

AD-A240 762**INTATION PAGE**Form Approved
OPM No. 0704-0188**2**

• 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
g this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE		3. REPORT TYPE AND DATES COVERED Final: 31 Jul 1991 to 01 Jun 1993	
4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: InterACT Corporation, InterACT Ada 1750A Compiler System, Rel 3.5, MicroVAX 3100 Cluster (Host) to InterACT MIL-STD-1750A Instruction Set Architecture Simulator (Target), 910705S1.11191				5. FUNDING NUMBERS DTIC SEP 19 1991 S D D	
6. AUTHOR(S) National Institute of Standards and Technology Gaithersburg, MD USA					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) National Institute of Standards and Technology National Computer Systems Laboratory Bldg. 255, Rm A266 Gaithersburg, MD 20899 USA				8. PERFORMING ORGANIZATION REPORT NUMBER NIST90ACT520_1_1.11	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Pentagon, RM 3E114 Washington, D.C. 20301-3081				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) InterACT Corporation, InterACT Ada 1750A Compiler System, Rel 3.5, Gaithersburg, MD, MicroVAX 3100 Cluster (Host) to InterACT MIL-STD-1750A Instruction Set Architecture Simulator, Release 2.3 (Bare Machine). (Target), ACVC 1.11					
91-11069 					
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 UNCLASSIFIED	18. SECURITY CLASSIFICATION UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT		

AVF Control Number: NIST90ACT520_1_1.11

DATE COMPLETED

BEFORE ON-SITE: 1991-06-07

AFTER ON-SITE: 1991-07-05

REVISIONS: 1991-07-31

Ada COMPILER

VALIDATION SUMMARY REPORT:

Certificate Number: 910705S1.11191

InterACT Corporation

InterACT Ada 1750A Compiler System, Release 3.5

MicroVAX 3100 Cluster => InterACT MIL-STD-1750A Instruction Set
Architecture Simulator, Release 2.3 (Bare Machine)

Prepared By:

Software Standards Validation Group

National Computer Systems Laboratory

National Institute of Standards and Technology

Building 225, Room A266

Gaithersburg, Maryland 20899

Accession For	J
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Date of Report	
Availability Codes	
Dist	Avail and/or Special
A-1	

1991
JUL 10 1991
NIST

AVF Control Number: NIST90ACT520_1_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 1991-07-05.

Compiler Name and Version: InterACT Ada 1750A Compiler System, Release 3.5

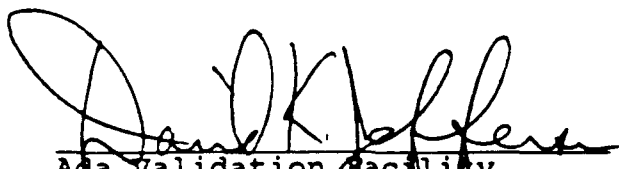
Host Computer System: MicroVAX 3100 Cluster running under VAX/VMS, Version 5.2

Target Computer System: InterACT MIL-STD-1750A Instruction Set Architecture Simulator, Release 2.3 (Bare Machine)

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

As a result of this validation effort, Validation Certificate 910705S1.11191 is awarded to InterACT Corporation. This certificate expires on 01 March 1993.

This report has been reviewed and is approved.

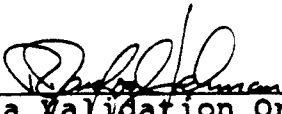


Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)



Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group

Computer Systems Laboratory (CLS)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899



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



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

APPENDIX A

Declaration of Conformance

Customer: InterACT Corporation

Ada Validation Facility: National Institute of Standards & Technology

ACVC Version: 1.11

Certificate Awardee InterACT Corporation

Ada Implementation

Ada Compiler Name: InterACT Ada 1750A Compiler System

Version: 3.5

Host Computer System: MicroVAX 3100 Cluster /VMS 5.2

Target Computer System: InterACT MIL-STD-1750A Instruction Set Architecture
Simulator Release 2.3 (bare machine)

Customer's Declaration

I, the undersigned, representing InterACT declare that InterACT has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration.

Signature

Date

TABLE OF CONTENTS

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

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 Fort Royal Road
Springfield VA 22161

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

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

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

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

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

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 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, Validation consisting of the test suite, the support programs, the ACVC Capability user's guide and the template for the validation summary (ACVC) report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification Office system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including

arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].

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

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 94 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. 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 91-05-03.

E28005C	B28006C	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. The inapplicability criteria for some tests are explained in documents issued by 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	C45532B
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`; for this implementation, there is no such type.

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

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.

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

D64005G uses 17 levels of recursive procedure calls nesting; this test exceeds the linkable size of 64KBytes.

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

C96005B uses values of type `DURATION`'s base type that are outside the range of type `DURATION`; for this implementation, the ranges are the same.

CA2009C and CA2009F check whether a generic unit can be instantiated before its body (and any of its subunits) is compiled; this implementation requires that generic bodies be located in the same file or precede the instantiation.

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

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

BL8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

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 18 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.

B33301B B55A01A B83E01C B83E01D B83E01E BA1001A BA1101B BC1109A
BC1109C BC1109D

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

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.

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).

EE3412C was graded passed by Test Modification as directed by the AVO. This test assumes that the support package REPORT uses TEXT_IO, and that thus calls to REPORT.SPECIAL_ACTION will increment the line count on the standard output file. But REPORT was modified to use the implementation-defined string I/O package named STRING_OUTPUT instead of TEXT_IO, because TEXT_IO is large in terms of object code size. STRING_OUTPUT is significantly smaller than TEXT_IO, and provides for the output needs of REPORT while allowing for the executable images of the tests to fit within a 64KByte memory limit. Because STRING_IO operations do not affect the status of TEXT_IO files--i.p., the line count for standard output file is unchanged--, line 46 of the test was changed as

follows:

```
from:  IF LINE /= C+2 THEN
to:    IF LINE /= C+1 THEN
```

Although REPORT is not a test, the modifications to it is recorded here to complete the record and to allow for accurate replication of this test environment. REPORT body was modified to use a package named STRING_OUTPUT rather than TEXT_IO because TEXT_IO is large in terms of compiled object code size. STRING_OUTPUT is significantly smaller than TEXT_IO and provides for the output needs of REPORT while allowing for the executable image of the tests to fit within a 64KByte memory limit.

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.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link described above, and run. The results were captured on the host computer system.

For technical information about this Ada implementation, contact:

Ms. Gail Ward
InterACT Corporation
417 5th Avenue
New York, New York, U.S.A. 10016

For sales information about this Ada implementation, contact:

Mr. Rich Colucci
InterACT Corporation
417 5th Avenue
New York, New York, U.S.A. 10016

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	3466	
b) Total Number of Withdrawn Tests	94	
c) Processed Inapplicable Tests	610	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	610	(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 directly onto the host computer.

The Ada source files are compiled on a MicroVAX 3100 Cluster under VAX/VMS using the InterACT Ada 1750A Compiler System. The Ada main programs are then linked on the MicroVAX Cluster using InterACT 1750A Linker which produces a load module in InterACT's own load format.

This load format is loaded and then executed within the InterACT MIL-STD-1750A Instruction Set Architecture Simulator which also executes on the MicroVAX 3100 Cluster. The Symbolic Debugging and Simulation System runs a set script consisting only of "load", "go", and "exit" commands. The MIL-STD-1750A Instruction Set Architecture Simulator is a complete instruction set simulator for the MIL-STD-1750A architecture. The 1750A Console Output instruction in the MIL-STD-1750A Instruction Set Architecture Simulator is defined to write character output (representing the Ada standard output) to a dedicated Ada output file. The dedicated Ada output file contains the output from the ACVC tests.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The InterACT MIL-STD-1750A Instruction Set Architecture Simulator, Release 2.3 (Bare Machine) [target computer system] runs on the host computer system. The executable images were transferred to the target computer system by the communications link described above, and run. The results were

captured on the host computer system.

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

For all tests the following explicit option was invoked:

```
/library=<library_name>
```

In addition to the above, the following explicit option was invoked for the B tests and E tests:

```
/list
```

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

APPENDIX A

MACRO PARAMETERS

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

Macro Parameter	Macro Value
\$MAX_IN_LEN	126 -- Value of V
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
\$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
\$ACC_SIZE	16
\$ALIGNMENT	1
\$COUNT_LAST	2_147_483_647
\$DEFAULT_MEM_SIZE	65536
\$DEFAULT_STOR_UNIT	16
\$DEFAULT_SYS_NAME	MIL_STD_1750A
\$DELTA_DOC	1.0/2.0**(SYSTEM.MAX_MANTISSA)
\$ENTRY_ADDRESS	12
\$ENTRY_ADDRESS1	13
\$ENTRY_ADDRESS2	14
\$FIELD_LAST	35
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
\$FORM_STRING	""
\$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
\$GREATER_THAN_DURATION	214_748.3647
\$GREATER_THAN_DURATION_BASE_LAST	214_749.3647
\$GREATER_THAN_FLOAT_BASE_LAST	2#0.1111111_11111111_111111#E127
\$GREATER_THAN_FLOAT_SAFE_LARGE	0.999999E128
\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	0.0
\$HIGH_PRIORITY	255

\$ILLEGAL_EXTERNAL_FILE_NAME1	ILLEGAL_FILE_NAME_1
\$ILLEGAL_EXTERNAL_FILE_NAME2	ILLEGAL_FILE_NAME_2
\$INAPPROPRIATE_LINE_LENGTH	-1
\$INAPPROPRIATE_PAGE_LENGTH	-1
\$INCLUDE_PRAGMA1	PRAGMA INCLUDE("A28006D1.TST")
\$INCLUDE_PRAGMA2	PRAGMA INCLUDE("B28006F1.TST")
\$INTEGER_FIRST	-32768
\$INTEGER_LAST	32767
\$INTEGER_LAST_PLUS_1	32768
\$INTERFACE_LANGUAGE	ASSEMBLY
\$LESS_THAN_DURATION	-214_748.3648
\$LESS_THAN_DURATION_BASE_FIRST	-214_749.3648
\$LINE_TERMINATOR	' '
\$LOW_PRIORITY	0
\$MACHINE_CODE_STATEMENT	NULL;
\$MACHINE_CODE_TYPE	NO_SUCH_TYPE
\$MANTISSA_DOC	31
\$MAX_DIGITS	9
\$MAX_INT	2147483647
\$MAX_INT_PLUS_1	2147483648
\$MIN_INT	-2147483648
\$NAME	NO_SUCH_INTEGER_TYPE
\$NAME_LIST	MIL_STD_1750A
\$NAME_SPECIFICATION1	NAME_SPEC_1
\$NAME_SPECIFICATION2	NAME_SPEC_2
\$NAME_SPECIFICATION3	NAME_SPEC_3

\$NEG_BASED_INT	16#FFFFFFFFE#
\$NEW_MEM_SIZE	65536
\$NEW_STOR_UNIT	16
\$NEW_SYS_NAME	MIL_STD_1750A
\$PAGE_TERMINATOR	' '
\$RECORD_DEFINITION	NEW INTEGER;
\$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
\$TASK_SIZE	16
\$TASK_STORAGE_SIZE	1024
\$TICK	0.000_100
\$VARIABLE_ADDRESS	16#1000#
\$VARIABLE_ADDRESS1	16#1800#
\$VARIABLE_ADDRESS2	16#2000#
\$YOUR_PRAGMA	N_A

APPENDIX B

COMPILATION SYSTEM OPTIONS

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

Chapter 4

The Ada Compiler

The Ada Compiler translates Ada source code into MIL-STD-1750A object code.

Diagnostic messages are produced if any errors in the source code are detected. Warning messages are also produced when appropriate.

Compile, cross-reference, and generated assembly code listings are available upon user request.

The compiler uses a program library during the compilation. An internal representation of the compilation, which includes any dependencies on units already in the program library, is stored in the program library as a result of a successful compilation.

On a successful compilation, the compiler generates assembly code, invokes the InterACT 1750A Assembler to translate this assembly code into object code, and then stores the object code in the program library. (Optionally, the generated assembly code may also be stored in the library.) The invocation of the Assembler is completely transparent to the user.

4.1. The Invocation Command

The Ada Compiler is invoked by submitting the following VAX/VMS command:

```
$ ada1750{qualifier} source-file-spec
```

4.1.1. Parameters and Qualifiers

Default values exist for all qualifiers as indicated below. All qualifier names may be abbreviated (characters omitted from the right) as long as no ambiguity arises.

source-file-spec

This parameter specifies the file containing the source text to be compiled. Any valid VAX/VMS filename may be used. If the file type is omitted from the specification, file type *ada* is assumed by default. If this parameter is omitted, the user will be prompted for it. The format of the source text is described in Section 4.2.

/list
/nolist (default)

The user may request a source listing by means of the qualifier **/list**. The source listing is written to the list file. Section 4.3.2 contains a description of the source listing.

If **/nolist** is active, no source listing is produced, regardless of any **LIST** pragmas in the program or any diagnostic messages produced.

In addition, the **/list** qualifier provides generated assembly listings for each compilation unit in the source file. Section 4.3.6 contains a description of the generated assembly listing.

/xref
/noxref (default)

A cross-reference listing can be requested by the user by means of this qualifier. If **/xref** is active and no severe or fatal errors are found during the compilation, the cross-reference listing is written to the list file. The cross-reference listing is described in Section 4.3.4.

