AD-A276 181

TION PAGE

Form Approved OPM No.

Public repo... and maintai... suggestion... 22202-430... ...our per response, including the time for reviewing instructions, searching existing data sources gathering ... d comments regarding this burden estimate or any other aspect of this collection of information, including ... orate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA ... agement and Budget, Washington, DC 20503.

1. AGENCY USE (Leave

2. REPORT

3. REPORT TYPE AND DATES

4. TITLE AND
DACS MIPS R3000 bare Ada Cross Compiler System, Version 4.7.1
Host: Sun SPARCstation IPX running under SunOS, Release 4.1.3
Target: DACS Sun SPARC/SunOS to MIPS R3000 Bare Instruction Set Architecture Simulator
Version: 4.7.1

5. FUNDING

6. Authors:

National Institute of Standards and Technology
Gaithersburg, Maryland

7. PERFORMING ORGANIZATION NAME(S) AND

National Institute of Standards and Technology
Building 255, Room A266
Gaithersburg, Maryland 20899
USA

8. PERFORMING ORGANIZATION

9. SPONSORING/MONITORING AGENCY NAME(S) AND

Ada Joint Program Office
The Pentagon, Rm 3E118
Washington, DC 20301-3080

10. SPONSORING/MONITORING AGENCY

11. SUPPLEMENTARY

12a. DISTRIBUTION/AVAILABILITY

Approved for Public Release; distribution unlimited

12b. DISTRIBUTION

13. (Maximum 200
DACS MIPS R3000 bare Ada Cross Compiler System, Version 4.7.1, Host: Sun SPARCstation IPX running under SunOS, Release 4.1.3, Target: DACS Sun SPARC/SunOS to MIPS R3000 Bare Instruction Set Architecture Simulator, Version 4.7.1

931119S1.11332

14. SUBJECT

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

15. NUMBER OF

16. PRICE

17. SECURITY CLASSIFICATION	18. SECURITY	19. SECURITY CLASSIFICATION	20. LIMITATION OF
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED

NSN

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

AVF Control Number: NIST92DDI510_5_1.11
DATE COMPLETED
BEFORE ON-SITE: 93-11-12
AFTER ON-SITE: 93-11-19
REVISIONS: 93-12-10

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

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 931119S1.11332
DDC-I, Inc.
DACS MIPS R3000 bare Ada
Cross Compiler System, Version 4.7.1
Sun SPARCstation IPX => DACS Sun SPARC/SunOS to
MIPS R3000 Bare Instruction Set Architecture Simulator,
Version 4.7.1

Prepared By:
Software Standards Validation Group
Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899
U.S.A.

AVF Control Number: NIST92DDI510_5_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on November 19, 1993.

Compiler Name and Version:	DACS MIPS R3000 bare Ada Cross Compiler System, Version 4.7.1
Host Computer System:	Sun SPARCstation IPX running under SunOS, Release 4.1.3
Target Computer System:	DACS Sun SPARC/SunOS to MIPS R3000 Bare Instruction Set Architecture Simulator, Version 4.7.1

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

As a result of this validation effort, Validation Certificate 931119S1.11332 is awarded to DDC-I, Inc. This certificate expires 2 years after ANSI/MIL-STD-1815B is approved by ANSI.

This report has been reviewed and is approved.

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

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

Computer Systems Laboratory (CSL)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899
U.S.A.

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

Ada Joint Program Office
~~Mr.~~ Dirk Rogers, Major, USAF
Acting Director
Ada Joint Program Office
Washington DC 20301
U.S.A.

NIST92DDI510_5_1.11

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Customer: DDC-I, Inc.

Certificate Awardee: DDC-I, Inc.

Ada Validation Facility: National Institute of Standards and Technology
Computer Systems Laboratory (CSL)
Software Standards Validation Group
Building 225, Room A266
Gaithersburg, Maryland 20899
U.S.A.

