

REPORT DOCUMENTATION PAGE

Form Approved
OPM No.

AD-A273 948



d to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering formation. Send comments regarding this burden estimate or any other aspect of this collection of information, including Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA, Office of Management and Budget, Washington, DC 20503.

REPORT

3. REPORT TYPE AND DATES

4. TITLE AND

RISCAE TRW RH32-targeted Ada Compiler, 1.0, Host: DEC VAXstation 4000, Target: RISCAE TRW RH32 Simulator running on the host 930901W1.11321

5. FUNDING

(2)

6.

Authors:

Wright-Patterson AFB

7. PERFORMING ORGANIZATION NAME(S) AND

Ada Validating Facility, Language Control Facility ASD/SCEL Bldg. 676, Room 135
Wright Patterson AFB, Dayton OH 45433

8. PERFORMING ORGANIZATION

9. SPONSORING/MONITORING AGENCY NAME(S) AND

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

DTIC
ELECTE
DEC 14 1993
S A

10. SPONSORING/MONITORING AGENCY

11. SUPPLEMENTARY

12a. DISTRIBUTION/AVAILABILITY

Approved for public release; distribution unlimited

12b. DISTRIBUTION

13. (Maximum 200

RISCAE TRW RH32-targeted Ada Compiler, 1.0, Host: DEC VAXstation 4000, Target: RISCAE TRW RH32 Simulator running on the host, ACVC 1.11

14. SUBJECT

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

15. NUMBER OF

16. PRICE

17. SECURITY CLASSIFICATION
UNCLASSIFIED

18. SECURITY CLASSIFICATION
UNCLASSIFIED

19. SECURITY CLASSIFICATION
UNCLASSIFIED

20. LIMITATION OF
UNCLASSIFIED

NSN

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

AVF Control Number: AVF-VSR-568.0893
Date VSR Completed: 16 September 1993
93-07-12-INT

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 930901W1.11321
Intermetrics Inc.
RISCAE TRW RH32-targeted Ada Compiler, 1.0
DEC VAXstation 4000 under VMS, 5.5 =>
RISCAE TRW RH32 Simulator running on the host under VMS, 5.5

(Final)

Prepared By:
Ada Validation Facility
645 C-CSG/SCSL
Wright-Patterson AFB OH 45433-6503

93-30268



4086

93 12 13 038

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 1 September 1993.

Compiler Name and Version: RISCAE TRW RH32-targeted Ada Compiler, 1.0

Host Computer System: DEC VAXstation 4000
under VMS, 5.5

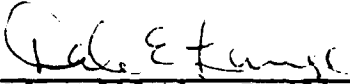
Target Computer System: RISCAE TRW RH32 Simulator running on the host
under VMS, 5.5

Customer Agreement Number: 93-07-12-INT

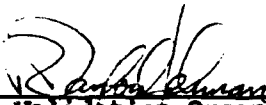
See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 930901W.11321 is awarded to Intermetrics Inc. This certificate expires two years after MIL-STD-1815B is approved by ANSI.

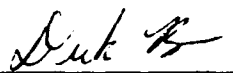
This report has been reviewed and is approved.



Ada Validation Facility
Dale E. Lange
Technical Director
645 CCSG/SCSL
Wright-Patterson AFB OH 45433-6503



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



Ada Joint Program Office
M. Dirk Rogers, Major, USAF
Acting Director
Department of Defense
Washington DC 20301

Accession For	
NTIS	CRAE ☒
DTIC	TAB ☐
Unannounced	☐
Justification	☐
By	
Distribution/	
Availability Code	
Dist	Availability or Special
A-1	

DTIC QUALITY INSPECTED 3

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: Intermetrics Inc.

Ada Validation Facility: 645 C-CSG/SCSL
Wright-Patterson AFB OH 45433-6503

ACVC Version: 1.11

Ada Implementation:

Ada Compiler Name and Version: RISCAL TRW RH32-targeted Ada Compiler, 1.0

Host Computer System: DEC VAXstation 4000
under VMS, 5.5

Target Computer System: RISCAL TRW RH32 Simulator running on the host
under VMS, 5.5

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.


