

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

AD-A240 512

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

RT 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
Corporation, InterACT Ada Mips Cross-Compiler System, Rel 2.0, MicroVAX 3100
Cluster (Host) to Lockheed Sanders STAR MVP R3000 (Target) 910705S1.11192

5. FUNDING NUMBERS

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 USA8. PERFORMING ORGANIZATION
REPORT NUMBER

NIST90ACT520_2_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. 20301-308110. 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 Corporation, InterACT Ada Mips Cross-Compiler System, Release 2.0, Gaithersburg,
MicroVAX 3100 Cluster (Host) to Lockheed Sanders STAR MVP R3000 (Target) ACVC 1.11.DTIC
SELECTED
SEP 19 1991
S D

91-11068



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
UNCLASSIFIED18. SECURITY CLASSIFICATION
UNCLASSIFIED19. SECURITY CLASSIFICATION
OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT

AVF Control Number: NIST90ACT520_2_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: 910705S1.11192

InterACT Corporation

InterACT Ada Mips Cross-Compiler System, Release 2.0

MicroVAX 3100 Cluster => Lockheed Sanders STAR MVP

R3000/R3010 board

(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	
NTIS GRA&I	☒
DIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution	
Date	
Dist	
A-1	



AVF Control Number: NIST90ACT520_2_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 Mips Cross-Compiler System, Release 2.0

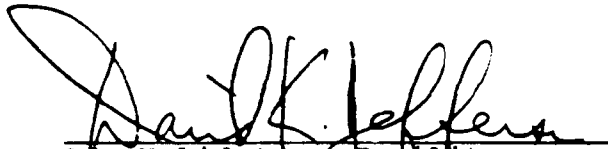
Host Computer System: MicroVAX 3100 Cluster running under VAX/VMS, Version 5.2

Target Computer System: Lockheed Sanders STAR MVP R3000/R3010 board (Bare Machine)

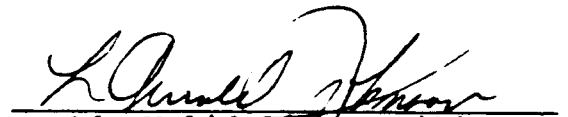
See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 910705S1.11192 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)

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


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


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 Mips Cross-Compiler System

Version: 2.0

Host Computer System: MicroVAX 3100 Cluster /VMS 5.2

Target Computer System: Lockheed Sanders STAR MVP R3000/R3010 Board
(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 Port Royal Road
Springfield VA 22161

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

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

1.2 REFERENCES

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

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

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

1.3 ACVC TEST CLASSES

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

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

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

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

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

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly 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
BD1302B	BD1306A	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 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)

C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

C24113I..K (3 TESTS) USE A LINE LENGTH IN THE INPUT FILE WHICH EXCEEDS 126 CHARACTERS.

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

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

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

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

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

C35713B, C45423B, B86001T, and C36006H 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`.

C4A013B contains a static universal real expression that exceeds the range of this implementation's largest floating-point type; this expression is rejected by the compiler.

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.

BD8001A, 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 17 tests.

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

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).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described by the information given in the initial pages of this report with additional information as follows:

In addition to the host computer system and the target computer system, there are execution controllers which are a pair of cooperating processes. The Remote Process Administrator (RPA) runs under VAX/VMS, and is a translator/downloader. The Remote Process Monitor (RPM) runs on the target Mips machine (the Lockheed Sanders STAR MVP R3000/R3010 board (Bare Machine)). The two processes communicate via a RS232 link.

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	3527	
b) Total Number of Withdrawn Tests	94	
c) Processed Inapplicable Tests	549	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests		0
f) Total Number of Inapplicable Tests	549	(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.

After an Ada program is compiled under VAX/VMS, the InterACT Embedded Systems Linker is run under VAX/VMS and produces a Mips load module. This module is in InterACT's proprietary load format.

