 is located. Additional linker commands are required if extended memory is to be used.

3.3. Linker options and their qualifiers

The available linker options and their qualifiers are summarized in Table 8. Defaults are shown in boldface.

Table 8. Linker options and their qualifiers

ADDRESS_STATE /INSTRUCTIONS ¦ /OPERANDS /AS=<address_state> [/REGION=<name>] [/PAGE_REGISTER=<n>] [/LOW_BOUND=<address>] [/HIGH_BOUND=<address>] [/WINDOW[=<n>]]	-- Assign physical memory locations -- to address state page registers
BLOCK_PROTECTION	-- Output block protection records
CLUSTER /NUMBER=<cluster_id> /AS=(<n>,...)	-- Associate an address state with a cluster
DEFINE /<symbol_name>=<value>	-- Specify link-time values for symbols.
EXIT	-- Terminate options list.
INPUT [/MAIN | /SPEC | /BODY | /OFM] [/WORKING_SUBLIB] [/NOSEARCH]	-- Identify object modules to be linked and -- specify the search path. <lib_comp>
LOCATE [/CONTROL_SECTION=CODE | GATE | DATA | LTRL | DESC] [/COMPONENT_NAME= <lib_comp> [/SPEC | /BODY | /OFM | /MAIN]] [/AT=<address>] [/IN=<region>] [/INSTRUCTIONS | /OPERANDS] [/AFTER_SECTION=<csect>] [/AFTER_COMPONENT=<lib_comp>] [/ALIGNMENT=<value>]	-- Specify addresses for control sections.
MAP [/[NO]IMAGE] [<file>]	-- Control link map generation.
MEMORY [/SIZE=<n>] [/DUAL] [/INSTRUCTIONS_SIZE=<n>] [/OPERANDS_SIZE=<n>]	-- Specify the size of memory in the target.
OUTPUT [/LOAD_MODULE[=<file>]] [/OBJECT_FORM=[<lib_comp>]]	-- Specify output format
QUIT	
REGION /LOW_BOUND=<address> /HIGH_BOUND=<address> [/FILL=<value>] [/NOFILL] [/INSTRUCTIONS | /OPERANDS] [/CONTROL_SECTION=CODE | GATE | DATA | LTRL | DESC] /UNUSED | <name>	-- Define and name memory regions.
TITLE <string>	-- Put title string as a header in EF file and link map

ADDRESS_STATE
/INSTRUCTIONS | /OPERANDS
/AS=<address_state>
[/REGION=<name>]
[/PAGE_REGISTER=<n>]
[/LOW_BOUND=<address>]
[/HIGH_BOUND=<address>]
[/WINDOW[=<n>]]

This option is used to assign physical memory locations to the page registers of a specific address state for applications using extended memory. The memory locations can be explicitly defined in the command, or the /REGION option can be used to define the locations implicitly.

`/INSTRUCTIONS` `/OPERANDS` — Specifies whether the command applies to the instruction page registers or to the operand page registers.

`/AS` — Specifies the address state that is being defined. Each number, 0 through 15, specifies an address state in the target machine.

`/REGION` — Specifies the named region used to set the page registers. The command uses as many page registers as needed to span the region. Regions included in multiple address states must be assigned the same page registers across state partition sets. The Linker outputs an error message if it cannot carry out this assignment successfully.

`/PAGE_REGISTER` — Specifies the starting page register to be used by the command. If a specific page register is not stated in the ADDRESS_STATE option, the linker uses the first unused page register found from the start of the address state.

`/LOW_BOUND` `/HIGH_BOUND` — The bounds of the physical memory to be assigned to sequential page registers. The command uses as many page registers as needed to span the memory specified. If only a low bound is specified, one page register will be assigned by the command. An <address> can be specified using a decimal value, a hexadecimal based literal in Ada syntax (16#hexadecimal#), or an unsigned hexadecimal value in VMS format (%Xhexadecimal).

`/WINDOW` — Causes the linker to mark the page register specified by the /PAGE_REGISTER qualifier as reserved for windowing. This is accomplished by marking the appropriate entry in the window reservation table, a table that exists in the run-time system but which the linker initializes at link time. The <value> option on the window

tells the TD_Window package how the page register should be used. A <value> greater than 100 (decimal) indicates that the page register is part of a group of page registers that spans more than one address state. This is how the linker establishes a window with visibility across only those address states that make up a state partition set.

If a page register has not been specified, windowing is enabled for the first unused page register in the address state. The /WINDOW qualifier is valid with the /OPERANDS qualifier and is incompatible with the /INSTRUCTIONS qualifier.