Customer Signature

8/31/93
Date

TABLE OF CONTENTS

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

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

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

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

INTRODUCTION

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint
Program Office, August 1990.
- [UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

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

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

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

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

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

INTRODUCTION

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 removing some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

INTRODUCTION

Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 2 August 1991.

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

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

IMPLEMENTATION DEPENDENCIES

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)

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`; for this implementation, there is no such type.

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

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

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

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

D64005G uses 17 levels of recursive procedure calls nesting; this level of nesting for procedure calls exceeds the capacity of the compiler.

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

CA2009C and CA2009F check whether a generic unit can be instantiated before 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.

IMPLEMENTATION DEPENDENCIES

AE2101C and EE2201D..E (2 tests) use instantiations of package SEQUENTIAL IO with unconstrained array types and record types with discriminants without defaults; these instantiations are rejected by this compiler.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT IO with unconstrained array types and record types with discriminants without defaults; these instantiations are rejected by this compiler.

The following 260 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)	CE2201F..N (9)	CE2203A	CE2204A..D (4)
CE2205A	CE2206A	CE2208B	CE2401A..C (3)
CE2401E..F (2)	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.)

IMPLEMENTATION DEPENDENCIES

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 10 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 B83033B B85013D

CA2009C and CA2009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the separate compilation of that unit's body; as allowed by AI-257, this implementation requires that the bodies of a generic unit be in the same compilation if instantiations of that unit precede the bodies. The implementation issues error messages at link time that the main program "has unresolved generic instantiations" and the tests cannot be executed.

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 and sales information about this Ada implementation, contact:

Mike Ryer
Intermetrics Inc.
733 Concord Avenue
Cambridge MA 02138-1002
(617) 661-1840

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system — if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b

PROCESSING INFORMATION

and f, below).

a) Total Number of Applicable Tests	3575
b) Total Number of Withdrawn Tests	95
c) Processed Inapplicable Tests	39
d) Non-Processed I/O Tests	260
e) Non-Processed Floating-Point Precision Tests	201
f) Total Number of Inapplicable Tests	500 (c+d+e)
g) Total Number of Tests for ACVC 1.11	4170 (a+b+f)

3.3 TEST EXECUTION

A TK-50 cartridge tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the TK-50 cartridge tape were loaded directly onto the host computer.

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 computer system, as appropriate, and run. The results were captured on the 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:

Option/Switch	Effect
Compiler Options	
/LIST	Generate a compilation listing with default listing format. Used for E tests.
-lc	Generate a compilation listing with continuous listing format. Used for all other tests.
Program Builder options	
/PASS=llink_options	Pass options to the linker locator.
Linker/Locator options	
/LL=adabase:[lib]trwmpk.lbl	Name of library index file to be used for unresolved references.
/C=adabase:[lib]trwmpk.llc	Name of file containing locator commands.

PROCESSING INFORMATION

/SE Eliminate unreferenced segments.

/US=("atat001","rtl_stack","address_space_descriptor")
Mark indicated symbols as referenced before
determining unreferenced segments.

Test output, compiler and linker listings, and job logs were captured on TK-50 cartridge 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	200 — Value of V
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	"" & (1..V/2 => 'A') & ""
\$BIG_STRING2	"" & (1..V-1-V/2 => 'A') & '1' & ""
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"

MACRO PARAMETERS

\$MAX_STRING_LITERAL ' ' & (1..V-2 => 'A') & ' '

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

Macro Parameter	Macro Value
\$ACC_SIZE	32
\$ALIGNMENT	4
\$COUNT_LAST	2_147_483_647
\$DEFAULT_MEM_SIZE	2_147_483_648
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	TRW_RH32
\$DELTA_DOC	2.0**(-31)
\$ENTRY_ADDRESS	16#1#
\$ENTRY_ADDRESS1	16#2#
\$ENTRY_ADDRESS2	16#3#
\$FIELD_LAST	2_147_483_647
\$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	90_000.0
\$GREATER_THAN_DURATION BASE LAST	10_000_000.0
\$GREATER_THAN_FLOAT_BASE LAST	3.41E+38
\$GREATER_THAN_FLOAT_SAFE LARGE	1.0E38