The execution controllers are a pair of cooperating processes. The Remote Process Administrator (RPA) runs under VAX/VMS, and is a translator/downloader. The Remote Process Monitor (RPM) runs on the target Mips machine (the Lockheed Sanders STAR MVP R3000/R3010 board (Bare Machine)). The two processes communicate via a RS232 link. The RPM (running on the Mips target computer) is constantly executing waiting for requests from the RPA process on the host computer.

The RPA is invoked with a Mips load module as input. The RPA translates the load module to one or more Unix style (a.out) format files. The RPA then instructs the RPM to download the file(s) via a pair of Ethernet server/client processes.

The RPA then directs the RPM to start the execution of the Ada program. The RPM starts the execution of the Ada program by branching to the program's starting address.

As the Ada program executes, it calls on the RPM to perform input/output. The RPM converses with the RPA (executing on the

host computer), conveying input/output between the Ada program and the RPA which logs the output data in a disk file under VAX/VMS.

When the Ada program finishes its execution, it gives control back to the RPM. The RPA then gives control back to the user under VAX/VMS.

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 executable images were transferred to the target computer system by the communications link described above, and run. 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	32
\$ALIGNMENT	2
\$COUNT_LAST	2_147_483_647
\$DEFAULT_MEM_SIZE	4*1024*1024*1024
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	MIPS
\$DELTA_DOC	1.0/2.0**(SYSTEM.MAX_MANTISSA)
\$ENTRY_ADDRESS	SYSTEM.MODx
\$ENTRY_ADDRESS1	SYSTEM.TLBL
\$ENTRY_ADDRESS2	SYSTEM.TLBS
\$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	131_071.0
\$GREATER_THAN_DURATION_BASE_LAST	131_072.0
\$GREATER_THAN_FLOAT_BASE_LAST	2#1.0#E129
\$GREATER_THAN_FLOAT_SAFE_LARGE	2#0.111111111111111111111111#E126
\$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 -2147483648
 \$INTEGER_LAST 2147483647
 \$INTEGER_LAST_PLUS_1 2147483648
 \$INTERFACE_LANGUAGE ASSEMBLY
 \$LESS_THAN_DURATION -131_072.0
 \$LESS_THAN_DURATION_BASE_FIRST -131_073.0
 \$LINE_TERMINATOR ''
 \$LOW_PRIORITY 0
 \$MACHINE_CODE_STATEMENT NULL;
 \$MACHINE_CODE_TYPE NO_SUCH_TYPE
 \$MANTISSA_DOC 31
 \$MAX_DIGITS 15
 \$MAX_INT 2147483647
 \$MAX_INT_PLUS_1 2147483648
 \$MIN_INT -2147483648
 \$NAME NO_SUCH_INTEGER_TYPE
 \$NAME_LIST MIPS
 \$NAME_SPECIFICATION1 NAME_SPEC_1
 \$NAME_SPECIFICATION2 NAME_SPEC_2
 \$NAME_SPECIFICATION3 NAME_SPEC_3

\$NEG_BASED_INT	16#FFFFFFFFE#
\$NEW_MEM_SIZE	4*1024*1024*1024
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	MIPS
\$PAGE_TERMINATOR	' '
\$RECORD_DEFINITION	NEW_INTEGER;
\$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	1024
\$TICK	2.0**(-14)
\$VARIABLE_ADDRESS	16#800E0000#-2**32
\$VARIABLE_ADDRESS1	16#800E8000#-2**32
\$VARIABLE_ADDRESS2	16#80100000#-2**32
\$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 Mips R2000/R3000 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 Mips 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:

```
$ adamips{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=adamips_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 `adamips_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=adamips_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 `adamips_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[ = full_optimizations | limit_optimizations ]`
`/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.

If `/debug=full_optimizations` is specified (the default if `/debug` is active), the compiler will generate code with all optimizations enabled, even though this may result in some unreliable symbolic debug information being produced. If `/debug=limit_optimizations` is specified, the compiler will suppress those optimizations which might result in unreliable symbolic debug information. These optimizations include code motion across Ada statement boundaries, and not storing the values of Ada variables to memory across statement boundaries. Users may also wish to specify this option to make the generated machine code more understandable relative to the Ada source code.

`/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`
`/nopprogress` (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.

`/gisa`

Use of this qualifier directs the compiler to accept an extended set of address clauses for interrupt entries, corresponding to additional interrupts found in the GISA architecture (see Sections F.5 and F.8).

Examples of qualifier usage