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: DACS MIPS R3000 bare Ada Cross Compiler System, Version 4.7.1

Host Computer System: Sun SPARCstation IPX running under SunOS, Release 4.1.3

Target Computer System: DACS Sun SPARC/SunOS to MIPS R3000 Bare Instruction Set Architecture Simulator, Version 4.7.1

Declaration:

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

Paul M. [signature] 11-19-93

Customer Signature — Date
Company DDC-I, Inc.
Title President

Paul M. [signature] 11-19-93

Certificate Awardee Signature — Date
Company DDC-I, Inc.
Title President

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro92] 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 [Pro92]. 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, Virginia 22161
U.S.A.

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, Virginia 22311-1772
U.S.A.

1.2 REFERENCES

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

[Pro92] Ada Compiler Validation Procedures, Version 3.1, Ada Joint Program Office, August 1992.

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

1.3 ACVC TEST CLASSES

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

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

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

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

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

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 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 Office (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 [Pro92].
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 95 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 91-08-02.

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

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. 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 test) 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 C86006H check for the predefined type SHORT_FLOAT; for this implementation, there is no such type.

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

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

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS 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_OVERFLOWS 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 creates a dependence on generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3.)

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.

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

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

CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A use an illegal file name in an attempt to create a file and expect NAME_ERROR to be raised; this implementation does not support external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 71 tests.

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

B22003A	B26001A	B26002A	B26005A	B28003A	B29001A	B33301B
B35101A	B37106A	B37301B	B37302A	B38003A	B38003B	B38009A
B38009B	B55A01A	B61001C	B61001F	B61001H	B61001I	B61001M
B61001R	B61001W	B67001H	B83A07A	B83A07B	B83A07C	B83E01C
B83E01D	B83E01E	B85001D	B85008D	B91001A	B91002A	B91002B
B91002C	B91002D	B91002E	B91002F	B91002G	B91002H	B91002I
B91002J	B91002K	B91002L	B95030A	B95061A	B95061F	B95061G
B95077A	B97103E	B97104G	BA1001A	BA1101B	BC1109A	BC1109C
BC1109D	BC1202A	BC1202F	BC1202G	BE2210A	BE2413A	

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.

CA2009C and CA2009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the compilation of that unit's body; as allowed by AI-00408 and AI-00506, the compilation of the generic unit bodies makes the compilation unit that contains the instantiations obsolete.

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 adequately by the information given in the initial pages of this report.

For technical information about this Ada implementation, contact:

Forrest Holemon
410 North 44th Street, Suite 320
Phoenix, Arizona 85008 (U.S.A.)
Telephone: 602-275-7172
Telefax: 602-275-7502

For sales information about this Ada implementation, contact:

Mike Halpin
410 North 44th Street, Suite 320
Phoenix, Arizona 85008 (U.S.A.)
Telephone: 602-275-7172
Telefax: 602-275-7502

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 [Pro92].

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	3526	
b)	Total Number of Withdrawn Tests	95	
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.

The DDC-I Symbolic Debugger System (SDS) runs under Sun SPARCstation IPX. The DDC-I SDS contains the MIPS R3000 Bare Instruction Set Architecture Simulator, Version 4.7.1, and is used to download and schedule the target programs.

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 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:

-library -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#0.1111111111111111111111#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 ("B28006E1.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	: 2_147_483_648
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#FFFFFFFE#
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#800F0000#-2**32
VARIABLE_ADDRESS2	: 16#80100000#-2**32
YOUR_PRAGMA	: EXPORT_OBJECT

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 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 Unix assembler *as*(1) 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 Unix command:

% **adamips** {*option*} *source-file-name* {*source-file-name*}

4.1.1. Parameters and Options

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

source-file-name

This parameter specifies the file containing the source text to be compiled. Any valid Unix file name may be used.

If the file name specified does not have a suffix, then the suffix **.ada** is assumed.

More than one file name can be specified. Each *source-file-name* may contain pattern matching characters as defined by the shell (such as "*" and "?"). The compilation starts with the leftmost file name from the command are compiled in alphanumeric order. If any file name occurs more than once in this process, then it is compiled more than once.