/library=file-spec
/library=ada1750_library (default)

This qualifier specifies the current sublibrary and thereby also specifies the current program library which consists of the current sublibrary through the root sublibrary (see Chapter 2). If the qualifier is omitted, the sublibrary designated by the logical name *ada1750_library* is used as the current sublibrary.

Section 4.4 describes how the Ada compiler uses the current sublibrary.

/configuration_file=file-spec
/configuration_file=ada1750_config (default)

This qualifier specifies the configuration file to be used by the compiler in the current compilation.

If the qualifier is omitted, the configuration file designated by the logical name *ada1750_config* is used by default. Section 4.1.4 contains a description of the configuration file.

/save_source (default)
/nosave_source

This qualifier specifies whether the source text of the compilation unit is stored in the program library. In case that the source text file contains several compilation units the source text for each compilation unit is stored in the program library. The source texts stored in the program library can be extracted using the Ada PLU type command (see Chapter 3).

Specifying **/nosave_source** will prevent automatic recompilation by the Ada Recompiler, and is hence not recommended.

`/keep_assembly`
`/nokeep_assembly` (default)

When this qualifier is given, the compiler will store the generated assembly source code in the program library, for each compilation unit being compiled. By default this is not done. Note that while the assembly code is stored in the library in a compressed form, it nevertheless takes up a large amount of library space relative to the other information stored in the library for a program unit.

This qualifier does not affect the production of generated assembly listings.

`/check` (default)
`/nocheck[= (check_kind,...)]`

check_kind ::= `index` | `access` | `discriminant` | `length` | `range` |
`division` | `overflow` | `elaboration` | `storage` | `all`

When this qualifier is active (which is the default), all run time checks will be generated by the compiler.

When `/nocheck` is specified, the checks corresponding to the particular check kinds specified will be omitted. These kinds correspond to the identifiers defined for pragma `SUPPRESS` [Ada RM 11.7]. The default kind for `/nocheck` is `all`; that is, just specifying `/nocheck` results in all checks being suppressed.

Suppression of checks is done in the same manner as for pragma `SUPPRESS` (see Section F.2).

`/debug`
`/nodebug` (default)

When this qualifier is given, the compiler will generate symbolic debug information for each compilation unit in the source file and store the information in the program library. By default this is not done.

This symbolic debug information is used by the InterACT Symbolic Debugging System.

It is important to note that the identical object code is produced by the compiler, whether or not the `/debug` qualifier is active. There are some minor differences in the generated assembly code, due to some extra labels being generated in the debug case.

`/nooptimize`

A small portion of the optimizing capability of the compiler places capacity limits on the source program (e.g., number of variables in a compilation unit) that are more restrictive than those documented in Section F.13. If a compile produces an error message indicating that one of these limits has been reached, for example

```
*** 15625-0: Optimizer capacity exceeded. Too many names in a basic block.
```

then use of this `/nooptimize` qualifier will bypass this particular optimizing capability and allow the compilation to finish normally.

IMPORTANT NOTE: Do not use this qualifier for any other reason. Do not attempt to use it in its positive

form (/feoptimize), either with or without any of its keyword parameters. The /feoptimize qualifier as defined in the delivered command definition file is preset to produce the most effective optimization possible; any other use of it may produce either non-optimal or incorrect generated code. Similarly, do not use any other qualifiers defined in the delivered command definition file that are not documented in this manual. Such qualifiers are intended only for compiler maintenance purposes.

```
/progress
/noprogress      (default)
```

When this qualifier is given, the compiler will write a message to sys\$output as each pass of the compiler starts to run. This information is not provided by default.

Examples of qualifier usage

```
$ ada1750 navigation_constants
$ ada1750/list/xref event_scheduler
$ ada1750/prog/lib=test_versions.alb sys$user:[source]altitudes_b
```

4.1.2. The List File

The name of the list file is identical to the name of the source file except that it has the file type lis. The file is located in the current default directory. If any such file exists prior to the compilation, the newest version of the file is deleted. If the user requests any listings by specifying the qualifiers /list or /xref, a new list file is created.

The list file is a text file and its contents are described in Section 4.3.

4.1.3. The Diagnostic File

The name of the diagnostic file is identical to the name of the source file except that it has the file type err. It is located in the current default directory. If any such file exists prior to the compilation, the newest version of the file is deleted. If any diagnostic messages are produced during the compilation, a new diagnostic file is created.

The diagnostic file is a text file containing a list of diagnostic messages, each followed by a line showing the number of the line in the source text causing the message, and a blank line. There is no pagination and there are no headings. The file may be used by an interactive editor to show the diagnostic messages together with the erroneous source text (see Appendix A). The diagnostic messages are described in Section 4.3.5.

4.1.4. The Configuration File

Certain functional characteristics of the compiler may be modified by the user. These characteristics are passed to the compiler by means of a *configuration file*, which is a text file. The contents of the configuration file must be an Ada positional aggregate, written on one line, of the anonymous type *configuration_record*, which is described below. The configuration file is not accepted by the compiler in the following cases:

- the syntax does not conform with the syntax for a positional Ada aggregate of type *configuration_record*;
- a value is outside the ranges specified below;
- a value is not specified as a literal;
- `LINES_PER_PAGE` is not greater than `TOP_MARGIN + BOTTOM_MARGIN`;
- the aggregate occupies more than one line.

If the compiler is unable to accept the configuration file, an error message is issued and the compilation is terminated.

The definition of this anonymous type is

```
type OUTFORMATTING is
  record
    LINES_PER_PAGE : INTEGER range 30..100;
    --see Section 4.3.1
    TOP_MARGIN : INTEGER range 4.. 90;
    --see Section 4.3.1
    BOTTOM_MARGIN : INTEGER range 0.. 90;
    --see Section 4.3.1
    OUT_LINELENGTH : INTEGER range 80..132;
    --see Section 4.3.1
    SUPPRESS_ERRORNO : BOOLEAN;
    --see Section 4.3.5.1
  end record;
```

```
type INPUT_FORMATS is
  ( ASCII );
  --see Section 4.2
```

```
type INFORMATTING is
  record
    INPUT_FORMAT : INPUT_FORMATS;
    --see Section 4.2
    INPUT_LINELENGTH : INTEGER range 70..127;
    --see Section 4.2
  end record;
```

```
type configuration_record is
  record
    IN_FORMAT : INFORMATTING;
    OUT_FORMAT : OUTFORMATTING;
    ERROR_LIMIT : INTEGER;
    --see Section 4.3.5
  end record;
```

The Compiler System is delivered with a configuration file with the following content:

```
((ASCII, 126), (48, 5, 3, 100, FALSE), 200)
```

The name of this configuration file is passed to the compiler through the `/configuration_file` qualifier.

The OUTFORMATTING components have the following meaning:

- **LINES_PER_PAGE**: Specifies the maximum number of lines written on each page (including top and bottom margin).
- **TOP_MARGIN**: Specifies the number of lines on top of each page used for a standard heading and blank lines. The heading is placed in the middle lines of the top margin.
- **BOTTOM_MARGIN**: Specifies the minimum number of lines left blank in the bottom of the page. The number of lines available for the listing of the program is `LINES_PER_PAGE - TOP_MARGIN - BOTTOM_MARGIN`.
- **OUT_LINELENGTH**: Specifies the maximum number of characters written on each line. Lines longer than `OUT_LINELENGTH` are separated into two lines.
- **SUPPRESS_ERRORNO**: Specifies the format of error messages, see Section 4.3.5.1.

4.1.5. The Generated Assembly List File

When generated assembly list files are produced, there is one such file for each compilation unit in the source file. Generated assembly list files have a file type of `als`, and a file name of the compilation unit name suffixed with a `$s` if the compilation unit is a specification, or `$b` if the compilation unit is a body. All files are located in the current default directory. Unlike the source list file, existing generated assembly list files are not deleted upon recompilation.

Generated assembly list files are text files and their contents are described in Section 4.3.6.

4.2. The Source Text

The user submits one source text file in each compilation. The source text may consist of one or more compilation units [*Ada RM 10.1*].

On VAX/VMS the format of the source text specified in the configuration file (see Section 4.1.4) must be ASCII. This format requires that the source text is a sequence of ISO characters [ISO standard 646], where each line is terminated by one of the following termination sequences (CR means carriage return, VT means vertical tabulation, LF means line feed, and FF means form feed):

- a sequence of one or more CRs, where the sequence is neither immediately preceded nor immediately followed by any of the characters VT, LF, or FF;
- any of the characters VT, LF, or FF, immediately preceded and followed by a sequence of zero or more CRs.

In general, ISO control characters are not permitted in the source text with the following exceptions:

- the horizontal tabulation character (HT) may be used as a separator between lexical units;
- LF, VT, FF, and CR may be used to terminate lines, as described above.

The maximum number of characters in an input line is determined by the contents of the configuration file (see Section 4.1.4). The control characters CR, VT, LF, and FF are not considered part of the line. Lines containing more than the maximum number of characters are truncated and an error message is issued.

4.3. Compiler Output

The compiler may produce output in the list file, the generated assembly list file(s), the diagnostic file, and on `sys$output`. It also updates the program library if the compilation is successful (see Section 4.4).

The compiler may produce the following text output:

1. A listing of the source text with embedded diagnostic messages is written to the list file, if the qualifier `/list` is active.
2. A compilation summary is written to the list file, if `/list` is active.
3. A cross-reference listing is written to the list file, if `/xref` is active and no severe or fatal errors have been detected during the compilation.
4. A generated assembly listing of the compilation units within the source file is written to the generated assembly list file(s) if the qualifier `/list` is active, and if no errors have been detected during the compilation.
5. If there are any diagnostic messages, a diagnostic file containing the messages is written.
6. Diagnostic messages other than warnings are written to `sys$output`.

4.3.1. Format of the List File

The list file may include one or more of the following parts: a source listing, a cross-reference listing, and a compilation summary.

The parts of the list file are separated by page ejects. The contents of each part are described in the following sections.

The format of the output to the list file is controlled by the configuration file (see Section 4.1.4).

4.3.2. Source Listing

A source listing is an unmodified copy of the source text. The listing is divided into pages and each line is supplied with a line number.

The number of lines output in the source listing is governed by the following:

- parts of the listing can be suppressed by the use of LIST pragmas;
- a line containing a construct that caused a diagnostic message to be produced is printed even if it occurs at a point where listing has been suppressed by a LIST pragma.

An example of a source listing is shown in Chapter 10.

4.3.3. Compilation Summary

At the end of a compilation the compiler produces a summary that is output to the list file if the /list qualifier is active.

The summary contains information about:

- the type and name of the compilation unit, and whether it has been compiled successfully or not;
- the number of diagnostic messages produced, for each class of severity (see Section 4.3.5);
- which qualifiers were active;
- the VAX/VMS filename of the source file;
- the VAX/VMS filenames of the sublibraries constituting the current program library;
- the number of source text lines;
- elapsed real time and elapsed CPU time;
- a "Compilation terminated" message if the compilation unit was the last in the compilation, or "Compilation of next unit initiated" otherwise.

An example of a compilation summary is shown in Chapter 10.

4.3.4. Cross-Reference Listing

A cross-reference listing is an alphabetically sorted list of the identifiers, operators and character literals of a compilation unit. The list has an entry for each entity declared and/or used in the unit, with a few exceptions stated below. Overloading is evidenced by the occurrence of multiple entries for the same identifier.

For instantiations of generic units the visible declarations of the generic unit are included in the cross-reference listing as declared immediately after the instantiation. The visible declarations are the subprogram parameters for a generic subprogram and the declarations of the visible part of the package declaration for a generic package.

For type declarations all implicitly declared operations are included in the cross-reference listing.

Cross-reference information will be produced for every constituent character literal for string literals.

The following are not included in the cross-reference listing:

- pragma identifiers and pragma argument identifiers;
- numeric literals;
- record component identifiers and discriminant identifiers. For a selected name whose selector denotes a record component or a discriminant, only the prefix generates cross-reference information;
- a parent unit name following **separate**.

Each entry in the cross-reference listing contains:

- the identifier with at most 15 characters. If the identifier exceeds 15 characters, a bar ("|") is written in the 16th position and the remaining characters are not printed;
- the place of the definition, i.e., a line number if the entity is declared in the current compilation unit, otherwise the name of the compilation unit in which the entity is declared and the line number of the declaration;
- the numbers of the lines in which the entity is used. An asterisk ("**") after a line number indicates an assignment to a variable, initialization of a constant, or assignments to functions or user-defined operators by means of return statements.

An example of a cross-reference listing is shown in Chapter 10.

4.3.5. Diagnostic Messages

The Ada compiler issues diagnostic messages to the diagnostic file (see Section 4.1.3). Diagnostics other than warnings also appear on `sys$output`. If a source text listing is requested, diagnostics are also found embedded in the list file (see Section 4.1.2).

In a source listing a diagnostic message is placed immediately after the source line causing the message. Messages not related to a particular line are placed at the top of the listing. Every diagnostic message in the diagnostic file is followed by a line indicating the corresponding line number in the source text. The lines are ordered by increasing source line numbers. Line number 0 is assigned to messages not related to any particular line. In `sys$output` the messages appear in the order in which they are generated by the compiler.