MACRO PARAMETERS

\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
 1.0E38
 \$HIGH_PRIORITY 31
 \$ILLEGAL_EXTERNAL_FILE_NAME1
 NO_FILES_AT_ALL_1
 \$ILLEGAL_EXTERNAL_FILE_NAME2
 NO_FILES_AT_ALL_2
 \$INAPPROPRIATE_LINE_LENGTH
 -1
 \$INAPPROPRIATE_PAGE_LENGTH
 -1
 \$INCLUDE_PRAGMA1 PRAGMA INCLUDE ("A28006D1.ADT")
 \$INCLUDE_PRAGMA2 PRAGMA INCLUDE ("B28006F1.ADT")
 \$INTEGER_FIRST -2147483648
 \$INTEGER_LAST 2147483647
 \$INTEGER_LAST_PLUS_1 2147483648
 \$INTERFACE_LANGUAGE ASSEMBLY
 \$LESS_THAN_DURATION -90_000.0
 \$LESS_THAN_DURATION_BASE_FIRST
 -10000000.0
 \$LINE_TERMINATOR ASCII.LF
 \$LOW_PRIORITY 1
 \$MACHINE_CODE_STATEMENT
 Format_K_R'(T_JR,true,R3);
 \$MACHINE_CODE_TYPE Format_R_Lit
 \$MANTISSA_DOC 31
 \$MAX_DIGITS 15
 \$MAX_INT 2147483647
 \$MAX_INT_PLUS_1 2_147_483_648
 \$MIN_INT -2147483648
 \$NAME BYTE_INTEGER

MACRO PARAMETERS

\$NAME_LIST	HNW_RH32,TRW_RH32
\$NAME_SPECIFICATION1	NO_FILES_1
\$NAME_SPECIFICATION2	NO_FILES_2
\$NAME_SPECIFICATION3	NO_FILES_3
\$NEG_BASED_INT	16#F000000E#
\$NEW_MEM_SIZE	2147483648
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	HNW_RH32
\$PAGE_TERMINATOR	ASCII.FF
\$RECORD_DEFINITION	RECORD MNEMONIC:MNEMONIC ENUM; REG:REGISTER_ENUM; END RECORD;
\$RECORD_NAME	FORMAT_R
\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	2048
\$TICK	2.0*(-14)
\$VARIABLE_ADDRESS	16#3FFD0#
\$VARIABLE_ADDRESS1	16#3FFF4#
\$VARIABLE_ADDRESS2	16#3FFF8#
\$YOUR_PRAGMA	APART

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.

Inputs

.....

Invocation

adatrw [option...]file.ada

Options

/debug Generate debugging output. The **/debug** option causes the compiler to generate the appropriate code and data for operation with the RISCAC Debugger.

/error_log Generate error log file. The **/error_log** option causes the compiler to generate a log file containing all the error messages and warning messages produced during compilation. The error log file has the same name as the source file, with the extension **.err**. For example, the error log file for **simple.ada** is **simple.err**. The error log file is placed in the current working directory. In the absence of the **/list** option, the error log information is sent to the standard output stream.

/num_checks_suppress Suppress numeric checking. The **/num_checks_suppress** option suppresses two kinds of numeric checks for the entire compilation:

1. **division_check**
2. **overflow_check**

These checks are described in section 11.7 of the LRM. Using **/num_checks_suppress** reduces the size of the code. Note that there is a related **adatrw** option, **/all_checks_suppress** to suppress all checks for a compilation.

/all_checks_suppress Suppress all checks. The **/all_checks_suppress** option suppresses all automatic checking, including numeric checking. This option is equivalent to using **pragma suppress** on all checks. This option reduces the size of the code, and is good for producing "production quality" code or for benchmarking the compiler. Note that there is a related **adatrw** option, **/num_checks_suppress** to suppress only certain kinds of numeric checks.

/warning_suppress

Suppress warning messages. With this option, the compiler does not print warning messages about ignored pragmas, exceptions that are certain to be raised at run-time, or other potential problems that the compiler is otherwise forbidden to deem as errors by the LRM.

/no_delete

Keep internal form file. This option is for use by compiler maintainers. Without this option, the compiler deletes internal form files following code generation.

/list

Generate listing file. The **/list** option causes the compiler to create a listing. The formats of and options for listings are discussed in section 3.2.1.7. The default listing file generated has the same name as the source file, with the extension **.lst**. For example, the default listing file produced for **simple.ada** has the name **simple.lst**. The listing file is placed in the current working directory. In order to generate a listing in the continuous listing format, use the **-lc** switch rather than the

/list option. Note: **/list** also causes an error log file to be produced, as with the **/error_log** option.