```
$ adamips navigation_constants
$ adamips/list/xref event_scheduler
$ adamips/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 are two such files for each compilation unit in the source file. Generated assembly list files have a file type of `s` and `l`, 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 consists of two separate files. One (with file type `s`) is the generated Mips assembly source produced by the compiler for a compilation unit, prior to being assembled. The other (with file type `l`) is a disassembly listing of the object file produced by assembling the generated assembly source. (The assembly takes place as part of the `compile` command.) The disassembly listing shows the generated machine instructions (which are often different from the assembly source, due to the reorganizing and other transformations made by the Mips Assembler) and corresponding object code. Both files contain interleaved Ada source text, as described below.

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 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 `$file` directive 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), then no Ada source comments appear from that file.

The compiler unnests lexically nested subprogram bodies and task bodies in the generated code so that they appear textually after their parent scopes. This may lead to the Ada source comments for those nested bodies appearing twice in the generated code.

The bottom of the disassembly 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).

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.

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.

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 "Mips 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 Mips part of the link takes place. This consists of assembling the elaboration caller routine, then invoking the InterACT Mips Embedded Systems 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 Mips 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 Mips Runtime Executive Programmer's Guide*. This facility may also be used to modify or add to the standard Mips Embedded Systems Linker control statements that are used in the Mips part of the link; in this way, target memory may be precisely defined. The control statements involved are described in the *InterACT Mips Embedded Systems Linker Reference Manual*.

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 expanded memory link is being done). command.

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 Mips 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.

5.1. The Invocation Command

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

```
$ adamips/link{qualifier} main-program-name{,main-program-name}
```

As part of the "Mips 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 "Mips 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 = *adamips_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 *adamips_library* is used as current sublibrary.

/mp

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

/options = "-Dsymbol-name = value {,symbol-name = value}"

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 a quoted string, beginning with -D, containing one or more special symbol assignments that override the default values of these symbols. Numeric values are treated as decimal.

If a multiprogramming link is done, suffixes are used in the special symbol names to indicate which programs the overrides are for.

Since the `/options` value cannot be continued onto a new line, an alternative method is available if a large number of overrides must be specified. This involves creating a file of Mips Assembler preprocessor directives specifying the overrides, and then defining that file with the logical name `adamips_runtime_options`.

The names of these special symbols, their default values, and the runtime behavior that they control, are described in the *Ada Mips Runtime Executive Programmer's Guide*, as are the details of the alternative method.

```
/standard_control[ =file-spec]
/standard_control=adamips_standard_control      (default)
```

This qualifier specifies the file name of "standard" Mips Embedded Systems 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, `[ ]adamips.ctl` is used as a full or partial default. If the qualifier is omitted, the logical name `adamips_standard_control` 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" Mips Embedded Systems Linker control statements that are to be used for this particular link. If *file-spec* is omitted or only partially specified, `[ ]main-program-name.ctl` 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 Mips Embedded Systems 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 `object_file` statements for the elaboration caller routine(s) and main program(s).

The Compiler System is delivered with `adamips_standard_control` defined to a file that contains a default set of standard control statements. These consist of the minimal relocation statements required by the Mips Embedded Systems Linker, and various other necessary directives.

```
/user_rts=search-list
/user_rts=adamips_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 directories (including those RTE modules in the predefined library). If the qualifier is omitted, logical name `adamips_user_rts` is used, if the name has been defined.

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

This qualifier specifies a VAX/VMS search list that contains Mips-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 `adamips_target_rts` is used, if the name has been defined.