The diagnostic messages are classified according to their severity and the compiler action taken:

- Warning:** Reports a questionable construct or an error that does not influence the meaning of the program. Warnings do not hinder the generation of object code. Example: A warning will be issued for constructs for which the compiler detects that `CONSTRAINT_ERROR` will be raised at runtime.
- Error:** Reports an illegal construct in the source program. Compilation continues, but no object code will be generated. Examples: most syntax errors; most static semantic errors.
- Severe Error:** Reports an error which causes the compilation to be terminated immediately. No object code is generated. Example: a library unit mentioned by a `with` clause is not present in the current program library.
- Fatal Error:** Reports an error in the Compiler System itself. The compilation is terminated immediately and no object code is produced. InterACT should be informed about all such errors (see Appendix X). The user may be able to circumvent a fatal error by correcting the program or by replacing program constructs with alternative constructs. Fatal errors are unlikely to affect program library integrity.

The detection of more than a certain number of errors during a compilation is considered a severe error. The limit is defined in the configuration file (see Section 4.1.4).

4.3.5.1. Format and Content of Diagnostic Messages

For certain syntactically incorrect constructs, the diagnostic message consists of a pointer line and a text line. In all other cases a diagnostic message consists of a text line only.

The pointer line contains a pointer (^) to the offending symbol or to an illegal character.

The text line contains the following information:

- the diagnostic message identification "****".
- the message code XY-Z where

X is the message number

Y is the severity code, a letter showing the severity of the error:

W: warning
E: error
S: severe error
F: fatal error

Z is an integer which together with the message number X uniquely identifies the compiler location that generated the diagnostic message. Z is only useful for compiler maintenance purposes

The message code (with the exception of the severity code) is suppressed if the configuration file component `SUPPRESS_ERRORNO` has the value `TRUE` (see Section 4.1.4).

- the message text. The text may include one context-dependent field which contains the name of the offending symbol; if longer than 16 characters, only the first 16 characters are shown.

Examples of diagnostic messages are:

- *** 18W-3: Warning: Exception `CONSTRAINT_ERROR` will be raised here
- *** 320E-2: Name `OBJ` does not denote a type
- *** 535E-0: Expression in return statement missing
- *** 1508S-0: Specification for this package body not present in the library

Chapter 10 shows an example program with errors and the source listing and diagnostic file produced.

4.3.6. Generated Assembly Listing

The generated assembly listing is the output of the 1750A Assembler when it assembles the generated IEEE assembly source produced by the compiler for a compilation unit. (The assembly takes place as part of the compile command.)

The Ada source text appears as comments in the generated assembly code, with the source text corresponding to each Ada scope start, declaration, statement, and scope end appearing before the corresponding generated assembly code. The line number from the Ada source file also appears in these comments. If an Ada source text line is longer than 72 characters, it is truncated with a backslash `()` character in the listing.

If the compilation unit contains generic instantiations or inline subprogram calls where the original Ada source text is in a different file from the unit being compiled, the source text is brought in from that file and a comment is generated to indicate when that file is being referenced. If an Ada source file cannot be located (because the user has moved or deleted it since the original compilation, or because it is for a predefined library unit), a comment is issued to that effect, and comments are interleaved that supply only the source line numbers.

The compiler unnests lexically nested subprogram bodies and task bodies in the generated code so that they appear textually after their parent scopes. The Ada source line comments for these bodies do not appear in their lexical place in the parent scopes, but rather with the unnested generated code. Occasionally special compiler-generated routines appear in the generated code that have no particular correspondence to the Ada source. A comment is issued to this effect when this happens.

In addition to the interleaved Ada source, comments at the beginning of the assembly listing indicate the source file name that this compilation unit came from, the compilation unit name, and the sublibrary file name that it is being compiled into.

The bottom of the generated assembly listing shows the object code sizes of the compilation unit.

Note that labels and external names in the assembly listing often refer to program unit numbers, rather than (or in addition to) unit names; if necessary, correspondence can be established through use of Ada PLU (see Chapter 3).

4.3.7. Return Status

After a compilation the VAX/VMS DCL symbols \$status and \$severity will reflect whether the compilation was successful. The possible values of \$severity and the low-order bits of \$status are 1 (success) or 2 (error).

4.4. The Program Library

This section briefly describes how the Ada compiler changes the program library. For a more general description of the program library, see Chapter 2.

The compiler is allowed to read from all sublibraries constituting the current program library, but only the current sublibrary may be changed.

4.4.1. Correct Compilation

In the following examples it is assumed that the compilation units are correctly compiled, i.e., that no errors are detected by the compiler.

Compilation of a library unit which is a declaration

If a declaration unit of the same name exists in the current sublibrary, it is deleted together with its body unit and possible subunits. A new declaration unit is inserted in the sublibrary, together with an empty body unit.

Compilation of a library unit which is a subprogram body

A subprogram body in a compilation unit is treated as a secondary unit, if the current sublibrary contains a subprogram declaration or a generic subprogram declaration of the same name and this declaration unit is not invalid.

In all other cases it will be treated as a library unit, i.e.:

- when there is no library unit of that name;
- when there is an invalid declaration unit of that name;
- when there is a package declaration, generic package declaration, or an instantiated package or subprogram of that name.

Compilation of a library unit which is an instantiation

A possible existing declaration unit of that name in the current sublibrary is deleted together with its body unit and possible subunits. A new declaration unit is inserted.

Compilation of a secondary unit which is a library unit body

The existing body is deleted from the sublibrary together with its possible subunits. A new body unit is inserted.

Compilation of a secondary unit which is a subunit

If the subunit exists in the sublibrary, it is deleted together with its possible subunits. A new subunit is inserted.

4.4.2. Incorrect Compilations

If the compiler detects an error in a compilation unit, the program library will be kept unchanged.

If a source file consists of several compilation units and an error is detected in any of these compilation units, the program library will *not* be updated for *any* of the compilation units.

4.5. Instantiation of Generic Units

4.5.1. Order of Compilation

When instantiating a generic unit, it is required that the entire unit including body and possible subunits is compiled before the first instantiation or – at the latest – in the same compilation. This is in accordance with [Ada RM 10.3].

4.5.2. Generic Formal Private Types

This section describes the treatment of a generic unit with a generic formal private type, where there is some construct in the generic unit that requires that the corresponding actual type must be constrained if it is an array type or a type with discriminants, and instantiations exist with such an unconstrained type [Ada RM 12.3.2(4)].

This is considered an illegal combination. In some cases the error is detected when the instantiation is compiled, in other cases when a *constraint-requiring construct* of the generic unit is compiled:

1. If the instantiation appears in a later compilation unit than the first *constraint-requiring construct* of the generic unit, the error is associated with the instantiation which is rejected by the compiler.
2. If the instantiation appears in the same compilation unit as the first *constraint-requiring construct* of the generic unit, there are two possibilities:
 - (a) If there is a *constraint-requiring construct* of the generic unit after the instantiation, an error message appears with the instantiation.
 - (b) If the instantiation appears after all *constraint-requiring constructs* of the generic unit in that compilation unit, an error message appears with the *constraint-requiring construct*, but refers to the illegal instantiation.
3. The instantiation appears in an earlier compilation unit than the first *constraint-requiring construct* of the generic unit, which in that case appears in the generic body or a subunit. If the instantiation has been accepted, the instantiation corresponds to the generic declaration only, and does not include the body. Nevertheless, if the generic unit and the instantiation are located in the same sublibrary, then the compiler considers it an error. An error message is issued with the *constraint-requiring construct* and refers to the illegal instantiation. The unit containing the instantiation is not changed, however, and is not marked as invalid.

LINKER OPTIONS

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

Chapter 5

The Ada Linker

Before a compiled Ada program can be executed it must be linked into a load module by the Ada Linker.

A single program may be linked for either a non-expanded memory 1750A configuration, or for any size of expanded memory 1750A configuration.

In its normal and conventional usage, the Ada Linker links a single Ada program.

The Ada Linker also has the capability to link multiple Ada programs into one load module, where the programs will execute concurrently. This capability, which is outside the definition of the Ada language, is called *multiprogramming*, and is further discussed below.

The Ada link, while one command, can be seen as having two parts: an "Ada part" and a "1750A part".

The Ada part performs the link-time functions that are required by the Ada language. This includes checking the consistency of the library units, and constructing an elaboration order for those library units. Any errors found in this process are reported.

To effect the elaboration order, the Ada link constructs an assembly language "elaboration caller routine" that is assembled and linked into the executable load module. This is a small routine that, during execution, gets control from the Ada runtime executive initiator. It invokes or otherwise marks the elaboration of each Ada library unit in the proper order, then returns control to the runtime executive, which in turn invokes the main program. The action of the elaboration caller routine is transparent to the user.

If no errors are found in the Ada part of the link, the 1750A part of the link takes place. This consists of assembling the elaboration caller routine, then invoking the InterACT 1750A Linker, linking the program unit object modules (stored in the program library) and the elaboration caller routine together with the necessary parts of the Ada runtime executive (and some other runtime modules needed by the generated code). The output of the full Ada link is an executable load module file.

The invocations of the 1750A Assembler and Linker are transparent to the user. However, qualifiers on the Ada link command allow the user to specify additional information to be used in the target link. Through this facility, a wide variety of runtime executive optional features, customizations, and user exit routines may be introduced to guide or alter the execution of the program. These are described in the *Ada 1750A Runtime Executive Programmer's Guide*. This facility may also be used to modify or add to the standard 1750A Linker control statements that are used in the 1750A part of the link; in this way, target memory may be precisely defined. The control statements involved are described in the *InterACT 1750A Assembler and Linker User's Manual*.

Expanded Memory

Expanded memory is an optional hardware feature of the 1750A. Without the expanded memory option, the 1750A interprets 16-bit addresses as physical addresses, thereby allowing a maximum memory space of 64K words. With the expanded memory option, the 1750A can address up to 1M words of memory by maintaining a set of page tables which enable it to translate 16-bit logical addresses into 20-bit physical addresses. (This 1M of memory can be divided into any ratio of code to data. Some implementations of 1750A map instruction and data fetches into separate physical hardware areas, resulting in up to 2M words of memory being available - 1M of code, 1M of data.)

However, since the 1750A is fundamentally a 16-bit processor, it does not allow direct access to the entire 1M address space. Instead, it defines up to 16 "contexts" known as *address states*, only one of which may be active at any moment. Each address state consists of a logical memory space containing up to 64K of code and 64K of data.

To a large degree, the difficulties involved in working with 1750A expanded memory stem from one question: *which code and data go in which address state?* This is because jumping from one address state to another, known as context switching, is an expensive operation for real-time applications if done indiscriminately. Consequently, the decision concerning what to place in each address state is best left to the system designer. Once that decision is made, the Compiler System automates the rest of the process.

These address state bindings are done at Ada link time. The user specifies a main program, which will reside in address state 0, and any number of "top-level" compilation units, which will reside in address states specified by the user. Calls to any subprograms defined within these top-level units (and the elaboration-time call to the unit itself) will be made via the Long Call facility.

The Long Call facility allows a subprogram residing in one address state to call a subprogram residing in another address state. The actual call and return is handled automatically by the Compiler System. (The implementation consists of the 1750A Linker replacing call and return instructions with branch-to-executive instructions, through which the Ada runtime executive performs a context switch using tables set up by the 1750A Linker.) Passed parameters, including those passed by reference (arrays and records), are also handled automatically. Thus, inter-address state calls look no different, in the Ada source, than intra-address state calls, and there are no restrictions on such calls.

The Ada runtime executive also automatically handles task rendezvous across address states; thus an entry call may also involve a context switch, if the user has designated the compilation unit containing the task to reside in a different address state from the calling task.

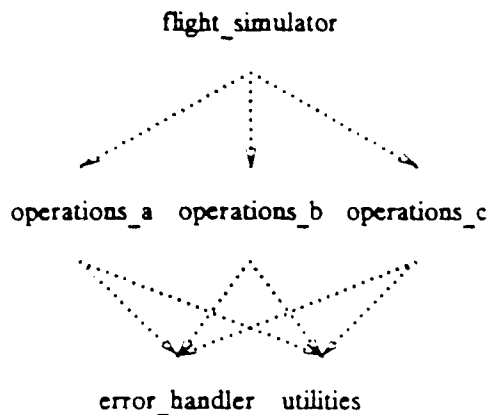
All units other than those specified by the user as "long called", automatically reside in every address state that references them. (The implementation consists of the 1750A Linker setting up page register tables that reflect this mapping, and the Ada runtime executive loading these page registers during initialization processing.) Thus, calls to these units, and references to their data, have no extra execution-time cost associated with them.

In addition to user-specified address states' contents, the Compiler System automatically includes in each defined address state (i.e. makes globally shareable) the Ada runtime system, including the system heap and the stacks for the main program and all tasks.

To summarize the memory capabilities of single program expanded memory support, a program linked for expanded memory may contain up to 1M of code. The program's local objects and access object space is limited to 64K. The program's library package objects (i.e., objects with static allocation) may occupy more than 64K, with no context switch overhead, to the extent that they are referenced in only some address states.

As an illustration, consider a simple application consisting of 6 library units: a main program `flight_simulator`,

and packages `operations_a`, `operations_b`, `operations_c`, `utilities`, and `error_handler`. The following diagram portrays the relationship between the units.



The arrows represent dependencies (i.e. "with" clauses). Thus, `flight_simulator` calls subprograms (or task entries) in the `operations` packages, all of which make use of `error_handler`. In addition, each of the `operations` makes use of a unit named `utilities`.