Local visibility

If the compilation unit residing in the current address state is to be visible only to one range of page registers, the value passed should be the address state in which it will be visible plus one. For instance, if the compilation unit is to be visible in address state 3, <value> should be 4. Thus, if four page registers are to be used for windowing in address state two, and three are requested at run time (through use of the windowing routines), those three page registers are visible within only the address state specified.

Global visibility

If the compilation unit is to be globally visible, the value %X100 should be passed as <value>. Globally visible units are taken as containing a "pool" of address state-page register pairs from which the windowing package will pull registers for windows. For instance, when three page registers are used to window, those three page registers will be visible from *all* address states when this value is specified.

Private visibility

If the compilation unit is to be visible on a "private" basis, a value in the range %X200..%X2FF should be specified. When windowing is requested, all address state-page register pairs that match the same value (in the range %X200 to %X2FF) are made visible in their appropriate address states. Only the first 255 values are allowed (since there can only be 255 combinations).

BLOCK_PROTECTION

This option specifies that block protection records are to be output in the load module. Block protection is set for each used 1024-word section of memory containing control sections of type CODE, GATE, DESC, and/or LTRL. The linker will not set block protection for a 1K section containing any portion of a DATA control section. To ensure that all CODE, GATE, DESC, and LRTL control sections are write protected, use the LOCATE and REGION commands to group them together, apart from DATA control sections.

The default is no block protection.

CLUSTER
/NUMBER=<cluster_id>
/AS=(<n>,...)

This option is used to associate address states with a cluster so that the linker can identify the types of procedure calls that can occur, i.e.:

1. Calls within an address state
2. Calls between address states within a cluster
3. Calls between address states in different clusters

`/NUMBER` Specifies the cluster as a number in the range 0..N, where N corresponds to the number of clusters used in the application.

`/AS` Specifies the address states, separated by commas, to be associated with the given cluster. Each address state is represented by a number in the range 0..15 corresponding to the 16 available address states.

DEFINE
/<symbol>=<value>

This option enables the user to define global symbols referenced in the Ada low-level support modules and other imported OF modules. Multiple DEFINE options can be used.

`<symbol>` Is specified by the user or is one of the names that must be defined when the standard TD1750 run-time environment is used. Tables 7 and 8 list the symbols to be defined in the mandatory linker options file.

`<value>` Is specified by the user, by using a decimal value, a hexadecimal based literal in Ada syntax (16#hexadecimal#), or an unsigned hexadecimal value in VMS format (%Xhexadecimal). Use only locations after the PSP0 run-time module.

EXIT

This option is used to specify the end of user input interactively or from an options file. After the EXIT option, no options are processed and the link operation is performed. An end-of-file is equivalent to the EXIT option.

INPUT
[/MAIN | /SPEC | /BODY | /OFM]
[/WORKING_SUBLIB]
[/NOSEARCH]
<lib_comp>

This option specifies the name of the Ada library component to be linked, its usage, and its search path. Multiple INPUT options can be used. By default, the object associated with the specified name is included in the linked output.

When the default search path for a specified library component is used, the linked output includes all compilation units in the current library that comprise the extended family of the component. The search path qualifiers /NOSEARCH and /WORKING_SUBLIB can be used to override the default and include only the component itself or only those components of the extended family in the working sublibrary.

The following qualifiers specify the kind of library component input:

/SPEC : library unit
/BODY : secondary unit
/OFM : imported OF
/MAIN : main program library unit

/SPEC
/BODY
/OFM

Use /SPEC only for library units without bodies, since those are the only library units that have object associated with them. For all other compilation units, use /BODY, or /MAIN, if the unit is the main program unit. Using /SPEC or /BODY only makes sense when combined with /WORKING_SUBLIB since the library or secondary unit is automatically brought into the link if needed and excluded if not needed. /WORKING_SUBLIB instructs the linker to use the version of the library or secondary unit located in the working sublibrary rather than any other version in another sublibrary.

/MAIN

Specifies that the library component is to be the main program in the link. The library component name specified in the first INPUT option is used as the default OUTPUT and MAP name, unless the user specifies other names. Specification of both /OFM and /MAIN is an error. Do not use a <main> parameter on the command line for a link-only operation if the name of the unit is

specified in the linker options file with INPUT/MAIN.

/WORKING_SUBLIB — Specifies to include in the linked output only those extended family components in the current working sublibrary.

/NOSEARCH — Specifies to include in the linked output only the specified input component. This qualifier is only effective for imported OF, as when used in combination with /OFM.