```
/debug
/noddebug      (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.

```
/eslink_qualifiers = "Mips Embedded Systems Linker qualifiers"
```

This qualifier specifies a string containing one or more command qualifiers to be passed to the execution of the Mips Embedded Systems 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 "Mips part", but before that command file has actually been invoked.

When used with *number* = 1, the command file is invoked, but stops before the Mips Embedded Systems Linker is invoked, leaving the temporary subdirectory and its files in place. When used with *number* = 2, it executes the Mips Embedded Systems 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

Some examples of single program and multiprogramming links:

```
$ adamips/link flight_simulator ! single program
$ adamips/link/mp able,baker,charlie ! multiprogramming
```

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

```
$ adamips/link/opt="-Drtheapsz1=48*1024,rtmstacksz1=8000" flight_simulator
```

An example of overriding values when multiprogramming is involved (the system heap size is overridden for each program):

```
$ adamips/link/mp/opt="-Drtheapsz1=20*1024,rtheapsz2=12*1024,rtheapsz3=50*1024" able,baker,charlie
```

Now, an example of introducing "user" Mips Embedded Systems Linker control statements:

```
$ adamips/link/control test_driver
```

where `test_driver.ctl` in the current directory contains

```
search_path is
  [dms.object]
end
object_file is
  dmscheck
end
informational messages are off
```

Now, an example of the use of user and target RTE directories:

```
$ define adamips_target_rts [tektronics.io.test],[tektronics.io]
$ adamips/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 `adamips_target_rts` logical name, and lastly in the standard RTE directory.

To revert to referencing only the standard RTE directory:

```
$ deassign adamips_target_rts
$ adamips/link flight_simulator
```

5.2. Load Module Output

If an Ada linking is successfully completed, the Mips Embedded Systems 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 Mips 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 Mips Embedded Systems Linker memory map file, *main-program-name.map*. (See the *InterACT Mips Embedded Systems Linker Reference Manual* for contents.)
6. An assembly listing of the generated module that elaborates all library units, *e\$main-program-name.s* and *J*. If a multiprogramming link is done, separate listings are produced for each program. It may sometimes be useful to see the expanded assembly source of this module, if default runtime executive values have been overridden; this may be produced by use of the `/noassemble` qualifier of the InterACT Mips Assembler.
7. If a multiprogramming link is done, an assembly listing of a generated module that communicates program information to the Ada runtime executive, *\$mpt.s* (no disassembly listing is produced as this module has no code). The same note applies as above about expanded assembly source.

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 "Mips part" of the link.

5.3.1. 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 Recompiler (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 Mips R2000/R3000 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, `useslm` for the InterACT Mips 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 Mips Runtime Executive Programmer's Guide*.

APPENDIX F OF THE Ada 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 Ada Mips Cross-Compiler System, 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

One predefined integer type is implemented, INTEGER. It has the following attributes:

INTEGER'FIRST	=	-2_147_483_648
INTEGER'LAST	=	2_147_483_647
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'FIRST	=	-2#1.0#E128
FLOAT'LAST	=	2#0.11111111111111111111#E128
FLOAT'MACHINE_EMAX	=	128
FLOAT'MACHINE_EMIN	=	-125
FLOAT'MACHINE_MANTISSA	=	24
FLOAT'MACHINE_OVERFLOW	=	TRUE
FLOAT'MACHINE_RADIX	=	2
FLOAT'MACHINE_ROUNDS	=	TRUE
FLOAT'SAFE_EMAX	=	125
FLOAT'SAFE_LARGE	=	2#0.11111111111111111111#E125

DURATION'FIRST	=	-131_072.0
DURATION'FORE	=	7
DURATION'LARGE	=	1.31071999938965E05
DURATION'LAST	=	131_071.0
DURATION'MANTISSA	=	31
DURATION'SAFE_LARGE	=	DURATION'LARGE
DURATION'SAFE_SMALL	=	DURATION'SMALL
DURATION'SIZE	=	32
DURATION'SMALL	=	2**(-14) = 6.10351562500000E-05

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`.

Language ASSEMBLY

Ada programs may call assembly language subprograms that have been assembled with the VAX/VMS-hosted InterACT Mips Assembler. The compiler generates a call to the name of the subprogram (in all upper case). If a call to a different external name is desired, use pragma `INTERFACE_SPELLING` in conjunction with pragma `INTERFACE` (see Section F.3).

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 InterACT Mips Embedded Systems Linker can find it.

Languages C, FORTRAN, and Pascal

It is possible to use pragma `INTERFACE` to call subprograms written in these other languages supported by MIPS Computer Systems, Inc. compilers. This is because the object code format and the compiler protocols [*MIPS Appendix D*] used by the Compiler System are the same as those used in the MIPS-supplied compilers.

To do this, compile such subprograms on a MIPS computer system (making sure they are compiled for a big-endian configuration), and then transfer the object files (and any language runtime library object files needed by the subprograms) to VAX/VMS. (Make sure the transfer preserves the binary nature of the files.) Then proceed as with assembly language subprograms.

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 32 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 Mips 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).

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` results in the corresponding checking code not being generated. The implementation of arithmetic operations is such that, in general, pragma `SUPPRESS` with `DIVISION_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 Mips Runtime Executive Programmer's Guide* for details).

F.2.14. Pragma SYSTEM_NAME

This pragma has no effect. The only possible `SYSTEM_NAME` is `Mips`. 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. 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 in all upper case is used as the external name. Note that the Mips Assembler is case-sensitive; the second argument must be used if the external name is to be other than all upper case.

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, "Baker");
end GLOBAL;
```