/library

Default: **ada.lib**

Use alternate library. The **/library** option specifies an alternative name for the program library.

COMPILATION SYSTEM OPTIONS

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.

Inputs

.....

Invocation

BAMP <i>{option ...}</i> <i>{main-procedure-name}</i>
--

Options

/compiler=compiler_name

Default: ada.lib

Use alternate compiler. The **/compiler** option specifies the complete (non relative) directory path to the RISCAEAda compiler. This option overrides the compiler program name stored in the program library. The **/compiler** option is intended for use by system maintainers.

/main_suppress

Suppress main program generation step. The **/main_suppress** option suppresses the creation and additional code generation steps for the temporary main program file. The **/main_suppress** option can be used when a simple change has been made to the body of a compilation unit. If unit elaboration order is changed, or if the specification of a unit is changed, or if new units are added, then this option should not be used. The **/main_suppress** option saves a few seconds, but places an additional bookkeeping burden on you. The option should be avoided under most circumstances. Note that invoking **bamp** with the **/load_suppress** option followed by another invocation of **bamp** with the **/main_suppress** option has the same effect as an invocation of **bamp** with neither option (**/load_suppress** and **/main_suppress** neutralize each other).

/library=library-name

Default: ada.lib

Use alternate library. The **/library** option specifies the name of the program library to be consulted by the **bamp** program. This option overrides the default library name.

/load_suppress

No link. The option suppresses actual object file linkage, but creates and performs code generation on the main program file. Note that invoking **bamp** with the option followed by another invocation of **bamp** with the **/main_suppress** option has the same effect as an invocation of **bamp** with neither option. That is, **/load_suppress** and **/main_suppress** neutralize each other.

/display_commands

No operations. The **/display_commands** option causes the **bamp** command to do a "dry run": it prints out the actions it takes to generate the executable program, but does not actually perform those actions. The same kind of information is printed by the **/print_operations** option.

/output=output-file-name

Use alternate executable file output name. The **/output** option specifies the name of the executable program file written by the **bamp** command. This option overrides the default output file name which is the main procedure name concatenated to 8 characters with file extension **.ab**.

/pass=link-options

Pass options to the Linking Locator. The **/pass** option specifies the Linking Locator options which are passed directly to the Linking Locator.

/print_operations

Print operations. The **/print_operations** option causes the **bamp** command to print out the actions it takes to generate the executable program as the actions are performed.

/verbose

Link verbosely. The **/verbose** option causes the **bamp** command to print out information about what actions it takes in building the main program such as:

- The name of the program library consulted.
- The library search order (listed as "uses" of the library units used by the program).
- The name of the main program file created (as opposed to the main procedure name).
- The elaboration order.
- The name of the executable load module created.

Inputs

.....

Invocation

LLINK [*PROG1*.[*OL LM ROM*]] [...] [*options*]

Linker Options

- /LIB=(lib[,lib2,...]
)** Name library index files to be searched for unresolved externals. If the index file indicates that a given external can be resolved by reading a particular module, that module is included in the link. The *Librarian* section explains how library files are built and managed. If a module name in the library index file is not a full pathname, *llink* searches for the module in the directory containing the index file.
- /LL=(ifn[,ifn2,...])** Read library index to be searched from file *ifn*. Index file *ifn* lists all libraries that would be specified on the command line if the **/LIB** switch were used.
Note: The linker portion of the linking locator may not always search the libraries in the order given. See the *Library Searches* subsection for more details.
- /SE** Eliminate unreferenced segments in object modules during linking. This option has no effect unless the **/US** option is used.
- /US=("sym","sym
2"...)** Mark the specified symbols as referenced before determining unreferenced segments. The symbols may be global symbols or segment names. This option has no effect unless the **/SE** option is used.

Locator Options

Locate processing is done by default. If the **/LO** switch is present, locate processing is not performed. When locate processing is performed, output is written to *PROG.AB* unless the **/O** switch is specified.