The system designer knows the application requires more than 64K of physical memory, thus the expanded memory option must be used. Units might be assigned to address states as follows:

address state 0	address state 1	address state 2
<code>flight_simulator</code>	<code>operations_a</code>	<code>operations_b</code>
<code>(operations_a)</code>	<code>error_handler</code>	<code>operations_c</code>
<code>(operations_b)</code>	<code>utilities</code>	<code>(error_handler)</code>
<code>(operations_c)</code>		<code>utilities</code>

The units in parentheses indicate units not actually resident in the listed address state but referenced via the long call facility. Thus, address state 0 contains only `flight_simulator` which makes long calls to the operations. Address state 1 contains `operations_a`, `utilities`, and `error_handler`. And address state 2 contains `operations_b` and `operations_c` which make long calls to `error_handler`, and ordinary calls to `utilities`.

Note that both `error_handler` and `utilities` are accessed in address states 1 and 2. But whereas `utilities` actually resides in both address states (i.e. one physical instance mapped into both address states), `error_handler` resides only in address state 1.

Multiprogramming

As stated above, multiprogramming is the capability of linking multiple Ada programs into one load module, where the programs will execute concurrently. As this concept is outside the definition of the Ada language, the discussion of multiprogramming here is specific to this Compiler System's implementation.

In multiprogramming, Ada units (comprising code, literals, and/or data) that are common to more than one program are linked but once, and are shared by those programs. With respect to code and literals, this has no effect upon execution, and results in more efficient memory utilization. However, with respect to data, this

means that the actions of one Ada program can affect, and possibly cause erroneous behavior in, another Ada program. Such an interaction may be desired, as in the case of a common library package's data being used to communicate between programs. If such an interaction is *not* desired, the program units that would otherwise be common may be rewritten as generic units, and instantiated with a different name for each program that uses them.

Elaboration of common units is only done once, by the "first" program that depends on them. This ordering is defined by the order in which the programs are named to the Ada link command (and not by their address state order, if an expanded memory link is being done).

In order to ensure that units are elaborated before being referenced, the runtime executive elaborates the units of each program serially, waiting for the elaborations for one program to finish before going on to the next program's elaborations. When all elaborations have completed, the main programs themselves are eligible to execute. Programs, and any tasks within them, are scheduled by their Ada priority on a global basis. See the *Ada 1750A Runtime Executive Programmer's Guide* for more details on this process, and on the criteria by which programs are scheduled and dispatched.

The main programs involved in a multiprogramming link must all be present within the same program library.

Multiprogramming may be done on either a non-expanded memory or an expanded memory 1750A configuration. In the latter case it is used in conjunction with the single program expanded memory linking features described above. One or more programs may be defined to an address state.

Note that it is not necessary to use multiprogramming to take advantage of a 1750A expanded memory configuration. Multiprogramming is often best suited towards real-time "operating systems" implemented in Ada, where each application running under the operating system is represented as an Ada main program, and where communication requirements among the programs are minor or absent.

5.1. The Invocation Command

The Ada Linker is invoked by submitting the following VAX/VMS command:

```
$ ada1750/link{qualifier} main-programs [long-called-units]

main-programs ::= main-program-name (single program link)
                  | {main-program-name{/as-qualifier | /options-qualifier}} (multiprogramming link)

long-called-units ::= {unit-name{/as-qualifier}}
```

As part of the "1750A part" of an Ada link, a temporary subdirectory is created below the current default directory. Use of this subdirectory, the name of which is constructed from the VAX/VMS process-id, permits concurrent linking in the same current default directory. The subdirectory contains work files only, and it and its contents are deleted at the end of the link.

A consequence of the use of this subdirectory is that an Ada link cannot be done from a current default directory that is eight directory levels deep, as that is the VAX/VMS limit for directory depth.

Infrequently, a control-C or control-Y interrupt of an Ada link will leave the subdirectory present. If this happens, the subdirectory and its contents must be deleted, in order that subsequent links (by that process, in that

current default directory) may take place.

5.1.1. Parameters and Qualifiers

Default values exist for all qualifiers as indicated below. All qualifier names may be abbreviated (characters omitted from the right) as long as no ambiguity arises.

main-program-name

If a single program link is being done, *main-program-name* must specify a main program which is a library unit of the current program library, but not necessarily of the current sublibrary. The library unit must be a parameterless procedure. Note that *main-program-name* is the identifier of an Ada procedure; it is not a VAX/VMS file specification.

When *main-program-name* is used as the file name in Ada link output (for the load module, memory map file, etc.), the file name will be truncated to 29 characters if necessary.

If a multiprogramming link is being done, multiple *main-program-names* are specified, separated by commas. The first name supplied is the one used for the file name in Ada link output.

The first three of the qualifiers below pertain to the "Ada part" of the Ada link. The remaining qualifiers pertain to the "1750A part" of the link.

/log [= *file-spec*]
/nolog (default)

The qualifier specifies whether a log file is to be produced during the linking. By default no log file is produced. If */log* is specified without a file specification, a log file named *main-program-name.log* is created in the current default directory. If a file specification is given, that file is created as the log file. The contents of the log file are described in Section 5.3.

/library = *file-spec*
/library = *ada1750_library* (default)

This qualifier specifies the current sublibrary and thereby also the current program library, which consists of the current sublibrary through the root sublibrary (see Chapter 2). If the qualifier is omitted, the sublibrary designated by the logical name *ada1750_library* is used as current sublibrary.

/mp

This qualifier specifies that a multiprogramming link be done. By default a single program link is done.

/as [= *address-state*]

This qualifier is used in two contexts. It must be used after the Ada link command verb to indicate that an expanded memory link be done (whether single program or multiprogramming). By default a non-expanded memory link is done. In this context it is used without an *address-state* value.

If a single program expanded memory link is being done, this qualifier is also used after each *long-called-unit*, to

specify the address state that unit will reside in.

If a multiprogramming link is being done, this qualifier may be used after each *main-program-name*, to specify the address state that program will reside in. However, if the qualifier is not used after any *main-program-names*, the programs are assigned by the Ada link to address states in their order of appearance, one per address state, starting with address state 0.

/options [=*macro-name*]

This qualifier is used to override certain default values that are used by the Ada runtime executive. If the qualifier is omitted, no overriding takes place.

The qualifier specifies the name of an assembly language macro containing one or more conditional assembly directives that override the default values of certain assembly-time symbols. (Note that this is a *macro* name, not a VAX/VMS file name.) If **/options** is specified without a *macro-name*, *main-program-name* is used as the macro name.

The names of these assembly-time symbols, their default values, and the runtime behavior that they control, are described in the *Ada 1750A Runtime Executive Programmer's Guide*. A macro file containing the definition of this macro must be available to the 1750A Assembler at the time of the link by one of the means documented in the *InterACT 1750A Assembler and Linker User's Manual*.

If a multiprogramming link is done, the **/options** qualifier may appear either after the Ada link command verb, in which case it applies to every program being linked (or, if no *macro-name* is given, each *main-program-name* default applies to each program being linked), or it may appear after some or all of the *main-program-names*, in which case it applies to only those programs (and supercedes for those programs a **/options** qualifier used after the command verb, if any).

/standard_control [=*file-spec*]
/standard_control = **adalink_standard_control** | **adalink_expmem_control** (default)

This qualifier specifies the file name of "standard" 1750A Linker control statements that are to be used for all links for an installation or project. If *file-spec* is omitted or only partially specified, []**adalink.lod** is used as a full or partial default. If the qualifier is omitted, the logical name **adalink_standard_control** or **adalink_expmem_control** (if an expanded memory link is being done) is assumed to define such a file, using the same partial default. If that logical name is not defined or the specified file does not exist, no standard control statements are used.

/control [=*file-spec*]

This qualifier specifies the file name of "user" 1750A Linker control statements that are to be used for this particular link. If *file-spec* is omitted or only partially specified, []*main-program-name.lod* is used as a full or partial default. If the qualifier is omitted or the specified file does not exist, no user control statements are used.

The files designated by the **/standard_control** and **/control** qualifiers are used to form the full input control statement stream to the 1750A Linker, in this concatenated order:

```
/standard_control file    (if it exists)
< statements generated by the Ada part of the link >
/control file            (if qualifier active and it exists)
```

The statements generated by the Ada part of the link are usually just SELECT or ADDRSTATE statements for the elaboration caller routine(s) and main program(s).

The Compiler System is delivered with `adalink_standard_control` and `adalink_expmem_control` defined to files that contain default sets of standard control statements. These consist of the minimal SECTION statements required by the 1750A Linker, and various other necessary directives.

```
/user_rts=search-list
/user_rts=adalink_user_rts      (default)
```

This qualifier specifies a VAX/VMS search list that contains either user-dependent RTE modules, such as a change to the task scheduler for a particular application, or pragma INTERFACE (ASSEMBLY) bodies for subprograms that are not library units (see Section F.2). Modules in this search list's directory(ies) are taken ahead of those in the directories specified by `/target_rts` (see below) and those in the standard RTE directory. If the qualifier is omitted, logical name `adalink_user_rts` is used, if the name has been defined.

```
/target_rts=search-list
/target_rts=adalink_target_rts  (default)
```

This qualifier specifies a VAX/VMS search list that contains 1750A-implementation(target)-dependent runtime executive (RTE) modules, such as modules to do character I/O for a particular simulator or microprocessor. Modules in this search list's directory(ies) are taken ahead of those in the standard RTE directory. If the qualifier is omitted, logical name `adalink_target_rts` is used, if the name has been defined. Note however that if pragma `NO_DYNAMIC_OBJECTS_OR_VALUES_USED` is specified (see Section F.3), this qualifier has no effect.

```
/debug
/nodebug      (default)
```

When this qualifier is given, the Ada Linker will produce a symbolic debug information file, containing symbolic debug information for all program units involved in the link that were compiled with the `/debug` compiler qualifier active. By default no such file is produced, even if some of the program units linked were compiled with `/debug` active.

This symbolic debug information file is used by the InterACT Symbolic Debugging System.

The `show/containers` command of Ada PLU may be used to determine which units in the program library have debug information containers, i.e., which units were compiled with `/debug` active.

It is important to note that the identical executable load module is produced by the Ada Linker, whether or not the `/debug` qualifier is active.

```
/actlink_qualifiers="1750A Linker qualifiers"
```

This qualifier specifies a string containing one or more command qualifiers to be passed to the execution of the 1750A Linker.

```
/stop[=number]
```

This qualifier, when used with no *number*, results in the Ada link stopping after the "Ada part" has done all Ada-required checking, and has created a VAX/VMS DCL command file (located in the temporary

subdirectory) that executes the "1750A part", but before that command file has actually been invoked.

When used with *number* = 1, the command file is invoked, but stops before the 1750A Linker is invoked, leaving the temporary subdirectory and its files in place. When used with *number* = 2, it executes the 1750A Linker but then stops before the symbolic debug information file is produced.

This qualifier is useful for trouble-shooting, or for giving the user an intervention point for Ada link customizations not covered by any of the available options.

5.1.2. Examples

A single program, 64K memory link:

```
$ ada1750/link flight_simulator
```

A single program link for 128K expanded memory:

```
$ ada1750/link/as flight_simulator
```

In the above case, no long called units are necessary since only one address state is being used. Now an example of a single program, greater than 128K expanded memory link, where long calls are necessary, for the illustration presented at the beginning of this Chapter:

```
$ ada1750/link/as flight_simulator operations_a/as=1,error_handler/as=1,-
operations_b/as=2,operations_c/as=2
```

Some multiprogramming examples, with 64K and then expanded memory:

```
$ ada1750/link/mp able,baker,charlie
```

```
$ ada1750/link/mp/as able,baker,charlie
```

```
$ ada1750/link/mp/as able/as=0,baker/as=1,charlie/as=2
```

The last two examples above are equivalent. However, the following sort of assignment can only be done using the second form:

```
$ ada1750/link/mp/as able/as=0,baker/as=1,charlie/as=1,dog/as=4
```

Now, an example of overriding default runtime executive values, in this case the system heap size and main stack size:

```
$ ada1750/link/opt flight_simulator
```

where `flight_simulator.mac` in the current directory contains

```

FLIGHT_SIMULATOR MACRO
  INEAP_SIZE      ASET 24*1024
  IMAINSTACK_SIZE ASET 8*1024
ENDMAC

```

Some examples of overriding values when multiprogramming is involved:

```
$ ada1750/link/mp/as/opt=large_stack able,baker,charlie
```

would use `large_stack.mac` for all three programs, while

```
$ ada1750/link/mp/as/opt able,baker,charlie
```

would use `able.mac`, `baker.mac`, and `charlie.mac` for the three programs respectively. Alternatively,

```
$ ada1750/link/mp/as/opt=large_stack able,baker/opt=small_stack,charlie
```

would use `large_stack.mac` for ABLE and CHARLIE, but use `small_stack.mac` for BAKER, while

```
$ ada1750/link/mp/as able,baker/opt,charlie
```

would use `baker.mac` for BAKER, and all default values for ABLE and CHARLIE.

Now, an example of introducing "user" 1750A Linker control statements:

```
$ ada1750/link/control test_driver
```

where `test_driver.lod` in the current directory contains

```

PAGESIZE 60
SELECT [dma.object]DMACHECK
NOLOAD

```

Note that the `SELECT` statement specifies the directory where the object module `dmacheck.rlc` is located. Now, an example of the use of user and target RTE directories:

```

$ define adalink_target_rts [tektronics.lo.test],[tektronics.lo]
$ ada1750/link/user_rts=sys$user:[test_stor_mgr] flight_simulator

```

Runtime executive modules will be looked for in the directory specified by the `/user_rts` qualifier, then in the two directories specified by the `adalink_target_rts` logical name, and lastly in the standard RTE directory.