may be accessed in the following assembly language fragment

```
lw    $8,ABLE        # get value of ABLE
la    $9,Baker        # get address of Baker
```

F.3.2. Pragma IMPORT

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

```
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 in all upper case is used as the external name. Note that the Mips Assembler is case-sensitive; the second argument must be used if the external name is to be other than all upper case.

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, "Baker");
end GLOBAL;
```

are actually defined and allocated in the following assembly language fragment

```
.globl  ABLE
.lcomm  ABLE, 4

.globl  Baker
.lcomm  Baker, 8
```

F.3.3. 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). Note that the Mips Assembler is case-sensitive; this pragma must be used if the external name is to be other than all upper case. 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");
```

F.3.4. 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 object name in all upper case is used as the external name. Note that the Mips Assembler is case-sensitive; the second argument must be used if the external name is to be other than all upper case.

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 (Nips);

  SYSTEM_NAME       : constant NAME := Nips;

  STORAGE_UNIT      : constant := 8;
  MEMORY_SIZE       : constant := 4 * 1024 * 1024 * 1024;

  MIN_INT           : constant := -2_147_483_647-1;
  MAX_INT           : constant := 2_147_483_647;
  MAX_DIGITS        : constant := 15;
  MAX_MANTISSA      : constant := 31;
  FINE_DELTA        : constant := 1.0 / 2.0 ** MAX_MANTISSA;
  TICK              : constant := 1.0 / 2.0 ** 14;

  subtype PRIORITY   is INTEGER range 0..255;

  type INTERFACE_LANGUAGE is (Assembly, C, Fortran, Pascal);

  -- these are the possible ADDRESS values for interrupt entries
  MODx      : constant := 1 * 2**2;    -- (MOD is reserved word)
  TLBL      : constant := 2 * 2**2;
  TLBS      : constant := 3 * 2**2;
  AdEl      : constant := 4 * 2**2;
  AdES      : constant := 5 * 2**2;
  lBE       : constant := 6 * 2**2;
  DBE       : constant := 7 * 2**2;
  Sys       : constant := 8 * 2**2;
  Sp        : constant := 9 * 2**2;
  R1        : constant := 10 * 2**2;
  CpU       : constant := 11 * 2**2;
  Dvf       : constant := 12 * 2**2;
  Reserved13 : constant := 13 * 2**2;
  Reserved14 : constant := 14 * 2**2;
  Reserved15 : constant := 15 * 2**2;
  BU0       : constant := 2**0 * 2**8;
  BU1       : constant := 2**1 * 2**8;
  IP0       : constant := 2**0 * 2**10;
  IP1       : constant := 2**1 * 2**10;
  IP2       : constant := 2**2 * 2**10;
  IP3       : constant := 2**3 * 2**10;
  IP4       : constant := 2**4 * 2**10;
  IP5       : constant := 2**5 * 2**10;
  -- these are only meaningful for the GISA processor
  GISA0     : constant := IP0 + 1 + 0;
  GISA1     : constant := IP0 + 1 + 1;
  GISA2     : constant := IP0 + 1 + 2;
  GISA3     : constant := IP0 + 1 + 3;
  GISA4     : constant := IP0 + 1 + 4;
  GISA5     : constant := IP0 + 1 + 5;