The format of the source text is described in Section 4.2.1.

-list

The user may request a source listing by means of this option. The source listing is written to the list file. Section 4.3.1 contains a description of the source listing.

If the option is not present, no source listing is produced, regardless of any use of pragma LIST in the program or of any diagnostic messages produced.

In addition, this option provides generated assembly listings for each compilation unit in the source file. Section 4.3.3 contains a description of the generated assembly listing.

-long_error_messages

This option specifies that diagnostic messages in the listing file and in the error file are extended with:

- a more elaboration description of the error
- recommended user action
- one or more references to the Ada Reference Manual

A complete list of all diagnostic messages can be found in the *Diagnostics Guide*. (Note that the compiler issues a few messages related to representation clauses and implementation-dependent pragmas that may not appear in the *Diagnostics Guide*.)

-nowarnings

This option controls whether compiler warning messages are suppressed or not. By default, they are not suppressed.

-xref

A cross-reference listing can be requested by the user by means of this option. If it is present 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 4.3.1.3.

-library *file-name*

This option 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 option is omitted, the sublibrary designated by the environment variable **ADAMIPS_LIBRARY** is used as the current sublibrary.

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

-configuration_file *file-name*

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

If the option is omitted, the configuration file designated by the file name **$release/compiler/config** is used by default. Section 4.2.2 contains a description of the configuration file.

-nocheck *check_kind*, {*check_kind*}

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

By default, all run time checks will be generated by the compiler.

When the **-nocheck** option is used, 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*]. There is no default kind for -nocheck; to suppress all checks, specify **-nocheck all**.

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

-gisa

Use of this option 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).

-nosave_source

By default, 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).

By using the -nosave_source option, this saving of the source text will not occur. While this will reduce somewhat the space needed by the program library, it will also prevent automatic recompilation by the Ada Recompiler, and hence is not recommended for normal use.

-keep_assembly

When this option 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 option does not affect the production of generated assembly listings.

-progress

When this option is given, the compiler will write a message to the standard output as each pass of the compiler starts to run, as well as the name and type of each compilation unit. This information is not provided by default.

-debug limit_optimizations | full_optimizations

When this option 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 DACS Unix to MIPS R3000 Bare Symbolic Cross Debugging System.

If **-debug full_optimizations** is specified, the compiler will generate code with all optimizations enabled. This code will be the same object code as if the option had not been specified at all (though there may be some minor differences in the generated assembly code, due to some extra labels being present). However, this full level of optimization 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; not storing the values of Ada variables to memory across statement boundaries; the elimination of unnecessary library package elaboration routines; and the transformation of certain kinds of tasks into more efficient "monitor tasks". Users may also wish to specify this option to make the generated machine code more understandable relative to the Ada source code.

The remaining options pertain to the various optimizing components of the compiler. By default, the compiler operates with all optimizations turned on. The principal reason why users might want to turn off some optimizations is covered by the **-debug** option described above, and that option should be used accordingly.

The options described below directly turn off particular optimizing components, and should only be used to circumvent the capacity or other problems described below.

-nofeoptimize

This pertains to the "front end" optimizer. This sometimes 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

```
*** 1562S-0: Optimizer capacity exceeded. Too many names in a basic block.
```

then use of this option will bypass this optimizer and allow the compilation to finish normally.

-nogoptimize

This pertains to the "g-code" (intermediate language) optimizer. This optimizer presents no special capacity or other problems, so use of this option is unlikely to be necessary.

-nobeoptimize

This pertains to the "back end" optimizer. This optimizer is the most powerful in the compiler, and accordingly uses a fairly large amount of host resources, in both CPU time and virtual memory. If such resource utilization is causing a problem or is undesired, then this option may be used.