- /C=cfn** Read locator commands from file *cfn*.
- /LO** Suppress locate processing (link only). If no ROM processing option is specified, write output to *PROG.LN*.
- /P=n** Pad the size of all segments by *n* bytes.
- /P=n%** Pad the size of all segments to *n* percent of their original size (*n* must be > 100).

ROM Processing Options

ROM processing is performed if and only if some ROM processing option is present. If locate processing is also performed, output is written to *PROG.AB*. If only ROM processing is performed, the output is written to *PROG.RMP* by default.

/B=segname Specify the name of the segment to be created. The default name is *rom-pOutSeg*.

/RC=class1[,class2,...] Specifies that all segments of the named class(es) will be processed.

/RS=seg1[,seg2,...] Specifies that the named segment(s) will be processed.

Symbol Options

/K[="sym,sym2,..."] Keep only the named global symbols in the output module; suppress all others. If no symbols are named, suppress all global symbols.

/SP[="sym,sym2,..."] Suppress the named global symbols in the output module; keep all others. Global symbols are generated by the compiler and assembler for global variables and procedures. The compiler's rules for forming global symbol names are described in the *RISCAE Software Programmer's Manual*. Note that the names specified in */SP* and */K* must be formed via these conventions. Generally all global symbols must be retained in the output module to permit any further references to be resolved during later links. Specific global symbols may be suppressed to mask name conflicts. The switches which apply to global symbols are mutually exclusive. If no debugging is intended and the link is complete, all symbols may be stripped. Stripping symbols reduces the amount of disk space required to hold the output module and speeds up the execution of *llink* and the formatter. It does not affect the size of the user program or the download hex file generated by the formatter.

Miscellaneous Options

/0 (Zero) Displays the version number of the executable (for technical support purposes).

/I[=ifn] This switch specifies that the names of input object modules are to be taken from the file *ifn*. The input module names should be listed in the file, one per

line. The name of the first module listed will be used as a default for constructing the name of the linked output file. If *ifn* is omitted, the names of the files are read from the terminal.

/O[=ofn]

This switch specifies the name of the output file. If the switch is omitted, output will be written to *PROG.LN* or *PROG.RMP*, depending on the switches specified.

Verbose mode. Reports the following linking actions as performed:

- - The names of the object modules being read
- - The names of the library index files being searched.
- - The name of the output module.

/W

This switch inhibits warning messages. If *llink* is not performing the locate function, the "unresolved externals" warning is the only warning message that *llink* can emit. This can safely be suppressed if unresolved external references are expected. Other warning messages represent error conditions and should not in general be ignored or suppressed.

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 -2147483648 .. 2147483647;
type LONG_INTEGER is range -2147483648 .. 2147483647;
type SHORT_INTEGER is range -32768 .. 32767;
type BYTE_INTEGER is range -128 .. 127;

type FLOAT is digits 6 range -3.40282E+38 .. 3.40282E+38;
type LONG_FLOAT is digits 15 range -1.79769E+308 .. 1.79769E+308;