```

```

GISA6      : constant := IP0 + 1 + 6;
GISA7      : constant := IP0 + 1 + 7;
GISA8      : constant := IP0 + 1 + 8;
GISA9      : constant := IP0 + 1 + 9;
GISA10     : constant := IP0 + 1 + 10;
GISA11     : constant := IP0 + 1 + 11;
GISA12     : constant := IP0 + 1 + 12;
GISA13     : constant := IP0 + 1 + 13;
GISA14     : constant := IP0 + 1 + 14;
GISA15     : constant := IP0 + 1 + 15;
GISA16     : constant := IP0 + 1 + 16;
GISA17     : constant := IP0 + 1 + 17;
GISA18     : constant := IP0 + 1 + 18;
GISA19     : constant := IP0 + 1 + 19;
GISA20     : constant := IP0 + 1 + 20;
GISA21     : constant := IP0 + 1 + 21;
GISA22     : constant := IP0 + 1 + 22;
GISA23     : constant := IP0 + 1 + 23;
GISA24     : constant := IP0 + 1 + 24;
GISA25     : constant := IP0 + 1 + 25;
GISA26     : constant := IP0 + 1 + 26;
GISA27     : constant := IP0 + 1 + 27;
GISA28     : constant := IP0 + 1 + 28;
GISA29     : constant := IP0 + 1 + 29;
GISA30     : constant := IP0 + 1 + 30;
GISA31     : constant := IP0 + 1 + 31;

```

```
end SYSTEM;
```

Note that since timers are not part of the Mips architecture specification, different Mips R2000/R3000 target implementations may contain timers with varying characteristics. This has an effect on the granularity of the CLOCK function in package CALENDAR. The value of the named number TICK above, which represents the granularity, corresponds to the Mips R2000/R3000 target implementation that the InterACT Ada Mips Cross-Compiler System is validated upon. It also is the most common value for the different Mips R2000/R3000 target implementations that the Compiler System supports; however, for some supported target implementations, it is incorrect.

For more details on timers and the different Mips R2000/R3000 target implementations, see the *Ada Mips Runtime Executive Programmer's Guide*.

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..2**32-1;
for U_BIG_INT'SIZE use 32;  -- illegal, range outside of 32-bit 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 (32 or 64 for floating point types, 32 for access and task types).

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 conversion, 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: TSTORAGE_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 Mips 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 Mips 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: TSMALL

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 storage units, 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. The permitted values are 1, 2, and 4.

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

- integer and enumeration types that may be represented in 32 bits or less;
- statically-bounded arrays or records composed solely of the above;
- all others.

Components of the "32-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 32-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 1K words (32K bits) in size.

The ordering of bits within storage units is defined to be big-endian. That is, bit 0 is the most significant bit and bit 31 is the least significant bit. Note that this convention differs from the one used in [MIPS p. 2-6] for bit-ordering.