Examples of option usage

```
% adamips navigation_constants

% adamips -list -long -xref event_scheduler.a

% adamips -prog -lib test_versions.alb -debug limit /usr1/source/altitudes_b
```

[*remainder of chapter deleted*]

LINKER OPTIONS

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

Chapter 5
The Ada Linker

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

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 DACS Unix to MIPS R3000 Bare Cross 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, options 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 *DACS Unix to MIPS R3000 Bare Ada Run-Time System User's Guide*. This facility may also be used to modify or add to the standard DACS Unix to MIPS R3000 Bare Cross 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 *DACS Unix to MIPS R3000 Bare Cross Linker Reference Manual*.

[*portion of chapter deleted*]

5.1. The Invocation Command

The Ada Linker is invoked by submitting the following Unix command:

% **adamips.link** {*option*} *main-program-name* {*main_program_name*}

As part of the "MIPS part" of an Ada link, a temporary subdirectory is created in **/tmp** (unless the **-stop** option has been used, in which case it is created below the current directory). Use of this subdirectory, the name of which is constructed from the Unix process-id, permits concurrent linking in the same current directory. The subdirectory contains work files only, and it and its contents are deleted at the end of the link.

5.1.1. Parameters and Options

Default values exist for all options as indicated below. All option 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 Unix 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 spaces. The first name supplied is the one used for the file name in Ada link output.

The first three of the options below pertain to the "Ada part" of the Ada link. The remaining options pertain to the "MIPS part" of the link.

-log *file-name*

This option specifies whether a log file is to be produced during the linking. By default no log file is produced.

The contents of the log file are described in Section 5.3.

-warnings

This option specifies whether warnings are output if detected by the linker. Warnings can be generated when conflicts between target program attributes and the specified qualifiers are found, and when a package has an inconsistent body. By default warnings are suppressed.

-library *file-name*

This option 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 option is omitted, the sublibrary designated by the environment variable **ADAMIPS_LIBRARY** is used as current sublibrary.

-mp

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

-options "*symbol-name=value{,symbol-name=value}*"

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

The option specifies a quoted string, 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 option 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 Assembler preprocessor directives specifying the overrides, and then defining that file with the environment variable **adamips_rte_opts**.

A full list of the names of these special symbols, their default values, and the runtime behavior that they control, is given in the *Ada Run-Time System User's Guide*, as are the details of the alternative method.

A few of the most commonly used of these symbols are listed here, for the sake of convenience.

Most Commonly Used Run-Time Override Symbols		
Function	**Symbol**	**Default**
Program Heap Size	rtheapsz1=	102400 (100K)
Main Procedure Stack Size	rtmstacksz1=	102400 (100K)
Default Task Stack Size	rttstacksz1=	8192 (8K)
Maximum Number of Concurrent Tasks	rtttcbs=	50
Default Task Priority	rttprty1=	0

See the *Examples* section below for examples of actual usages of these symbols.

-standard_control *file-name*

This option specifies the file name of "standard" DACS Unix to MIPS R3000 Bare Cross Linker control statements that are to be used for all links for an installation or project.

-control *file-name*

This option specifies the file name of "user" DACS Unix to MIPS R3000 Bare Cross Linker control statements that are to be used for this particular link.

The files designated by the previous two options are used to form the full input control statement stream to the DACS Unix to MIPS R3000 Bare Cross Linker, in this concatenated order:

"standard" control file (if it exists)
<statements generated by the Ada part of the link>
"user" control file (if option present 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 the environment variables described above defined to files that contain default sets of standard control statements. These consist of the minimal **relocation** statements required by the DACS Unix to MIPS R3000 Bare Cross Linker, and various other necessary directives.

-user_rts *directory-list*

This option specifies a colon-separated list of directories 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 list's directory(ies) are taken ahead of those in the directories specified by the **-target_rts** option (see below) and those in the standard RTE directories (including those RTE modules in the predefined library). If the option is omitted, environment variable **adamips_user_rts** is used, if it has been defined.

-target_rts *directory-list*

This option specifies a colon-separated list of directories 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 list's directory(ies) are taken ahead of those in the standard RTE directory. If the option is omitted, environment variable **adamips_target_rts** is used, if it has been defined.