When the name of the OUTPUT and MAP files are not specified, the linker names the files after the first INPUT option's named component.

LOCATE
[/CONTROL_SECTION=CODE | GATE | DATA | LTRL | DESC]
[/COMPONENT_NAME= <lib_comp>
[/SPEC | /BODY | /OFM | /MAIN]] [/AT=<address>] [/IN=<region>]
[/INSTRUCTIONS | /OPERANDS]
[/AFTER_COMPONENT=<lib_comp>]
[/AFTER_SECTION=CODE | GATE | DATA | LTRL | DESC]
[/ALIGNMENT=<value>]

This option specifies the location of a control section in physical memory or within a named region of physical memory.

/CONTROL_SECTION — Specifies the name of the control section to be located. The valid control section names for the 1750A are CODE, GATE, DATA, LTRL, and DESC. The type used by the LOCATE can alternatively be specified by the /CONTROL_SECTION qualifier in the REGION command if the /IN qualifier is used. If /CONTROL_SECTION is not specified, the linker assumes that the LOCATE command refers to all control sections not explicitly located by other LOCATE commands.

/COMPONENT_NAME — Specifies the name of the library component that is to be located and has one of the following qualifiers:

/SPEC : library unit (default)
/BODY : secondary unit
/OFM : imported unit
/MAIN : main program unit

/SPEC
/BODY
/OPM
/MAIN

/SPEC should be used only for library units without bodies, since those are the only library units that have object associated with them. All other compilation units should be located using the /BODY qualifier or the /MAIN qualifier if it is a main program unit. If no /COMPONENT_NAME qualifier is used with the LOCATE option, the linker assumes that LOCATE refers to all compilation units not explicitly located by other LOCATE commands.

/AT

Specifies that the image is to be located at the specified memory location. The address specified must be a valid 1750A memory location. The <address> can be specified by using a decimal value, a hexadecimal based literal in Ada syntax (16#hexadecimal#), or an unsigned hexadecimal value in VMS format (%Xhexadecimal). (%Xhexadecimal). Use only locations after the PSP0 run-time module.

/IN

Specifies that the image is to be located in the first available location in the named memory region. The memory region, <region>, must have been defined by a previous REGION command. This qualifier can be used in conjunction with the /AFTER_COMPONENT qualifier to locate compilation units in a specific order in a memory region and/or with the /ALIGNMENT qualifier to specify alignment. The type of the control sections located in the region can be specified in the REGION declaration. The linker outputs an error if the type in the LOCATE command does not match the type of the region.

/INSTRUCTIONS

Locates the control section or component in the instruction memory bank when dual memory is being used.

/OPERANDS

Locates the control section or component in the operand memory bank when dual memory is being used.

The linker does not allow the user to place instruction control sections or components (CODE, GATE) in operand memory nor operand control sections or components (DATA, LTRL, DESC) in instruction memory.

The MEMORY/DUAL statement must precede any LOCATE statement in a dual memory link. If neither /INSTRUCTIONS nor /OPERANDS are specified, the linker locates the control section according to its memory type (instructions or operands). If the user specifies a

region in which to locate a component but does not specify a control section, the component is located according to the type of region specified. If the user specifies an address using /AT, the control section or component is located at that address in the correct memory bank.

A LOCATE statement using /AFTER_SECTION or /AFTER_COMPONENT can have unintended results in a dual memory link. For example, if a CSECT A is operand memory located at 1000 through 10FF in operand memory and the user locates the instruction memory CSECT B using /AFTER_SECTION=A, CSECT B will be placed in instruction memory starting at 1100, i.e., after CSECT A but in the instruction rather the operand memory bank.

/AFTER_COMPONENT Specifies that the image is to be located immediately after the specified component (using default alignment or /ALIGNMENT). The specified component must have been located explicitly using /AT in a previous LOCATE command. /AFTER_COMPONENT can be used in conjunction with /AFTER_SECTION.

/AFTER_SECTION Specifies that the image is to be located immediately after the specified section (using default alignment or /ALIGNMENT). The specified control section must have been located explicitly in a previous LOCATE command. /AFTER_SECTION may be used in conjunction with /AFTER_COMPONENT.

/ALIGNMENT Specifies the byte alignment of the image, if any. If no alignment is specified, control sections on the 1750A start in consecutive word locations. <value> can be specified using the Ada syntax for decimal or based literals or by using the VMS unsigned hexadecimal format %X value.