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

None are defined.

F.8. Address Clauses

Address clauses are allowed for variables (objects that are not constants), and for interrupt entries. Address clauses are not allowed for constant objects, or for subprogram, package, or task 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. (Note that the effect of such address clauses may depend on the context in which they are instantiated; for example, whether multiple address clauses specifying the same address are erroneous may depend on whether they are instantiated into library packages or subprograms.)

F.8.1. Address Clauses for Variables

Address clauses for variables must be static expressions of type ADDRESS in package SYSTEM.

It is the user's responsibility to reserve space at link time for the object. See the *Mips Embedded Systems Linker Reference Manual* for the means to do this. Note that to conform with Compiler System assumptions, space so reserved should begin and end on 16-byte storage boundaries, even if the variable itself is not allocated on a 16-byte storage boundary. Also note that any bit-addressed object (a packed array or a record with a representation clause) must be allocated on a fullword (4-byte) boundary.

Type ADDRESS is a 32-bit signed integer. Thus, addresses in the memory range 16#8000_0000#..16#FFFF_FFFF# (i.e., the upper half of target 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_FFFF#; -- legal

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

ADDR_HIGH : constant := 16#FFFF_FFFF# - 2**32;
Y : INTEGER;
for Y use at ADDR_HIGH; -- legal, equivalent to unsigned 16#FFFF_FFFF#
```

F.8.2. Address Clauses for Interrupt Entries

Address clauses for interrupt entries do not use target addresses but rather, the values in the target Cause register that correspond to particular interrupts. For convenience these values are defined as named numbers in package SYSTEM, corresponding to the mnemonics used in [MIPS pp. 5-4, 5-5]. Note that if the /gisa compile qualifier is active, indicating that the target is the Westinghouse GISA architecture, an additional set of interrupt values is available (see Sections 4.1.1 and F.5).

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 target machine state and with the Runtime Executive. For these details, see the *Ada Mips 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. Note however that due to compiler assumptions about the machine alignment properties of objects, unchecked conversions from `SYSTEM.ADDRESS` to access objects must be done on 4-byte (word) aligned addresses only.

For values of a task type, the data is the address of the task's Task Control Block (see the *Ada Mips 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..32) of BOOLEAN;
pragma PACK (BOOL_ARR);

function UC is new UNCHECKED_CONVERSION (BOOL_ARR, INTEGER);  -- OK, both have size 32

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 32

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
SI : BITS_8;     -- actual data is leftmost byte in object's word
SM := UC (SI);   -- 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 `SYSTEMADDRESS_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 is 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 (ASSEMBLY)` may be used as a (non-inline) 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 Mips R2000/R3000 target implementation. Depending on the implementation, this may be a console, a workstation disk drive, simulator files, etc. See the *Ada Mips 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 device connected to the standard output file. (It does not use the actual standard output file object of `TEXT_IO`, so `TEXT_IO` state functions such as `COL`, `LINE`, and `PAGE` are unaffected by use of this package).

The specification of `STRING_OUTPUT` 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 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.

F.13. Implementation-dependent Predefined Library Units

In addition to the predefined library units required by [*Ada RM Annex C*], the predefined library in the Compiler System is delivered with several other library units that application developers may be interested in. These are:

- package `STRING_OUTPUT`, described in Section F.11 above
- a number of packages constituting the Application Runtime Interfaces, which allow for applications to access or control runtime executive functions in ways that are in addition to, or an alternate to, standard Ada language features.
- generic package `GENERIC_MATH_FUNCTIONS`. This is a public domain math package, taken from the Ada Software Repository, based on the algorithms of Cody and Waite. It supplies a set of elementary mathematics functions. The source for both the specification and the body of the package can be extracted from the predefined library through the Ada PLU type command.

In addition to these units, there are also a number of units in the predefined library that are used as part of the runtime system itself. These are "called" by the code generated by the compiler, and are not intended for direct use by application developers.