-debug

When this option 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 option. By default no such file is produced, even if some of the program units linked were compiled with a debug option.

This symbolic debug information file is used by the DACS Unix to MIPS R3000 Bare Symbolic Cross Debugging System.

The **show -invocation_command** command of Ada PLU may be used to determine what options units in the program library were compiled with.

Note that the identical executable load module is produced by the Ada Linker, whether or not this option is used.

Occasionally, some Ada language constructs may not be supported in the symbolic debug information file, meaning that certain Ada names may not be known to the Symbolic Cross Debugging System. When this happens, warnings will be issued during the Ada link, so that the user is aware of what names fall into this situation. Also, some Ada names or source lines may not be present in the symbolic debug information file, and thus will not be known to the Symbolic Cross Debugging System, because they have been "optimized away", e.g. by dead code elimination. When this happens, informational messages will be issued during the Ada link.

Examples of these warnings and informational messages:

```
*** Warning:  Renaming package BDT is not present
              in symbolic debug information file; refer to renamed package.

*** Informational: Source lines 655 thru 674 of file
    ADA$DISK:[ADAQA.TESTING.MIPS]MATH2.ADA;1 are unreachable
    and are not referenced in the symbolic debug information file.
```

-noinform

By default, the "diagnostic traces" of the Ada runtime executive are linked in and activated. These traces print out information when unusual conditions occur, such as unhandled exceptions and task deadlock. See the *Ada Run-Time System User's Guide* for full details.

By using the **-noinform** option, these diagnostic traces will not be linked in or activated.

-trace

When this option is present, the "optional traces" of the Ada runtime executive are linked in (but not activated). These traces print out information during normal program execution, to assist in debugging and in better understanding program behavior. See the *Ada Run-Time System User's Guide* for full details.

By default, the optional traces are not linked in.

-eslink_options *"DACS Unix to MIPS R3000 Bare Cross Linker options"*

This option specifies a string containing one or more command options to be passed to the execution of the DACS Unix to MIPS R3000 Bare Cross Linker.

-stop *number*

This option, when used with *number* 0, results in the Ada link stopping after the "Ada part" has done all Ada-required checking, and has created a command file (Unix Bourne shell script) (located in the temporary subdirectory) that executes the "MIPS part", but before that file has actually been invoked.

When used with *number* 1, the file is invoked, but stops before the DACS Unix to MIPS R3000 Bare Cross Linker is invoked, leaving the temporary subdirectory and its files in place. When used with *number* 2, it executes the DACS Unix to MIPS R3000 Bare Cross Linker but then stops before the symbolic debug information file is produced.

This option 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 -options "rtheapsz1=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 -options "rtheapsz1=20*1024,rtheapsz2=12*1024,rtheapsz3=50*1024" able baker char

Now, an example of introducing "user" DACS Unix to MIPS R3000 Bare Cross Linker control statements:

% adamips.link -control test_driver.ctl test_driver

where **test_driver.ctl** in the current directory contains

```
search_path is
   /dma/object
end
object_file is
   dmacheck
end
informational messages are off
```

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

% setenv adamips_target_rts "/tektronix/io/test:/tektronix/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** option, then in the two directories specified by the **adamips_target_rts** environment variable, and lastly in the standard RTE directory.

To revert to referencing only the standard RTE directory:

% unsetenv adamips_target_rts
% adamips.link flight_simulator

[*remainder of chapter deleted*]