type DURATION is delta 2**-14 range -86400.0 .. 86400.0;
```

```
.....
end STANDARD;
```

Appendix F

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

The Ada language allows for certain machine-dependencies in a controlled manner. No machine-dependent syntax or semantic extensions or restrictions are allowed. The only allowed implementation-dependencies correspond to implementation-dependent pragmas and attributes, certain machine-dependent conventions as mentioned in Chapter 13, and certain allowed restrictions on representation clauses.

The reference manual of each Ada implementation must include an appendix (called Appendix F) that describes all implementation-dependent characteristics. The Appendix F for a given implementation must list in particular:

1. *The form, allowed places, and effect of every implementation-dependent pragma.*
2. *The name and the type of every implementation-dependent attribute.*
3. *The specification of the package SYSTEM.*
4. *The list of all restrictions on representation clauses.*
5. *The conventions used for any implementation-generated name denoting implementation-dependent components.*
6. *The interpretation of expressions that appear in address clauses, including those for interrupts.*
7. *Any restriction on unchecked conversions.*
8. *Any implementation-dependent characteristics of the input-output packages*

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

9. Any implementation-dependant characteristics of tasking.
10. Other implementation dependencies.

F.1: Pragmas

F.1.1: Predefined Language Pragmas

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

F.1.1.1: Pragmas ELABORATE, LIST, OPTIMIZE, PAGE, AND PRIORITY

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

F.1.1.2: Pragma SUPPRESS

Form: pragma SUPPRESS (identifier [, (ON =>] name));
where the identifier and name, if present, are as specified in LRM B(14). Suppression of the following run-time checks are supported:

ACCESS_CHECK

DISCRIMINANT_CHECK

INDEX_CHECK

LENGTH_CHECK
RANGE_CHECK
DIVISION_CHECK
OVERFLOW_CHECK
ELABORATION_CHECK
STORAGE_CHECK

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

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

- The suppression of DISCRIMINANT_CHECK has no effect if the pragma is not in the same declarative part as the type to which it applies.
- The suppression of ELABORATION_CHECK has no effect on a task body.
- The suppression of STORAGE_ERROR does not suppress the check that an allocator does not require more space than is available.

F.1.1.3: Pragma INLINE

Form: Pragma INLINE (*subprogram_name_comma_list*)

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

Effect: If the subprogram body has already been compiled, or is in the same compilation unit as the call, and if the subprogram does not contain nested subprograms, the code is expanded in-line at every call site and is subject to all optimizations. If the subprogram to be inlined is recursive, only the first call is inlined and the recursive call is a normal call.

Exception handlers for the INLINE subprogram are handled as for block statements.

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

F.1.1.4: Pragma INTERFACE

Form: Pragma INTERFACE (*language_name*, *subprogram_name* [, "*link_name*"])
where the *language_name* must be *assembly*, *builtin*, or *internal*, and the *subprogram_name* is as specified in the LRM B(5). The optional *link_name* parameter is a string literal specifying the entry point label of the non-Ada subprogram named in the second parameter. If *link_name* is omitted, then *link_name* defaults to the value of *subprogram_name*.

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

Effect: Specifies that a subprogram will be provided outside the Ada program library and will be callable with a specified calling interface. Neither an Ada body nor an Ada body_stub may be provided for a subprogram for which INTERFACE has been specified. *link_name* is used as the entry point label of the subprogram. The *language_name* *builtin* and *internal* are reserved for use by RISCAC compiler maintainers in run time support packages.

Use: Use with a subprogram being provided via another program language and for which no body will be given in any Ada program.

The calling conventions for an Ada program calling a pragma INTERFACE (assembly) subprogram are according to the RISCAE Run Time Model described in Appendix C of the RISCAE Software Programmer's Manual.

F.1.1.5: Pragma PACK

Form: Pragma PACK (*type_simple_name*)

Allowed Places: As specified in LRM 13.1 (12)

Effect: The effect of pragma PACK is to minimize storage consumption by discrete component types whose ranges permit packing. Refer to the RISCAE Software Programmer's Manual for more information about the effect of pragma PACK.