To revert to referencing only the standard RTE directory:

```

$ deassign adalink_target_rts
$ ada1750/link flight_simulator

```

5.2. Load Module Output

If an Ada linking is successfully completed, the 1750A Linker produces an executable load module file named *main-program-name.abs* in the current default directory.

The load module is in InterACT load module format, which may require further reformatting before being loaded into 1750A hardware or simulators (see Chapter 8).

5.2.1. Symbolic Debug Information Output

If an Ada linking with the `/debug` qualifier active is successfully completed, a symbolic debug information file named *main-program-name.d* is created in the current default directory. This file is used by the InterACT Symbolic Debugging System.

5.3. Linker Text Output

The Ada Linker produces the following text output:

1. Diagnostic messages other than warnings are written to `sys$output`, and all messages are written to the log file if `/log` is active.
2. An elaboration order list is written to the log file if `/log` is active.
3. A required recompilations list is written to `sys$output` if not empty, and to the log file if `/log` is active.
4. A linking summary is written to the log file if `/log` is active.
5. A 1750A Linker memory map file, *main-program-name.map*. (See the *InterACT 1750A Assembler and Linker User's Manual* for contents.)
6. An assembly listing of the generated module that elaborates all library units, *e\$main-program-name.als*. If a multiprogramming link is done, separate listings are produced for each program.
7. If a multiprogramming link is done, an assembly listing of a generated module that communicates program information to the Ada runtime executive, *\$mpt.als*.

Note that the log file contains information relevant to the "Ada part" of the link, while the memory map file contains information relevant to the "1750A part" of the link.

3.1. Diagnostic Messages

The Ada Linker may issue two kinds of diagnostic messages, warnings and severe errors.

A warning reports something which does not prevent a successful linking, but which might be an error. A warning is issued if the body unit is invalid or is lacking an object code container for a program unit which formally does not need a body. The linking summary on the log file contains the total number of warnings issued.

A severe error message reports an error which prevents a successful linking. Any inconsistency detected by the linker will cause a severe error message, e.g., if some required unit does not exist in the library or if some time stamps do not agree.

Examples of diagnostic messages from the Ada Linker can be found in Chapter 10.

5.3.2. Elaboration Order List

The elaboration order list contains an entry for each unit included, and shows the order in which the units will be elaborated. For each unit the unit type, the compilation time, and the dependencies are shown. Furthermore, any elaboration inconsistencies are reported.

When a multiprogramming link is done, the elaboration order lists will contain the full elaboration order of each program, without regard to multiprogramming. These orders can be compared to the elaboration caller assembly listing for a program, to see which elaborations were omitted due to multiprogramming.

5.3.3. Required Recompilations List

The required recompilations list reflects any inconsistencies detected in the library, that prevented the link from taking place.

The entries in the list contain the unit name, and an indication of the unit being a declaration unit, a body unit, or a subunit. The list is in a recommended recompilation order, consistent with the dependencies among the units.

If the number of recompilations is small, they can usually be performed by hand using this list. Otherwise, the Ada Recompile (see Chapter 6) may be used to accomplish the recompilation in a fully automatic way.

Examples of required recompilation lists can be found in Chapter 10.

5.3.4. Return Status

After an Ada link the VAX/VMS DCL symbols `$status` and `$severity` will reflect whether the link was successful. The possible values of `$severity` and the low-order bits of `$status` are any of the values defined by DCL.

5.3.5. Linking Summary

The linking summary contains the following information:

- parameters and active qualifiers;
- the VAX/VMS file names of the sublibraries constituting the current program library;
- the number of each type of diagnostic messages;
- a termination message, telling whether a linking has terminated successfully or unsuccessfully.

5.4. Commands for Defining the Target Environment

There are a number of different target environments that Ada programs can run in, due to different implementations of the MIL-STD-1750A architecture.

Each of these environments may require some changes to either the standard linker control statements, or the runtime executive modules, that are used in an Ada link. These changes may be effected by various Ada link qualifiers and their logical name defaults, as described in Section 5.1.1. However, convenience commands, of the form `use-` (for example, `useact` for the InterACT 1750A Instruction Set Architecture Simulator), exist to define the appropriate Ada link logical names. These commands are invoked before an Ada link, and remain in effect for subsequent Ada links until changed by another such command.

These commands are described in full detail in the *Ada 1750A Runtime Executive Programmer's Guide*.

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:

package STANDARD is

type INTEGER is range -32_768 .. 32_767;

type LONG_INTEGER is range -2_147_483_648..2_147_483_647;

type FLOAT is digits 6
range -1.0*2.0**127..0.999999*2.0**127;

type LONG_FLOAT is digits 9
range -1.0*2.0**127 .. 0.99999999*2.0**127;

type DURATION is delta 1.0E-04
range -214_748.3648..214_748.3647;

end STANDARD;

Appendix F

Appendix F of the Ada Reference Manual

This appendix describes all implementation-dependent characteristics of the Ada language as implemented by the InterACT Ada 1750A Compiler, including those required in the Appendix F frame of *Ada RM*.

F.1. Predefined Types in Package STANDARD

This section describes the implementation-dependent predefined types declared in the predefined package STANDARD [*Ada RM Annex C*], and the relevant attributes of these types.

F.1.1. Integer Types

Two predefined integer types are implemented, INTEGER and LONG_INTEGER. They have the following attributes:

INTEGER'FIRST	=	-32_768
INTEGER'LAST	=	32_767
INTEGER'SIZE	=	16
LONG_INTEGER'FIRST	=	-2_147_483_648
LONG_INTEGER'LAST	=	2_147_483_647
LONG_INTEGER'SIZE	=	32

F.1.2. Floating Point Types

Two predefined floating point types are implemented, FLOAT and LONG_FLOAT. They have the following attributes:

FLOAT'DIGITS	=	6
FLOAT'EPSILON	=	9.53674316406250E-07
FLOAT'FIRST	=	-1.0 * 2.0**127
FLOAT'LARGE	=	1.93428038904620E + 25
FLOAT'LAST	=	0.999999 * 2.0**127
FLOAT'MACHINE_EMAX	=	127

Float'MACHINE_EMIN	=	-128
Float'MACHINE_MANTISSA	=	23
Float'MACHINE_OVERFLOWS	=	TRUE
Float'MACHINE_RADIX	=	2
Float'MACHINE_ROUNDS	=	FALSE
Float'MANTISSA	=	21
Float'SAFE_EMAX	=	127
Float'SAFE_LARGE	=	Float'LAST
Float'SAFE_SMALL	=	$0.5 * 2.0^{**}(-127)$
Float'SIZE	=	32
Long_Float'DIGITS	=	9
Long_Float'EPSILON	=	$9.31322574615479E-10$
Long_Float'FIRST	=	$-1.0 * 2.0^{**}127$
Long_Float'LARGE	=	$2.0^{**}124 * (1.0 - 2.0^{**}(-31))$
Long_Float'LAST	=	$.99999999 * 2.0^{**}127$
Long_Float'MACHINE_EMAX	=	127
Long_Float'MACHINE_EMIN	=	-128
Long_Float'MACHINE_MANTISSA	=	39
Long_Float'MACHINE_OVERFLOWS	=	TRUE
Long_Float'MACHINE_RADIX	=	2
Long_Float'MACHINE_ROUNDS	=	FALSE
Long_Float'MANTISSA	=	31
Long_Float'SAFE_EMAX	=	127
Long_Float'SAFE_LARGE	=	Long_Float'LAST
Long_Float'SAFE_SMALL	=	$0.5 * 2^{**}(-127)$
Long_Float'SIZE	=	48

F.13. Fixed Point Types

Two kinds of anonymous predefined fixed point types are implemented, *fixed* and *long fixed* (which are not defined in package STANDARD, but are used here only for reference), as well as the predefined type DURATION.

For objects of *fixed* types, 16 bits are used for the representation of the object. For objects of *long fixed* types, 32 bits are used for the representation of the object.

For *fixed* and *long fixed* there is a virtual predefined type for each possible value of *small* [Ada RM 3.5.9]. The possible values of *small* are the powers of two that are representable by a LONG_FLOAT value, unless a length clause specifying TSMALL is given, in which case the specified value is used.

The lower and upper bounds of these types are:

lower bound of <i>fixed</i> types	=	$-32768 * small$
upper bound of <i>fixed</i> types	=	$32767 * small$
lower bound of <i>long fixed</i> types	=	$-2_147_483_648 * small$
upper bound of <i>long fixed</i> types	=	$2_147_483_647 * small$

A declared fixed point type is represented as that predefined *fixed* or *long fixed* type which has the largest value of *small* not greater than the declared delta, and which has the smallest range that includes the declared range

constraint.

Any fixed point type T has the following attributes:

T'MACHINE_OVERFLOWS = TRUE
T'MACHINE_ROUNDS = FALSE

The predefined fixed point type DURATION has the following attributes:

DURATION'AFT = 4
DURATION'DELTA = 1.0E-04
DURATION'FIRST = -214_748.3648
DURATION'FORE = 7
DURATION'LARGE = DURATION'LAST
DURATION'LAST = 214_748.3647
DURATION'MANTISSA = 31
DURATION'SAFE_LARGE = DURATION'LARGE
DURATION'SAFE_SMALL = DURATION'SMALL
DURATION'SIZE = 32
DURATION'SMALL = 1.0E-04

F.2. Predefined Language Pragmas

This section lists all language-defined pragmas and any restrictions on their use and effect as compared to the definitions given in *Ada RM*.

F.2.1. Pragma CONTROLLED

This pragma has no effect, as no automatic storage reclamation is performed before the point allowed by the pragma.

F.2.2. Pragma ELABORATE

As in *Ada RM*.

F.2.3. Pragma INLINE

This pragma causes inline expansion to be performed, except in the following cases:

1. The whole body of the subprogram for which inline expansion is wanted has not been seen. This ensures that recursive procedures cannot be inline expanded.
2. The subprogram call appears in an expression on which conformance checks may be applied, i.e., in a subprogram specification, in a discriminant part, or in a formal part of an entry declaration or accept statement.

3. The subprogram is an instantiation of the predefined generic subprograms `UNCHECKED_CONVERSION` or `UNCHECKED_DEALLOCATION`. Calls to such subprograms are expanded inline by the compiler automatically.
4. The subprogram is declared in a generic unit. The body of that generic unit is compiled as a secondary unit in the same compilation as a unit containing a call to (an instance of) the subprogram.
5. The subprogram is declared by a renaming declaration.
6. The subprogram is passed as a generic actual parameter.

A warning is given if inline expansion is not achieved.

F.2.4. Pragma INTERFACE

This pragma is supported for the language names defined by the enumerated type `INTERFACE_LANGUAGE` in package `SYSTEM`. Languages other than BIF support Ada calls to subprograms whose bodies are written in that language. Language BIF (for "built-in function") supports inline insertion of assembly language macro invocations; the macros themselves may consist of executions of 1750A hardware built-in functions, or of any sequence of 1750A instructions. Thus, pragma `INTERFACE` (BIF) serves as an alternative to machine code insertions.

Language ASSEMBLY

For pragma `INTERFACE` (ASSEMBLY), the compiler generates a call to the name of the subprogram. The subprogram name must not exceed 31 characters in length. Parameters and results, if any, are passed in the same fashion as for a normal Ada call (see Appendix P).

Assembly subprogram bodies are not elaborated at runtime, and no runtime elaboration check is made when such subprograms are called.

Assembly subprogram bodies may in turn call Ada program units, but must obey all Ada calling and environmental conventions in doing so. Furthermore, Ada dependencies (in the form of context clauses) on the called program units must exist. That is, merely calling Ada program units from an assembly subprogram body will not make those program units visible to the Ada Linker.

A pragma `INTERFACE` (ASSEMBLY) subprogram may be used as a main program. In this case, the procedure specification for the main program must contain context clauses that will (transitively) name all Ada program units.

If an Ada subprogram declared with pragma `INTERFACE` (ASSEMBLY) is a library unit, the assembled subprogram body object code module must be put into the program library via the Ada Library Injection Tool (see Chapter 7). The Ada Linker will then automatically include the object code of the body in a link, as it would the object code of a normal Ada body.

If the Ada subprogram is not a library unit, the assembled subprogram body object code module cannot be put into the program library. In this case, the user must direct the Ada Linker to the directory containing the object code module (via the `/user_rts` qualifier, see Section 5.1), so that the 1750A Linker can find it.

Language BIF

For pragma `INTERFACE` (`BIF`), the compiler generates an inline macro invocation that is the name of the subprogram. The subprogram name must not exceed 31 characters in length. Subprogram parameters and results, if any, are passed in the same fashion as for a normal Ada call (see Appendix P), except that the macro invocation replaces the call. However, subprogram parameters may be passed in registers if pragma `INTERFACE_PARAMETERS` is used (see Section F.3.7). Use of this pragma, as well as pragma `INTERFACE_SCRATCH` and, if desired, pragma `INTERFACE_RESULT` (again, see Section F.3.7) is recommended for most efficient usage of pragma `INTERFACE` (`BIF`). No macro arguments are passed on the invocation.