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

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

type FLOAT is digits 6
     range -2#1.0#E128 .. 2#0.111111111111111111111#E128

type LONG_FLOAT is digits 15
   range -2#1.0#E1024 ..
    2#0.111111111111111111111111111111111111111111111111111#E024;

type DURATION is delta 2**(-14) range -131_072.0 .. 131_071.0;

end STANDARD;
```

Appendix C
Appendix F of the Ada Reference Manual

This appendix includes in its entirety Appendix F from the *DACS Unix to MIPS R3000 Bare Ada Cross Compiler System User's Guide.*

Note that the implementation-specific portions of the package STANDARD are included in this appendix, as Section F.1.

This Page Intentionally Left Blank

Appendix F
Appendix F of the Ada Reference Manual

This appendix describes all implementation-dependent characteristics of the Ada language as implemented by the DACS Unix to MIPS R3000 Bare Ada 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

No other predefined integer types (such as SHORT_INTEGER or LONG_INTEGER) are implemented, as there are no corresponding underlying machine base types.

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#0.111111111111111111111111#E128
FLOAT'MACHINE_EMAX	=	128
FLOAT'MACHINE_EMIN	=	-125
FLOAT'MACHINE_MANTISSA	=	24
FLOAT'MACHINE_OVERFLOWS	=	TRUE
FLOAT'MACHINE_RADIX	=	2
FLOAT'MACHINE_ROUNDS	=	TRUE
FLOAT'SAFE_EMAX	=	125
FLOAT'SAFE_LARGE	=	2#0.111111111111111111111111#E125
FLOAT'SAFE_SMALL	=	2#0.1#E-125
FLOAT'SIZE	=	32

LONG_FLOAT'DIGITS	=	15
LONG_FLOAT'FIRST	=	-2#1.0#E1024
LONG_FLOAT'LAST	=	2#0.11111111111111111111111111111111111111111111111111111#E1024
LONG_FLOAT'MACHINE_EMAX	=	1024
LONG_FLOAT'MACHINE_EMIN	=	-1021
LONG_FLOAT'MACHINE_MANTISSA	=	53
LONG_FLOAT'MACHINE_OVERFLOWS	=	TRUE
LONG_FLOAT'MACHINE_RADIX	=	2
LONG_FLOAT'MACHINE_ROUNDS	=	TRUE
LONG_FLOAT'SAFE_EMAX	=	1023
LONG_FLOAT'SAFE_LARGE	=	2#0.11111111111111111111111111111111111111111111111111111#E1023
LONG_FLOAT'SAFE_SMALL	=	2#0.1#E-1023
LONG_FLOAT'SIZE	=	64

No other predefined floating point types (such as SHORT_FLOAT) are implemented, as there are no corresponding underlying machine base types.

F.1.3. Fixed Point Types

One kind of anonymous predefined fixed point type is implemented, *fixed* (which is not defined in package STANDARD, but is used here only for reference), as well as the predefined type DURATION.

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

For *fixed* there is a virtual predefined type for each possible value of *small* [*Ada RM 3.5.9*]. The possible values of *small* are the powers of two that are representable by a LONG_FLOAT value, unless a length clause specifying T'SMALL is given, in which case the specified value is used.

The lower and upper bounds of these types are:

lower bound of *fixed* types	=	-2_147_483_648 * *small*
upper bound of *fixed* types	=	2_147_483_647 * *small*

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

Any fixed point type T has the following attributes:

T'MACHINE_OVERFLOWS	=	TRUE
T'MACHINE_ROUNDS	=	TRUE

Type DURATION

The predefined fixed point type DURATION has the following attributes:

DURATION'AFT	=	5
DURATION'DELTA	=	DURATION'SMALL
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, excep. .n 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 DACS Unix to MIPS R3000 Bare Assembler (VAX/VMS and Sun SPARC/SunOS hosts) or the Unix assembler *as*(1) (MIPS RISC/os or DECStation/ULTRIX hosts; for the latter, assemblies must be done using the **-EB** option; otherwise, object code will be produced according to the host's little-endianism).

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** option, see Section 5.1), so that the DACS Unix to MIPS R3000 Bare Cross Linker can find it.

Languages C, C++, Fortran, and Pascal

It is possible to use pragma INTERFACE to call subprograms written in these other languages supported by MIPS Technologies, Inc. derived compilers. (These are the compilers licensed by MIPS for their RISC/os systems, by Silicon Graphics for their IRIX systems, by Digital for their DECStation ULTRIX systems, etc.) 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. (Note however that special data mapping is done peculiar to the other languages, e.g. it is the user's responsibility to null-terminate Ada strings when passing them to C, to reconcile Ada versus Fortran array layouts, etc.)

To do this for VAX/VMS or Sun SPARC/SunOS hosts, compile such subprograms on a MIPS derived 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 or Sun SPARC/SunOS. (Make sure the transfer preserves the binary nature of the files.) Then proceed as with assembly language subprograms.

To do this for MIPS RISC/os or DECStation/ULTRIX hosts, compile such subprograms using the normal Unix compile command (cc(1), etc.). Note that if the host system is DECStation/ULTRIX, compiles must be done using the -EB option; otherwise, object code will be produced according to the host's little-endianism.

Note that C++ is not a valid language name to pragma INTERFACE; use C instead.

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 *DACS Unix to MIPS R3000 Bare Ada Run-Time System User'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 *DACS Unix to MIPS R3000 Bare Ada Run-Time System User'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 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).

Objects which are allocated indirectly by the compiler (such as dynamically-sized arrays and renames of dynamically-addressed objects) must be so interpreted by non-Ada routines.

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 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 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 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

F.4.1. X'PASSED_BY_REFERENCE

For a prefix X that denotes a formal parameter (of either a subprogram or an entry) or any type, this attribute yields the value TRUE if the formal parameter is (or would be, in the case of a type, assuming a formal parameter of that type) passed by reference; it yields the value FALSE otherwise, that is, when the formal parameter is (would be) passed by copy-in/copy-back [*Ada RM 6.2* (*6-8*)]. The value of this attribute is of the predefined type BOOLEAN.

Examples of the use of this attribute:

```
type SOME_TYPE is ...

B : BOOLEAN := SOME_TYPE'PASSED_BY_REFERENCE;

...

accept E (PARAM : SOME_TYPE) do
   if PARAM'PASSED_BY_REFERENCE then
      ...
   else
      ...
   end if;
end E;
```

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 (Mips);

   SYSTEM_NAME           : constant NAME := Mips;

   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;
   IBE                   : constant :=  6 * 2**2;
   DBE                   : constant :=  7 * 2**2;
   Sys                   : constant :=  8 * 2**2;
   Bp                    : constant :=  9 * 2**2;
   RI                    : constant := 10 * 2**2;
   CpU                   : constant := 11 * 2**2;
   Ovf                   : constant := 12 * 2**2;
   Reserved13            : constant := 13 * 2**2;
   Reserved14            : constant := 14 * 2**2;
   Reserved15            : constant := 15 * 2**2;
   SW0                   : constant := 2**0 * 2**8;
   SW1                   : 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 := IPO + 1 + 13;
    GISA14              : constant := IPO + 1 + 14;
    GISA15              : constant := IPO + 1 + 15;
    GISA16              : constant := IPO + 1 + 16;
    GISA17              : constant := IPO + 1 + 17;
    GISA18              : constant := IPO + 1 + 18;
    GISA19              : constant := IPO + 1 + 19;
    GISA20              : constant := IPO + 1 + 20;
    GISA21              : constant := IPO + 1 + 21;
    GISA22              : constant := IPO + 1 + 22;
    GISA23              : constant := IPO + 1 + 23;
    GISA24              : constant := IPO + 1 + 24;
    GISA25              : constant := IPO + 1 + 25;
    GISA26              : constant := IPO + 1 + 26;
    GISA27              : constant := IPO + 1 + 27;
    GISA28              : constant := IPO + 1 + 28;
    GISA29              : constant := IPO + 1 + 29;
    GISA30              : constant := IPO + 1 + 30;
    GISA31              : constant := IPO + 1 + 31;

end SYSTEM;
```

Note that since timers are not part of the MIPS R3000 architecture specification, different MIPS 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 that granularity, corresponds to the MIPS R3000 target implementation that the DACS Unix to MIPS R3000 Bare Ada Cross Compiler System is validated upon. It also is the most common value for the different MIPS 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 R3000 target implementations, see the *DACS Unix to MIPS R3000 Bare Ada Run-Time System User'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 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: T'STORAGE_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 *DACS Unix to MIPS R3000 Bare Ada Run-Time System User's Guide* for full details on how the storage in collections is managed.