Use: Pragma PACK is used to reduce storage size. Size reduction usually implies an increased cost of accessing components. The decrease in storage size may be offset by increase in size of accessing code and by slowing of accessing operations.

F.1.1.6: Pragas SYSTEM, NAME, STORAGE_UNIT, MEMORY_SIZE, CONTROLLED

These pragmas are not supported and are ignored

F.1.1.7: Pragma SHARED

Form: pragma SHARED (*variable_simple_name*)
where *variable_simple_name* is of any scalar type except *long_float*.

Allowed Places: As specified in LRM B(2) : SHARED

Effect Direct reading and direct updating of the specified variable must be implemented as an indivisible operation. In addition, the implementation must ensure that each reference of the variable is made directly from/to memory (i.e. not from a temporary copy of the variable).

Use: This is used to cause every read or update of a variable to be a "synchronization" point for that variable.

F.1.2: RISCAE-Defined Pragas

F.1.2.1: F.1.2.2: Pragma APART

Form: pragma APART (*variable_name* [, *segname*])
where *variable_name* must refer to a static object, (i.e. a variable declared in a library unit package specification or body, or in a package specification or body contained in a library unit package) and *segname*, if provided, must be a string literal which specifies the name of the segment containing the object.

Allowed Places: Pragma APART is allowed in the declarative region of a library unit package specification or body, or in a package specification or body contained in a library unit package. The declaration of the variable must be in the same declarative region as the pragma and must proceed the occurrence of the pragma.

Effect: The object will be placed in a segment that is not included in group "data" and consequently, is addressed directly using an APART data base register rather than the global base register (GBR). If *segname* is provided, it is used as the name of the segment for the object. Otherwise, the object is located in segment "aidata" for initialized objects or "aadata" for uninitialized objects.

Use: The RISCAE run time model specifies that static objects and constants be addressed using base offset addressing mode with the global base register (GBR) and that there is a limit of 64K bytes of total size of such data. Pragma APART can be used to specify objects which are not to be addressed using the global base register (GBR). Pragma APART may be used if specific data items need to be located

further APART than 64K bytes or in a large program for which the total size of static objects and constants is larger than 64K bytes. Refer to the RISCAE Programmer's Reference Manual for more information about location and addressing of static objects.

F.1.2.3: Pragma INDIRECT

Form: pragma INDIRECT (*subprogram_name*)

where *subprogram_name* is the name of a subprogram which is declared in the same declarative region. A body is not allowed for a subprogram to which pragma INDIRECT applies.

Allowed Places: Pragma INDIRECT must appear within the same declarative part as the subprogram to which it applies, following the subprogram, and prior to the first use of the subprogram.

Effect: A call to a subprogram to which pragma INDIRECT applies will cause the compiler to generate a call to the address provided by the first parameter of the subprogram with parameters 2 through N of the subprogram being treated as parameters 1 through N-1. This provides the ability to save the address of a subprogram in a variable or data structure so that it may be called later.

Use: This is used in run time system code. It should not normally be used in application programs.

F.1.2.4: Pragma CONTIGUOUS

Form: pragma CONTIGUOUS (*record_type_name*)

Allowed Places: Pragma CONTIGUOUS must appear within the same declarative part as the type to which it applies, following the type declaration but prior to any forcing occurrence of the type.

Effect: Pragma CONTIGUOUS alters the layout of an Ada discriminant record type. Normally an array whose bound depends on a discriminant is mapped on to a pointer to a dynamically allocated string. This pragma forces the compiler to lay out the record and array in a single object.

Use: This is used in run time system code and should only be used as it is used in the run time. It should not normally be used in application programs.

F.2: Standard Types and Implementation-Dependant Attributes

There are no implementation-dependent attributes provided by the RISCAE Ada compiler. The following sections define the standard types supported by the RISCAE Ada compiler and the implementation-dependent values of their attributes.