A macro file must exist at the time of the compile containing a macro definition with the same name as the subprogram. This macro file must be available by one of the means documented in the *InterACT 1750A Assembler and Linker User's Manual*.

Languages JOVLAL and FORTRAN

These languages may also be specified for pragma `INTERFACE`, but are equivalent to language `ASSEMBLY`. The compiler generates calls to such subprograms as if they were Ada subprograms, and does not do any special data mapping or parameter passing peculiar to the `InterACT JOVLAL` or `FORTRAN` compilers.

F.2.5. Pragma LIST

As in *Ada RM*.

F.2.6. Pragma MEMORY_SIZE

This pragma has no effect. See pragma `SYSTEM_NAME`.

F.2.7. Pragma OPTIMIZE

This pragma has no effect.

F.2.8. Pragma PACK

This pragma is accepted for array types whose component type is an integer, enumeration, or fixed point type that may be represented in 16 bits or less. (The pragma is accepted but has no effect for other array types.)

The pragma normally has the effect that in allocating storage for an object of the array type, the components of the object are each packed into the next largest 2^n bits needed to contain a value of the component type. This calculation is done using the *minimal size* for the component type (see Section F.6.1 for the definition of the minimal size of a type).

However, if the array's component type is declared with a size specification length clause, then the components of the object are each packed into exactly the number of bits specified by the length clause. This means that if the specified size is not a power of two, and if the array takes up more than a word of memory, then some components will be allocated across word boundaries. This achieves the maximum storage compaction but makes for less efficient array indexing and other array operations.

Some examples:

```

type BOOL_ARR is array (1..32) of BOOLEAN;  -- BOOLEAN minimal size is 1 bit
pragma PACK (BOOL_ARR);                    -- each component packed into 1 bit

type TINY_INT is range -2..1;               -- minimal size is 2 bits
type TINY_ARR is array (1..32) of TINY_INT;
pragma PACK (TINY_ARR);                    -- each component packed into 2 bits

type SMALL_INT is range 0..63;              -- minimal size is 6 bits, not a power of two
type SMALL_ARR is array (1..32) of SMALL_INT;
pragma PACK (SMALL_ARR);                  -- thus, each component packed into 8 bits

type SMALL_INT_2 is range 0..63;            -- minimal size is 6 bits, but
for SMALL_INT_2'SIZE use 6;                -- this time length clause is used
type SMALL_ARR_2 is array (1..32) of SMALL_INT_2;
pragma PACK (SMALL_ARR_2);                -- thus, each component packed into 6 bits;
                                           -- some components cross word boundaries

```

Pragma PACK is also accepted for record types but has no effect. Record representation clauses may be used to "pack" components of a record into any desired number of bits; see Section F.6.3.

F.2.9. Pragma PAGE

As in *Ada RM*.

F.2.10. Pragma PRIORITY

As in *Ada RM*. See the *Ada 1750A Runtime Executive Programmer's Guide* for how a default priority may be set.

F.2.11. Pragma SHARED

This pragma has no effect, in terms of the compiler (and a warning message is issued). However, based on the current method of code generation, the effect of pragma SHARED is automatically achieved for all scalar and access objects.

F.2.12. Pragma STORAGE_UNIT

This pragma has no effect. See pragma SYSTEM_NAME.

F.2.13. Pragma SUPPRESS

Only the "identifier" argument, which identifies the type of check to be omitted, is allowed. The "[ON =>] name" argument, which isolates the check omission to a specific object, type, or subprogram, is not supported.

Pragma SUPPRESS with all checks other than DIVISION_CHECK and OVERFLOW_CHECK results in the corresponding checking code not being generated. The implementation of arithmetic operations is such that, in general, pragma SUPPRESS with DIVISION_CHECK and OVERFLOW_CHECK has no effect. In this case, runtime executive customizations may be used to mask the overflow interrupts that are used to implement these checks (see the *Ada 1750A Runtime Executive Programmer's Guide* for details). However, in certain cases

involving multiplication by constants or numeric type conversions, pragma `SUPPRESS` with `DIVISION_CHECK` or `OVERFLOW_CHECK` results in code being generated such that the overflow interrupt cannot occur.

F.2.14. Pragma `SYSTEM_NAME`

This pragma has no effect. The only possible `SYSTEM_NAME` is `MIL_STD_1750A`. The compilation of pragma `MEMORY_SIZE`, pragma `STORAGE_UNIT`, or this pragma does not cause an implicit recompilation of package `SYSTEM`.

F.3. Implementation-dependent Pragmas

F.3.1. Program Library Basis Pragmas

Certain pragmas defined by this Compiler System apply to Ada programs as a whole, rather than to individual compilation units or declarative regions. These pragmas are

- `NO_DYNAMIC_OBJECTS_OR_VALUES_USED`
- `NO_DYNAMIC_MULTIDIMENSIONAL_ARRAYS_USED`
- `SET_MACHINE_OVERFLOW_FALSE_FOR_ANONYMOUS_FIXED`

These pragmas apply on a program library wide basis, and thus apply to any and all programs compiled and linked from a given program library. The meanings of these pragmas is described in the subsections below; the way in which these pragmas are specified is described in this subsection.

These pragmas may only be specified within the implementation-defined library unit `LIBRARY_PRAGMAS`, which in turn may only be compiled into a root (predefined) sublibrary. If either of these restrictions are not honored, the pragmas have no effect.

The contents of this library unit when delivered are

```
package LIBRARY_PRAGMAS is
  NO_DYNAMIC_OBJECTS_OR_VALUES_USED : constant BOOLEAN := FALSE;
  NO_DYNAMIC_MULTIDIMENSIONAL_ARRAYS_USED : constant BOOLEAN := FALSE;
  SET_MACHINE_OVERFLOW_FALSE_FOR_ANONYMOUS_FIXED : constant BOOLEAN := FALSE;
end LIBRARY_PRAGMAS;
```

In order to specify any or all of the pragmas, the source for this package is modified to include the pragmas after the constant declarations (the source file is defined by the logical name `actada_library_pragmas`). For example,

```

package LIBRARY_PRAGMAS is
    NO_DYNAMIC_OBJECTS_OR_VALUES_USED : constant BOOLEAN := FALSE;
    NO_DYNAMIC_MULTIDIMENSIONAL_ARRAYS_USED : constant BOOLEAN := FALSE;
    SET_MACHINE_OVERFLOW_FALSE_FOR_ANONYMOUS_FIXED : constant BOOLEAN := FALSE;
    pragma NO_DYNAMIC_OBJECTS_OR_VALUES_USED;
    pragma SET_MACHINE_OVERFLOW_FALSE_FOR_ANONYMOUS_FIXED;
end LIBRARY_PRAGMAS;

```

This modified source is then compiled into the predefined library.

In addition to the effects described in the subsections below, the pragmas have the effect of changing the initialization value to TRUE for the corresponding constant objects.

If unit LIBRARY_PRAGMAS is modified and compiled by the user, *it must be compiled before any other user compilation unit*. If it is not, the program will be erroneous.

Note that while these pragmas apply to an entire program library, it is possible to create more than one program library (via the Ada PLU command `create/root`; see Chapter 3), with each library having these pragmas specified or not according to user desire.

An example sequence for specifying the pragmas for the delivered program library:

```

$ set def sys$user:[libraries]
$ copy actada_library_pragmas []library_pragmas_s.ada
$ eve library_pragmas_s.ada
<add desired pragmas, as described above>
$ ada1750/lib=predefined_library library_pragmas_s
$ ada1750/plu ! create user libraries under predefined
create application.alb predefined_library
exit
$ define ada1750_library application.alb

```

An example sequence for specifying the pragmas for a new program library, leaving the delivered program library intact:

```

$ set def sys$user:[libraries]
$ ada1750/plu ! create new predefined library
create/root pragmas_root.alb
exit
$ copy actada_library_pragmas []library_pragmas_s.ada
$ eve library_pragmas_s.ada
<add desired pragmas, as described above>
$ ada1750/lib=pragmas_root.alb library_pragmas_s
$ ada1750/plu ! create user libraries under new predefined
create application.alb pragmas_root.alb
exit
$ define ada1750_library application.alb

```

F.3.2. Pragma NO_DYNAMIC_OBJECTS_OR_VALUES_USED

This pragma works on a program library basis. See the subsection at the beginning of this section for how such pragmas are used.

Use of this pragma informs the compiler that all created objects and all computed values have statically known sizes. The language usages that do *not* meet this assertion are

- TIMAGE for integer types
- arrays objects or values of (sub)types with non-static index constraints, or with component subtypes with non-static index constraints
- array aggregates of an unconstrained type
- catenations (even with statically sized operands)
- collections with non-static sizes

Programs that violate the assertion of this pragma are erroneous.

The effect of this pragma is to use a different, and more efficient, set of compiler protocols for runtime stack organization and register usage. These variant protocols are described in Appendix P.

F.3.3. Pragma NO_DYNAMIC_MULTIDIMENSIONAL_ARRAYS_USED

This pragma works on a program library basis. See the subsection at the beginning of this section for how such pragmas are used.

Use of this pragma informs the compiler that all declarations of multidimensional array types or objects have static index constraints [*Ada RM 4.9 (11)*], and that the component subtypes of such arrays, if arrays themselves, also have static index constraints. That is, all multidimensional arrays have statically known size. Programs that violate the assertion of this pragma are erroneous.

The effect of this pragma is to use a special technique, known as *bias vectors*, in the generated code for the calculation of array indexed component offsets for multi-dimensional arrays. This technique involves building a data structure that contains some precomputed offsets, and then indexing into that structure. The major advantage of this technique is that few or no multiplication operations need be generated. The major drawback is that additional literal area space is required, although this can be minimized if the first dimension of the array is the shortest.

The bias vector data structures are allocated as part of elaboration of the constrained array subtype declaration (or object declaration that implicitly declares such a subtype).

Bias vectors are not used if the array index base type is LONG_INTEGER or if pragma PACK applies to the array.

F.3.4. Pragma `ESTABLISH_OPTIMIZED_REFERENCE` and `ASSUME_OPTIMIZED_REFERENCE`

These pragmas are used to direct the compiler to generate code that more efficiently references objects in a package. This efficiency is achieved by using a base register to address the package objects.

Pragma `ESTABLISH_OPTIMIZED_REFERENCE` instructs the compiler to load a base register with the beginning address of the objects in the designated package, and to access such objects using the base register. The pragma has the form

```
pragma ESTABLISH_OPTIMIZED_REFERENCE (package_name);
```

The pragma may appear anywhere within a program unit; the load and subsequent usage of the base register will begin at the point of the pragma appearance. The pragma applies only to the program unit it appears in; it does not apply to program units nested within that unit.

Pragma `ASSUME_OPTIMIZED_REFERENCE` instructs the compiler to assume that the designated package's beginning address has been loaded into a base register, and to access such objects using the base register. The pragma has the form

```
pragma ASSUME_OPTIMIZED_REFERENCE (package_name);
```

The pragma should appear at the beginning of the declarative part of a program unit. The pragma applies only to the program unit it appears in; it does not apply to program units nested within that unit. It is not necessary to use this pragma after an instance of pragma `ESTABLISH_OPTIMIZED_REFERENCE`; rather, it must be used in program units that are called from the unit that contains the pragma `ESTABLISH_OPTIMIZED_REFERENCE`. If there are intervening (in terms of calls) units between the unit containing pragma `ESTABLISH_OPTIMIZED_REFERENCE` and the unit desiring to use pragma `ASSUME_OPTIMIZED_REFERENCE`, then those intervening units must also use pragma `ASSUME_OPTIMIZED_REFERENCE`.

The pragmas apply only to packages that are library units. Only the objects in the specification part of the package, and within base register range of the package beginning, are accessed by base register.

Only one base register is used by these pragmas, that being register 12. Thus, the pragmas can be in effect for only one package at any given time during execution.

An example of the use of these pragmas:

```
package GLOBAL_VARS is
  ...
end GLOBAL_VARS;

with GLOBAL_VARS; use GLOBAL_VARS;
procedure P is
  pragma ESTABLISH_OPTIMIZED_REFERENCE (GLOBAL_VARS);
  ...
  procedure INNER is
    pragma ASSUME_OPTIMIZED_REFERENCE (GLOBAL_VARS);
  begin
    ...
  end INNER;
begin
  ...
  INNER;
```

```
...
end P;
```

F.3.5. Pragma EXPORT

This pragma is used to define an external name for Ada objects, so that they may be accessed from non-Ada routines. The pragma has the form

```
pragma EXPORT (object_name [,external_name_string_literal]);
```

The pragma must appear immediately after the associated object declaration. If the second argument is omitted, the object name is used as the external name. If the resulting external name is longer than 31 characters, it will be so truncated.

The associated object must be declared in a library package (or package nested within a library package), and must not be a statically-valued scalar constant (as such constants are not allocated in memory).

Identical external names should not be put out by multiple uses of the pragma (names can always be made unique by use of the second argument).