Specification of storage for a task activation: T'STORAGE_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 *DACS Unix to MIPS R3000 Bare Ada Run-Time System User'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: T'SMALL

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. However, if the type is used as the component type of an array type, then the only permitted value is 1.

In terms of allowable component clauses, record components fall into three classes, depending on their type:

- integer, enumeration, and fixed point types whose *minimal size* (see Section F.6.1) is less than 32 bits;
- statically-bounded array types declared with pragma PACK, and record types declared with a record representation clause;
- all others.

Components of the "less-than-32-bit integer/enumeration/fixed" 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, and less than or equal to 32 bits. Furthermore, if the storage place is less than 32 bits, the component may cross a word boundary.

Components of the "packed array/record rep clause" class may be given a component clause that specifies a storage place at any bit offset, if the size of the array or record is less than a word, or at a storage unit offset otherwise. The size of the storage place must be the same as the minimal size of the array or record type. Note that the component clause for an array or record component type cannot specify a representation different from that of the component's type.

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

If a component clause is used for a discriminant, that discriminant must be the only discriminant of the record type.

An example of the rule regarding array and record component types:

```
type SMALL_INT is range 0..15;

type INNER_REC is record
   A : SMALL_INT;
   B : SMALL_INT;
end record;

type BOOL_ARR is array (1..8) of BOOLEAN;

type REC_ILLEGAL is record
   IR : INNER_REC;
   BA : BOOL_ARR;
end record;
for REC_ILLEGAL use record
   IR at 0 range 0..7;    -- illegal, not enough storage space
   BA at 0 range 8..15;   -- illegal, not enough storage space
end record;

type INNER_REC_R is new INNER_REC;
for INNER_REC_R use record
   A at 0 range 0..3;
   B at 0 range 4..7;
end record;

type BOOL_ARR_P is new BOOL_ARR;
pragma PACK (BOOL_ARR_P);

type REC_LEGAL is record
   IR : INNER_REC_R;
   BA : BOOL_ARR_P;
end record;
for REC_LEGAL use record
   IR at 0 range 0..7;    -- OK, now that component type is packed
   BA at 0 range 8..15;   -- OK, now that component type has rep. clause
end record;
```

Component clauses do not have to be in storage order, and there may be gaps in storage between component clauses. No other components are allocated in such gaps.

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 *DACS Unix to MIPS R3000 Bare Cross 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.

Because the value of a variable with an address clause must also be stored in memory, rather than kept in a register, compilations of source units containing references to address clause variables are done with less optimizations than normal. The compiler issues a warning message when this happens. The user may want to isolate such references into small, separately compiled units, to limit the effect of this consequence.

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 option is present, 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 *DACS Unix to MIPS R3000 Bare Ada Run-Time System User'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 *DACS Unix to MIPS R3000 Bare Ada Run-Time System User'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 readjust 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
BI : BITS_8;      -- actual data is leftmost byte in object's word
  . . .
SM := UC (BI);   -- actual data is moved from leftmost to rightmost byte as part of conversion
```

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

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

F.10. Other Chapter 13 Areas

F.10.1. Change of Representation

Change of representation is allowed and supported by the compiler.

F.10.2. Representation Attributes

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

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

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

The X'SIZE attribute, when applied to a type, returns the *minimal 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 >= 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 R3000 target implementation. Depending on the implementation, this may be a console, a workstation disk drive, simulator files, etc. See the *DACS Unix to MIPS R3000 Bare Ada Run-Time System User'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 4.2.2);

- 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. These are described in the *DACS Unix to MIPS R3000 Bare Ada Run-Time System User's Guide*.

- 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.