The linker does not support the ability to locate a control section based on the relative placement of the specified section. The /AFTER_COMPONENT and /AFTER_SECTION qualifiers are restricted to use where an absolute address for the specified unit location has been defined using an /AT qualifier. Although the /AFTER_SECTION and /AFTER_COMPONENT qualifiers can be used for the ordering of compilation units when the specified section has an absolute location, this technique may result in fragmentation of memory regions. The use of the /ALIGNMENT qualifier may cause the same fragmentation problem. The best approach for ordering units within a CSECT is through the placement of the LOCATE commands within the options file.

The use of /IN, /AFTER_SECTION, and /AFTER_COMPONENT in the same LOCATE command is not allowed. Otherwise, /IN, /AFTER_COMPONENT, /AFTER_SECTION, and /ALIGNMENT can be used in any combination. The use of /IN, /AFTER_COMPONENT, /AFTER_SECTION, or /ALIGNMENT in the same LOCATE command as the /AT qualifier is an error.

Control sections are located according to the sequence of the LOCATE commands. To prevent unexpected overlapping of control sections, explicit LOCATE commands should precede any default LOCATE command, i.e., a command without a component name specified. A LOCATE command with no component name and no control section name causes the linker to locate all non-located sections starting at the base specified. The sections will be individually located to fit around any sections explicitly located by previous commands. For example, the following two commands would locate all the control sections for component Exc_Monitor:

```
LOCATE/COMPONENT=EXC_MONITOR/CONTROL_SECTION=CODE/AT=%X0038
LOCATE/AT=%XF12D
```

Once the CODE section had been located, any DATA, LTRL, GATE, and DESC sections for the component would be located individually starting at %XF12D, going before or after but not overlapping the CODE section.

Components in control sections can be located in multiple regions for use in multiple address states. Each component must be explicitly located by specifying its name, type, and control section. Alternatively, regions can be shared between address states.

An error occurs if:

1. An explicitly located control section does not fit into the designated region.

2. Multiple control sections are located so as to overlap.

3. A section is explicitly located so as to extend beyond the end of memory.

4. Control sections to be located by default do not fit into the space available from the designated address to the end of memory.

In addition, there is an implementation restriction that composite data objects that straddle logical address %X8000 may cause an exception when they are accessed. The exception may also occur from adding an offset to a base address, where the base address is located before %X8000, and the result of the addition yields an address beyond %X8000. The reason for this is that 1750A does not have unsigned arithmetic. %X8000 is the boundary beyond which the signed bit turns a positive number into a negative number. A linker

apprentice warning notifies you of the potential for this exception in a particular region. A workaround is to move the region manually by editing the linker options file.

MAP
[/[NO]IMAGE]
[<file>]

This option is used to request and control a link map listing. This option operates in the same way as the /MAP command line qualifier. A command line /MAP qualifier supercedes any MAP options in an options file. The default is /NOMAP. /NOMAP can be used to suppress MAP options specified in an options file.

`/IMAGE` Generates a memory image listing in addition to the map listing. The linker writes the image listing to the same file as the link map listing. This is the only optional section of the listing. The default is /NOIMAGE.

`<file>` Is the optional VMS file specification for the output. If <file> does not include a file extension, the linker will append a default extension of ".MAP". If the user does not specify an output file, the linker writes the listing to <unit>.MAP, where <unit> is the name of the main program (if present), the name specified as the command line parameter, or the name of the first INPUT option, modified as necessary to form a valid VMS file specification.

MEMORY
/SIZE=<n>
[/DUAL]
[/INSTRUCTIONS_SIZE=<n>]
[/OPERANDS_SIZE=<n>]

The MEMORY option is used to specify the size of physical memory in words in the target machine and to indicate if dual code and data memory banks are implemented. If a memory size is specified, the linker reports an error if a control section is located outside the bounds of the specified memory. If the linker is to build the unused memory table for use with the window packages, the memory size must be specified. In a dual memory link, the MEMORY command must precede any LOCATE or REGION commands.

`/SIZE` Specifies the number of words in physical memory available in the target. When MEMORY/SIZE is omitted, the linker defaults to the size of 65,536 words. /SIZE is incompatible with /INSTRUCTIONS_SIZE or /OPERANDS_SIZE, but it can be used without them in a dual memory link to set both the instruction and operand memory banks to the same size.

/DUAL — Specifies that dual code and data memory banks are implemented on the target machine. The linker then allows code and data to overlap in physical memory.

/INSTRUCTIONS_SIZE — Specifies the instruction memory bank size. The default is 64K. This qualifier is only valid for dual memory links and is incompatible with the /SIZE qualifier.