As an example of the use of this pragma, the objects in the following Ada library package

```
package GLOBAL is
  ABLE : FLOAT;
  pragma EXPORT (ABLE);

  BAKER : STRING(1..8);
  pragma EXPORT (BAKER, "global.baker");
end GLOBAL;
```

may be accessed in the following assembly language routine

```
MODULE   LOW_LEVEL
CSECT    CODE
...
EXTREF   ABLE
LDL      ABLE,R0          ; get value of ABLE
EXTREF   GLOBAL.BAKER
LD       #GLOBAL.BAKER,R2 ; get address of BAKER
...
END
```

F.3.6. Pragma IMPORT

This pragma is used to associate an Ada object with an object defined and allocated externally to the Ada program.

```
pragma IMPORT (object_name [,external_name_string_literal]);
```

The pragma must appear immediately after the associated object declaration. If the second argument is

omitted, the object name is used as the external name. If the resulting external name is longer than 31 characters, it will be so truncated.

The associated object must be declared in a library package (or package nested within a library package). The associated object may not have an explicit or implicit initialization.

As an example of the use of this pragma, the objects in the following Ada library package

```
package GLOBAL is
  ABLE : FLOAT;
  pragma IMPORT (ABLE);

  BAKER : STRING(1..8);
  pragma IMPORT (BAKER, "global.baker");

end GLOBAL;
```

are actually defined and allocated in the following assembly language module

```
MODULE   GLOBAL_VALUES
CSECT    DATA
...
EXTDEF   ABLE
ABLE     RES      2
EXTDEF   GLOBAL.BAKER
GLOBAL.BAKER DATAC 'abcdefgh'
...
END
```

F3.7. Pragasms `INTERFACE_PARAMETERS`, `INTERFACE_RESULT` and `INTERFACE_SCRATCH`

These pragmas are used in conjunction with pragma `INTERFACE` (BIF) to name the specific 1750A machine registers to be used during BIF processing.

The type `PRAGMA_INTERFACE_PARAMETER_LOCATIONS` in package `SYSTEM` defines names for the 1750A machine registers that must be used in association with these pragmas.

Registers 10, 11, and 15 should not be used with these pragmas as they serve special purposes in the compiler (see Appendix P for details). If they are used, it is the user's responsibility to save and/or restore the registers inside the BIF macro.

Sample usage of these pragmas:

```
function BIT_OPERATION (X, Y : INTEGER) return INTEGER;
pragma INTERFACE (BIF, BIT_OPERATION);
pragma INTERFACE_PARAMETERS (BIT_OPERATION, X => R4, Y => R5);
pragma INTERFACE_RESULT (BIT_OPERATION, R9);
pragma INTERFACE_SCRATCH (BIT_OPERATION, R6, R3);
```

Pragma `INTERFACE_PARAMETERS` specifies the 1750A machine registers that should be used to pass the actual parameters of the subprogram. If this pragma is not specified, the subprogram parameters will be passed according to standard compiler protocol (see Appendix P). The pragma has the form

```
pragma INTERFACE_PARAMETERS (subprogram_name,
                             parameter_name => pragma_interface_parameter_locations Enumeration_literal
                             [parameter_name => pragma_interface_parameter_locations Enumeration_literal]);
```

Pragma `INTERFACE_RESULT` specifies the 1750A machine register to be used for a function's return result. If this pragma is not provided, registers will be used according to standard compiler protocol (see Appendix P). The pragma has the form

```
pragma INTERFACE_RESULT (subprogram_name, pragma_interface_parameter_locations Enumeration_literal);
```

This pragma will only be accepted for a function and cannot be used if the result type is an array or record.

Pragma `INTERFACE_SCRATCH` is used to identify the 1750A machine registers that will be used as scratch registers inside the macro. If the pragma is provided, the compiler will only save those registers specified in the pragma prior to BIF execution. If this pragma is not provided, the compiler will save all necessary registers prior to BIF execution. The pragma has the form

```
pragma INTERFACE_SCRATCH (subprogram_name, pragma_interface_parameter_locations Enumeration_literal
                          [pragma_interface_parameter_locations Enumeration_literal]);
```

F3.8. Pragma `INTERFACE_SPELLING`

This pragma is used to define the external name of a subprogram written in another language, if that external name is different from the subprogram name (if the names are the same, the pragma is not needed). The pragma has the form

```
pragma INTERFACE_SPELLING (subprogram_name, external_name_string_literal);
```

The pragma should appear after the pragma `INTERFACE` for the subprogram. This pragma is useful in cases where the desired external name contains characters that are not valid in Ada identifiers. For example,

```
procedure CONNECT_BUS (SIGNAL : INTEGER);
pragma INTERFACE (ASSEMBLY, CONNECT_BUS);
pragma INTERFACE_SPELLING (CONNECT_BUS, "$CONNECT.BUS");
```

F3.9. Pragma `MEMORY_UNIT`

This pragma is used in the Compiler System's support for *memory association*. This is where Ada objects (whether variables or constants) are associated at compile time with different *classes* of memory. Then at link time, these classes of memory can be treated differently. For instance, objects can be associated with fast memory or slow memory; with local or global memory in a multiprocessor environment; with different areas of memory in a signal processor/array processor/SIMD type of architecture; and so on.

The classes of memory are implemented through the InterACT 1750A Linker CSECT and section facilities (see *InterACT Linker Reference Manual* for a complete description of these facilities).

The types `MEMORY_SECTION_NUMBER` and `USER_MEMORY_SECTIONS` in package `SYSTEM` define the CSECT numbers available for use in connection with this pragma; the first type defines all those available in the 1750A Linker, the second subtype those available to users (not reserved by the compiler or runtime

executive).

The basic scheme of the memory association support is that the user defines an enumeration type naming the different classes of memory, and then a enumeration representation clause assigning each of those classes to a CSECT number. `Pragma MEMORY_UNIT` is then defined for Ada objects (or types, applying to all objects of the type), specifying the memory class for that object. The compiler allocates the object in a CSECT with the corresponding CSECT number. The user then creates 1750A Linker SECTION control statements to allocate the memory classes as desired.

The following type declarations define the memory classes. The user must code them, and they must be visible wherever `pragma MEMORY_UNIT` appears.

```
type MEMORY_UNIT is
  (memory_unit Enumeration_Literal [memory_unit Enumeration_Literal]);
subtype RESERVED_MEMORY_UNITS is MEMORY_UNIT range
  memory_unit Enumeration_Literal..memory_unit Enumeration_Literal

for MEMORY_UNIT use
  (memory_unit Enumeration_Literal => csect_number
   [memory_unit Enumeration_Literal => csect_number]);
```

The first declaration defines all the types of memory that (static data and literal) objects and types can be associated with, and the CSECT numbers to which they will be allocated. The second declaration specifies which of these kinds of memory may share a CSECT with existing compiler CSECTs (e.g. if `memory_unit Enumeration_Literal` is to contain both the stack/heap and some static data).

Associations of particular objects and types to memory is accomplished by the following:

```
pragma MEMORY_UNIT (memory_unit Enumeration_Literal, simple-name[,simple-name]);
```

where `simple-name` is a type or object. Up to 32 objects and 32 data types may be specified within each occurrence of the pragma.

Any base type, derived type, or objects of them may be associated. Only one association is allowed for a type or an object. Once a type is associated, all objects of that type inherit the association. When associating a type, it is necessary for the type to be declared in same package as the pragma, and the pragma to be located before any objects of that type are declared. Any object can be associated providing that its type was not associated.

This pragma may be used in any compilation unit but subprogram variables may only be associated with a memory that shares the heap/stack area.

This pragma cannot be used in conjunction with address clauses, collections or pragmas `ESTABLISH_OPTIMIZED_REFERENCE` and `ASSUME_OPTIMIZED_REFERENCE`.

F.3.10. Pragma SET_MACHINE_OVERFLOW FALSE FOR ANONYMOUS_FIXED

This pragma works on a program library basis. See the subsection at the beginning of this section for how such pragmas are used.

The effect of this pragma is that any fixed point type `T` of anonymous predefined *fixed* type (i.e., represented in 16 bits) has the attribute

TMACHINE_OVERFLOW = FALSE

such that **NUMERIC_ERROR** is not raised in overflow situations [*Ada RM 4.5.7 (7)*].

The result of operations in overflow situations is either the lower or upper bound of the "virtual" predefined type for **T** ([*Ada RM 3.5.9 (10)*], this document Section F.1), depending on the direction of overflow. These bounds are $-32_768 * \text{TSMALL}$ and $32_767 * \text{TSMALL}$ respectively. These bounds will equal **TFIRST** and **TLAST** if the range constraint for **T** is so declared.

Note that this implementation of fixed point types relies on the 1750A fixed point overflow interrupt being enabled and not masked; any user exit or customization routines in the Ada runtime executive must not do differently.

F3.11. Pragma SUBPROGRAM_SPELLING

This pragma is used to define the external name of an Ada subprogram. Normally such names are compiler-generated, based on the program library unit number. The pragma has the form

pragma SUBPROGRAM_SPELLING (*subprogram_name* [*external_name_string_literal*]);

The pragma is allowed wherever a pragma **INTERFACE** would be allowed for the subprogram. If the second argument is omitted, the subprogram name is used as the external name. If the resulting external name is longer than 31 characters, it will be so truncated.

This pragma is useful in cases where the subprogram is to be referenced from another language.

F.4. Implementation-dependent Attributes

None are defined.

F.5. Package SYSTEM

The specification of package **SYSTEM** is:

```

package SYSTEM is

  type ADDRESS      is new INTEGER;
  ADDRESS_NULL      : constant ADDRESS := 0;

  type NAME          is (MIL_STD_1750A);

  SYSTEM_NAME       : constant NAME := MIL_STD_1750A;

  STORAGE_UNIT      : constant := 16;
  MEMORY_SIZE       : constant := 64 * 1024;

  MIN_INT           : constant := -2_147_483_647-1;
  MAX_INT           : constant := 2_147_483_647;
  MAX_DIGITS        : constant := 9;
  MAX_MANTISSA      : constant := 31;
  FINE_DELTA        : constant := 1.0 / 2.0 ** MAX_MANTISSA;
  TICK              : constant := 0.000_100;

  subtype PRIORITY   is INTEGER range 0..255;

  type INTERFACE_LANGUAGE is (ASSEMBLY, BIF, JOVIAL, FORTRAN);

  type MEMORY_SECTION_NUMBER is range 0..31;
  subtype USER_MEMORY_SECTIONS is MEMORY_SECTION_NUMBER range 16..31;

  type PRAGMA_INTERFACE_PARAMETER_LOCATIONS is
    (R0, R1, R2, R3, R4, R5, R6, R7,
     R8, R9, R10, R11, R12, R13, R14, R15);

end SYSTEM;

```

F.6. Type Representation Clauses

The three kinds of type representation clauses – length clauses, enumeration representation clauses, and record representation clauses – are all allowed and supported by the compiler. This section describes any restrictions placed upon use of these clauses.

Change of representation [Ada RM 13.6] is allowed and supported by the compiler. Any of these clauses may be specified for derived types, to the extent permitted by Ada RM.

F.6.1. Length Clauses

The compiler accepts all four kinds of length clauses.

Size specification: T'SIZE

The size specification for a type T is accepted in the following cases.

If T is a discrete type then the specified size must be greater than or equal to the *minimal size* of the type, which is the number of bits needed to represent a value of the type, and must be less than or equal to the size of the underlying predefined integer type.

The calculation of the minimal size for a type is done not only in the context of length clauses, but also in the context of pragma PACK, record representation clauses, the T'SIZE attribute, and unchecked conversion. The definition presented here applies to all these contexts.

The *minimal size* for a type is the minimum number of bits required to represent all possible values of the type. When the minimal size is calculated for discrete types, the range is extended to include zero if necessary. That is, both signed and unsigned representations are taken into account, but not biased representations. Also, for unsigned representations, the component subtype must belong to the predefined integer base type normally associated with that many bits.

Some examples:

```
type SMALL_INT is range -2..1;
for SMALL_INT'SIZE use 2;  -- OK, signed representation, needs minimum 2 bits

type U_SMALL_INT is range 0..3;
for U_SMALL_INT'SIZE use 2;  -- OK, unsigned representation, needs minimum 2 bits

type B_SMALL_INT is range 7..10;
for B_SMALL_INT'SIZE use 2;  -- illegal, would be biased representation
for B_SMALL_INT'SIZE use 4;  -- OK, the extended 0..10 range needs minimum 4 bits

type U_BIG_INT is range 0..65_535;
for U_BIG_INT'SIZE use 16;  -- illegal, range outside of 16-bit INTEGER predefined type
for U_BIG_INT'SIZE use 17;  -- OK, range within (17-bit maps to) 32-bit LONG_INTEGER predefined type
```

If T is a fixed point type then the specified size must be greater than or equal to the minimal size of the type, and less than or equal to the size of the underlying predefined fixed point type. The same definition of minimal size applies as for discrete types.

If T is a floating point type, an access type or a task type, the specified size must be equal to the number of bits normally used to represent values of the type (floating point types 32 or 48, access types 16, task types 16).

If T is an array type the size of the array must be static and the specified size must be equal to the minimal number of bits needed to represent a value of the type. This calculation takes into account whether or not the array type is declared with pragma PACK.

If T is a record type the specified size must be greater than or equal to the minimal number of bits needed to represent a value of the type. This calculation takes into account whether or not the record type is declared with a record representation clause.

The effect of a size specification length clause for a type depends on the context the type is used in.

The allocation of objects of a type is unaffected by a length clause for the type. Objects of a type are allocated to one or more storage units of memory. The allocation of components in an array type is also unaffected by a length clause for the component type (unless the array type is declared with pragma PACK); components are allocated to one or more storage units. The allocation of components in a record type is always unaffected by a length clause for any component types; components are allocated to one or more storage units, unless a record representation clause is declared, in which case components are allocated according to the specified component clauses.