/OPERANDS_SIZE — Specifies the operand memory bank size. The default is 64K. This qualifier is only valid for dual memory links and is incompatible with the /SIZE qualifier.

OUTPUT
[/LOAD_MODULE[=<file>]
[/OBJECT_FORM=[<lib_comp>]]

This option specifies the type, format, and name of the linked output. Only one OUTPUT option can be used.

/LOAD_MODULE — Specifies that one output of the linker should be an executable load module. /LOAD_MODULE is the default. <file> is the optional VMS file specification for the output. If <file> does not include a file type, the linker will append a .SO extension for the default ECSPO load module format, or a .I3E extension for IEEE-695 format if /IEEE has been specified. If no output file is specified, the linker will write the linked output to <unit>.SO or <unit>.I3E, as appropriate, where <unit> is the Ada name of the main program unit (if present), the name specified as the command line parameter, or the name specified as the first INPUT option, modified as necessary to form a valid VMS file specification.

/OBJECT_FORM — Specifies that the linker should produce linked OF and put it into the Ada library as an object form module component. <lib_comp> is any valid library component name. If the user does not specify an output file, the linker puts the output of the link into the library as the object form module: <unit>, which is the Ada name of the main program (if present), the name specified as the command line parameter, or the name specified as the first INPUT option.

QUIT

This option is used to terminate the entry of interactive options and to abort the link operation.

REGION
/LOW_BOUND=<address>
/HIGH_BOUND=<address>
[/CONTROL_SECTION=CODE | GATE | DATA | LTRL | DESC]
[/FILL=<value>] [/NOFILL]
[/INSTRUCTIONS | /OPERANDS]
/UNUSED | <name>

This option names a memory region and specifies its location in physical memory. This named memory region can be used in subsequent LOCATE and ADDRESS_STATE options. In addition, the REGION option can be used to specify an unused memory region and to specify a fill value for unused memory within a region. In a dual memory link, a region can be assigned to either an instruction or an operand memory bank or to both. Multiple REGION options can be used.

`<name>` Is a user-specified name for a region of memory. A name is required for every memory region except one marked as unused by means of the /UNUSED qualifier.

`/LOW_BOUND` `/HIGH_BOUND` Are the bounds of the memory region. These qualifiers and associated values are mandatory. An <address> can be specified as a decimal value, a hexadecimal based literal in Ada syntax (16#hexadecimal#), or an unsigned hexadecimal value in VMS format (%Xhexadecimal).

`/CONTROL_SECTION` Specifies the control sections to be included in the region. The valid control section types are CODE, GATE, DATA, LTRL, and DESC. This qualifier can be used to include a particular type of control section in a region.

`/FILL` `/NOFILL` Is an optional qualifier that specifies the value to be used to fill the unused locations in the region, if any. When /FILL is used with the /UNUSED qualifier, all memory locations in the region are filled with the specified value. The default fill-value for regions of type CODE or GATE is 7400, the branch to self instruction. The default fill-value for regions of type DATA, LTRL, or DESC is 0. The default is /NOFILL, which specifies that the unused regions should not be filled.

`/UNUSED` Is an optional qualifier used to specify that the linker is not to use this memory region. Unused memory regions do not need to be named. Unused regions are filled with word value 0, by default. This default value can be overridden by using /NOFILL or /FILL=<value>.

`/INSTRUCTIONS` Assigns the region to an instruction memory bank in a dual memory link. The memory type specified must not

conflict with the memory type of any control section specified.

`/OPERANDS` Assigns the region to an operand memory bank in a dual memory link. The memory type must not conflict wi'h the memory type of any control section specified.

If /INSTRUCTIONS or /OPERANDS is omitted in a dual memory link, the region is assigned to the memory bank corresponding to the control section specified. If no control section is specified, two duplicate regions are created, one in the instruction memory bank and one in the operand memory bank.

The MEMORY command must precede the REGION command in a dual memory link.

TITLE <string>

The TITLE option places the specified string in the Execute Form file as the first record, which is a header record in the ECSPO object module definition. Only one TITLE command is effective in an options file. The linker ignores any TITLE options occurring after the first one. If a link map is requested, the TITLE option places the specified string on the header record atop each page of the link map.

`<string>` Is a title string with a maximum length of 40 characters. Longer strings are truncated to 40 characters. Enclose the string in quotation marks (" ") if it contains spaces. The string is case sensitive, i.e., the linker does not change the case(s) of the characters in the string.