There are two important contexts where it is necessary to use a length clause to achieve a certain representation. One is with pragma PACK, when component allocations of a non-power-of-two bit size are desired (see Section F.2.8). The other is with unchecked conversions, where a length clause on a type can make that type's size equal to another type's, and thus allowed the unchecked conversion to take place (see Section F.9).

Specification of collection size: T'SORAGE_SIZE

This value controls the size of the collection (implemented as a local heap) generated for the given access type.

It must be in the range of the predefined type NATURAL. Space for the collection is deallocated when the scope of the access type is left.

See the *Ada Runtime Executive Programmer's Guide* for full details on how the storage in collections is managed.

Specification of storage for a task activation: TSTORAGE_SIZE

This value controls the size of the stack allocated for the given task. It must be in the range of the predefined type NATURAL.

It is also possible to specify, at link time, a default size for all task stacks, that is used if no length clause is present. See the *Ada Runtime Executive Programmer's Guide* for full details, and for a general description of how task stacks, and other storage associated with tasks, are allocated.

Specification of a *small* for a fixed point type

Any real value (less than the specified delta of the fixed point type) may be used.

F.6.2. Enumeration Representation Clauses

Enumeration representation clauses may only specify representations in the range of the predefined type INTEGER.

When enumeration representation clauses are present, the representation values (and not the logical values) are used for size and allocation purposes. Thus, for example,

```
type ENUM is (ABLE, BAKER, CHARLIE);
for ENUM use (ABLE => 1, BAKER => 4, CHARLIE => 9);

for ENUM'SIZE use 2;  -- illegal, 1..9 range needs minimum 4 bits
for ENUM'SIZE use 4;  -- OK

type ARR is array (ENUM) of INTEGER;  -- will occupy 9 words of storage, not 3
```

Enumeration representation clauses often lead to less efficient attribute and indexing operations, as noted in [Ada RM 13.3 (6)].

F.6.3. Record Representation Clauses

Alignment clauses are allowed, but the only permitted value is one.

In terms of allowable component clauses, record components fall into three classes:

- integer and enumeration types that may be represented in 16 bits or less;
- statically-bounded arrays or records composed solely of the above;
- all others.

Components of the "16-bit integer/enumeration" class may be given a component clause that specifies a storage place at any bit offset, and for any number of bits, as long as the storage place is greater than or equal to the

minimal size of the component type (see Section F.6.1) and does not cross a word boundary.

Components of the "array/record of 16-bit integer/enumeration" class may be given a component clause that specifies a storage place at any bit offset, if the size of the array/record is less than a word, or at a word offset otherwise, and for any number of bits, as long as the storage place is large enough to contain the component and none of the individual integer/enumeration elements of the array/record cross a word boundary. The component clause cannot specify a representation different from that of the component's type. Thus, an array component that is given a packed representation by a component clause must be of a type that is declared with pragma PACK; similarly, a record component that is given a non-standard representation by a component clause must be of a type that is declared with a record representation clause.

Components of the "all others" class may only be given component clauses that specify a storage place at a word offset, and for the number of bits normally allocated for objects of the underlying base type.

Components that do not have component clauses are allocated in storage places beginning at the next word boundary following the storage place of the last component in the record that has a component clause.

Records with component clauses cannot exceed 2K words (32K bits) in size.

F.7. Implementation-dependent Names for Implementation-dependent Components

None are defined.

F.8. Address Clauses

In general, address clauses are allowed and supported for objects, for subprogram and task units, and for interrupt entries. Address clauses are not allowed for package units.

Address clauses occurring within generic units are always allowed at that point, but are not allowed when the units are instantiated if they do not conform to the implementation restrictions described here. In addition, the effect of such address clauses may depend on the context in which they are instantiated (e.g. library package or subprogram; see below).

F.8.1. Address Clauses for Objects or Subprogram Units

Address clauses for objects or subprogram units must be static expressions of type ADDRESS in package SYSTEM.

Address clauses are not allowed for constant scalar objects with static initial values, as such objects are not allocated in memory.

Address clauses for objects declared within library packages cause the Compiler System to reserve space for the object at that address, since the object exists for virtually the entire length of Ada program execution. Address clauses for objects declared within subprograms do *not* cause space to be reserved for the object, since the object only exists during the subprogram's execution. It is the user's responsibility to reserve space for such objects (1750A Linker control statements may be used if desired).

Type ADDRESS is a 16-bit signed integer. Thus, addresses in the memory range 16#8000#..16#FFFF# (i.e., the upper half of 1750A memory) must be supplied as negative numbers, since the positive (unsigned) interpretations of those addresses are greater than ADDRESS'LAST. Furthermore, addresses in this range must be

declared as named numbers, with the named number (rather than a negative numeric literal) being used in the address clause. The hexadecimal address can be retained in the named number declaration, and user computation of the negative equivalent avoided, by use of the technique illustrated in the following example:

```
X : INTEGER;
for X use at 16#7FFF#; -- legal

Y : INTEGER;
for Y use at 16#FFFF#; -- illegal

ADDR_FFFF : constant := 16#FFFF# - 65536;
Y : INTEGER;
for Y use at ADDR_FFFF; -- legal, equivalent to unsigned 16#FFFF#
```

F.8.2. Address Clauses for Interrupt Entries

Address clauses for interrupt entries do not use type `SYSTEM.ADDRESS`; rather, the address clause must be a static integer expression in the range 0..15, naming the corresponding 1750A interrupt.

The following restrictions apply to interrupt entries. An interrupt entry must not have formal parameters. Direct calls to an interrupt entry are not allowed. An accept statement for an interrupt entry must not be part of a selective wait, i.e., must not be part of a select statement. If any exception can be raised from within the accept statement for an interrupt entry, the accept statement must include an exception handler.

When the accept statement is encountered, the task is suspended. If the specified interrupt occurs, execution of the accept statement begins. When control reaches end of the accept statement, the special interrupt entry processing ends, and the task continues normal execution. Control must again return to the point where the accept statement is encountered in order for the task to be suspended again, awaiting the interrupt.

There are many more details of how interrupt entries interact with the 1750A machine state and with the Runtime Executive. For these details, see the *Ada 1750A Runtime Executive Programmer's Guide*.

F.9. Unchecked Conversion

Unchecked type conversions are allowed and supported by the compiler.

Unchecked conversion is only allowed between types that have the same size. In this context, the size of a type is the *minimal size* (see Section F.6.1), unless the type has been declared with a size specification length clause, in which case the size so specified is the size of the type.

In addition, if `UNCHECKED_CONVERSION` is instantiated with an array type, that array type must be statically constrained.

In general, unchecked conversion operates on the data for a value, and not on type descriptors or other compiler-generated entities.

For values of scalar types, array types, and record types, the data is that normally expected for the object. Note that objects of record types may be represented in two ways that might not be anticipated: there are compiler-generated extra components representing array type descriptors for each component that is a discriminant-dependent array, and all dynamically-size array components (whether discriminant-dependent or not) are

represented indirectly in the record object, with the actual array data in the system heap.

For values of an access type, the data is the address of the designated object; thus, unchecked conversion may be done in either direction between access types and type `SYSTEM_ADDRESS` (which is derived from type `INTEGER`). (The only exception is that access objects of unconstrained access types which designate unconstrained array types cannot reliably be used in unchecked conversions.) The named number `SYSTEM_ADDRESS_NULL` supplies the type `ADDRESS` equivalent of the access type literal `null`.

For values of a task type, the data is the address of the task's Task Control Block (see the *Ada 1750A Runtime Executive Programmer's Guide*).

For unchecked conversions involving types with a size less than a full word of memory, and different representational adjustment within the word (scalar types are right-adjusted within a word, while composite types are left-adjusted within a word), the compiler will correctly re-adjust the data as part of the conversion operation.

Some examples to illustrate all of this:

```

type BOOL_ARR is array(1..16) of BOOLEAN;
pragma PACK (BOOL_ARR);

function UC is new UNCHECKED_CONVERSION (BOOL_ARR, INTEGER);  -- OK, both have size 16

type BITS_8 is array(1..8) of BOOLEAN;
pragma PACK (BITS_8);

function UC is new UNCHECKED_CONVERSION (BITS_8, INTEGER);  -- illegal, sizes are 8 and 16

type SMALL_INT is range -128..127;
function UC is new UNCHECKED_CONVERSION (BITS_8, SMALL_INT);  --OK, both have size 8

type BYTE is range 0..255;
function UC is new UNCHECKED_CONVERSION (BITS_8, BYTE);  --OK, both have size 8

type BIG_BOOLEAN is new BOOLEAN;
for BIG_BOOLEAN'SIZE use 8;
function UC is new UNCHECKED_CONVERSION (BITS_8, BIG_BOOLEAN);  --OK, both have size 8

SM : SMALL_INT;  -- actual data is rightmost byte in object's word
BI : BITS_8;    -- actual data is leftmost byte in object's word

SM := UC (BI);  -- actual data is moved from leftmost to rightmost byte as part of conversion

```

Calls to instantiations of `UNCHECKED_CONVERSION` are always generated as inline calls by the compiler.

The instantiation of `UNCHECKED_CONVERSION` as a library unit is not allowed. Instantiations of `UNCHECKED_CONVERSION` may not be used as generic actual parameters.

F.10. Other Chapter 13 Areas

F.10.1. Change of Representation

Change of representation is allowed and supported by the compiler.

F.10.2. Representation Attributes

All representation attributes [*Ada RM 13.7.2, 13.7.3*] are allowed and supported by the compiler.

For certain usages of the X'ADDRESS attribute, the resulting address is ill-defined. These usages are: the address of a constant scalar object with a static initial value (which is not located in memory), the address of a loop parameter (which is not located in memory), and the address of an inlined subprogram (which is not uniquely located in memory). In all such cases the value SYSTEM.ADDRESS_NULL is returned by the attribute, and a warning message is issued by the compiler.

When the X'ADDRESS attribute is used for a package, the resulting address of that of the machine code associated with the package specification.

The X'SIZE attribute, when applied to a type, returns the *minimum size* for that type. See Section F.6.1 for a full definition of this size. However, if the type is declared with a size specification length clause, then the size so specified is returned by the attribute.

Since objects may be allocated in more space than the minimum required for a type (see Section F.6.1), but not less, the relationship $O'SIZE \geq T'SIZE$ is always true, where O is an object of type T.

F.10.3. Machine Code Insertions

Machine code insertions are not allowed by the compiler. Note that pragma INTERFACE (BIF) may be used as an alternative to machine code insertions.

F.10.4. Unchecked Deallocation

Unchecked storage deallocation is allowed and supported by the compiler.

Calls to instantiations of UNCHECKED_DEALLOCATION are always generated as inline calls by the compiler.

The instantiation of UNCHECKED_DEALLOCATION as a library unit is not allowed. Instantiations of UNCHECKED_DEALLOCATION may not be used as generic actual parameters.

F.11. Input-Output

The predefined library generic packages and packages `SEQUENTIAL_IO`, `DIRECT_IO`, and `TEXT_IO` are supplied. However, file input-output is not supported except for the standard input and output files. Any attempt to create or open a file will result in `USE_ERROR` being raised.

`TEXT_IO` operations to the standard input and output files are implemented as input from or output to some visible device for a given implementation of MIL-STD-1750A. Depending on the implementation, this may be a console, a workstation disk drive, simulator files, etc. See the *Ada 1750A Runtime Executive Programmer's Guide* for more details. Note that by default, the standard input file is empty.

The range of the type `COUNT` defined in `TEXT_IO` and `DIRECT_IO` is `0..SYSTEM.MAX_INT`.

The predefined library package `LOW_LEVEL_IO` is empty.

In addition to the predefined library units, a package `STRING_OUTPUT` is also included in the predefined library. This package supplies a very small subset of `TEXT_IO` operations to the standard output file. The specification is:

```
package STRING_OUTPUT is
  procedure PUT (ITEM : in STRING);
  procedure PUT_LINE (ITEM : in STRING);
  procedure NEW_LINE;
end STRING_OUTPUT;
```

By using the `'IMAGE` attribute function for integer and enumeration types, a fair amount of output can be done using this package instead of `TEXT_IO`. The advantage of this is that `STRING_OUTPUT` is smaller than `TEXT_IO` in terms of object code size, and faster in terms of execution speed.

Use of `TEXT_IO` in multiprogramming situations (see Chapter 5) may result in unexpected exceptions being raised, due to the shared unit semantics of multiprogramming. In such cases `STRING_OUTPUT` may be used instead.

F.12. Compiler System Capacity Limitations

The following capacity limitations apply to Ada programs in the Compiler System:

- the space available for the constants of a compilation unit is 32K words;
- the space available for the static data of a compilation unit is 32K words;
- any single object can not exceed 32K words;
- the space available for the objects local to a subprogram or block statement is 32K words;
- the names of all identifiers, including compilation units, may not exceed the number of characters specified by the `INPUT_LINELENGTH` component in the compiler configuration file (see Section 4.1.4);

- a sublibrary can contain at most 4096 compilation units (library units or subunits). A program library can contain at most eight levels of sublibraries, but there is no limit to the number of sublibraries at each level. An Ada program can contain at most 32768 compilation units.

The above limitations are all diagnosed by the compiler. Most may be circumvented straightforwardly by using separate compilation facilities.