F.2.1: Standard Types

The following standard types are defined for the RISCAE RH32-targeted compiler.

type byte_integer is range -128 .. 127;

type short_integer is range -32768 .. 32767;

type integer is range -2147483648 .. 2147483647;

type long_integer is range -2147483648 .. 2147483647;

type float is digits 6 range -3.40282E+38 .. 3.40282E+38;

type long_float is digits 15 range -1.79769E+308 .. 1.79769E+308;

type duration is delta 2**-14 range -86400.0 .. 86400.0;

F.2.2: Implementation-Dependent Attributes

This section describes the implementation-dependent values of the attributes of the standard types.

Type INTEGER

INTEGER'SIZE

= 32 -- bits

INTEGER'FIRST	= $-(2^{**31})$	-- -2,147,483,648
INTEGER'LAST	= $(2^{**31} - 1)$	-- 2,147,483,648
Type LONG_INTEGER		
INTEGER'SIZE		= 32 - bits
INTEGER'FIRST	= $-(2^{**31})$	-- -2,147,483,648
INTEGER'LAST	= $(2^{**31} - 1)$	-- 2,147,483,648
Type SHORT_INTEGER.		
SHORT_INTEGER'SIZE	= 16 - bits	
SHORT_INTEGER'FIRST	= $-(2^{**15})$	-- -32,768
SHORT_INTEGER'LAST	= $(2^{**15} - 1)$	-- 32,767
Type BYTE_INTEGER.		
BYTE_INTEGER'SIZE	= 8 - bits	
BYTE_INTEGER'FIRST	= $-(2^{**7})$	-- -128
BYTE_INTEGER'LAST	= $(2^{**7} - 1)$	-- 127
Type FLOAT.		
FLOAT'SIZE		= 32 -- bits.
FLOAT'DIGITS		= 6
FLOAT'MANTISSA		= 21
FLOAT'EMAX		= 84
FLOAT'EPSILON		= $2.0^{**(-20)}$
FLOAT'SMALL		= $2.0^{**(-85)}$
FLOAT'LARGE		= $(1.0 - 2.0^{**(-21)}) * 2.0^{**84}$
FLOAT'MACHINE_ROUNDS		= true
FLOAT'MACHINE_OVERFLOWS		= true
FLOAT'MACHINE_RADIX		= 2
FLOAT'MACHINE_MANTISSA		= 24
FLOAT'MACHINE_EMAX		= 128
FLOAT'MACHINE_EMIN		= -125
FLOAT'SAFE_EMAX		= 125
FLOAT'SAFE_SMALL		= $2.0^{**(-126)}$
FLOAT'SAFE_LARGE		= $(1.0 - 2.0^{**(-21)}) * 2.0^{**125}$
Type LONG_FLOAT.		
LONG_FLOAT'SIZE		= 64 -- bits.
LONG_FLOAT'DIGITS		= 15
LONG_FLOAT'MANTISSA		= 51
LONG_FLOAT'EMAX		= 204
LONG_FLOAT'EPSILON		= $2.0^{**(-50)}$
LONG_FLOAT'SMALL		= $2.0^{**(-205)}$
LONG_FLOAT'LARGE		= $(1.0 - 2.0^{**(-51)}) * 2.0^{**204}$
LONG_FLOAT'MACHINE_ROUNDS		= true
LONG_FLOAT'MACHINE_OVERFLOWS		= true
LONG_FLOAT'MACHINE_RADIX		= 2
LONG_FLOAT'MACHINE_MANTISSA		= 53
LONG_FLOAT'MACHINE_EMAX		= 1024
LONG_FLOAT'MACHINE_EMIN		= -1021
LONG_FLOAT'SAFE_EMAX		= 1021
LONG_FLOAT'SAFE_SMALL		= $2.0^{**(-1022)}$

LONG_FLOAT_SAFE_LARGE = (1.0-1.0**51)*2.0**1021

Type DURATION.

DURATION'DELTA = 2.0**(-14) -- seconds
 DURATION'FIRST = -86_400.0
 DURATION'LAST = 86_400.0
 DURATION'SMALL = 2.0**(-14)

Type PRIORITY.

PRIORITY'FIRST = 1
 PRIORITY'LAST = 31

F.3: Package SYSTEM

package SYSTEM is

type ADDRESS is new integer;

NULL_ADDRESS : constant ADDRESS := 0;

type NAME is (hnw_rh32, trw_rh32);

SYSTEM_NAME : constant NAME := trw_rh32;

STORAGE_UNIT : constant := 8;

MEMORY_SIZE : constant := 2**31; -- In storage units

-- System-Dependent Named Numbers:

MIN_INT : constant := -2147483648;

MAX_INT : constant := 2147483647;

MAX_DIGITS : constant := 15;

MAX_MANTISSA : constant := 31;

FINE_DELTA : constant := 2.0**(-31);

TICK : constant := 2.0**(-14);

-- Other System-Dependent Declarations

-- Legal values for pragma PRIORITY.

-- There are 31 user priority levels.

-- The default priority, if not assigned by pragma, is 0.

subtype PRIORITY is INTEGER range 1 .. 31;

-- NOTE: The RISCAE kernel supports higher priorities

-- under which hardware interrupts are disabled.

end SYSTEM ;

F.4: Restrictions on Representation Clauses

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

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

F.4.1: Length Clauses

Size specification: **T SIZE**.

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

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

If the length clause can not be satisfied by the type, an error message will be generated. In addition, the following restrictions apply:

access type:	the only size supported is 32.
integer, fixed point, or enumeration type:	minimum size supported is 1, the maximum size that is supported is 32, the size of the largest predefined integer type. Biased representation is not supported.
floating point type:	the sizes supported are 32 and 64. Note that the size must satisfy the DIGITS requirement. No support is provided for shortened mantissa and/or exponent lengths.

Specification of collection size: **T STORAGE_SIZE**.

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

Specification of storage for a task activation: **T STORAGE_SIZE**.

The value specified by the length clause will be the total size of the stacks (primary and secondary) allocated for the task.

Specification of small for a fixed point type : **T SMALL**.

The value of **T SMALL** is subject only to the restrictions defined in the LRM(13.2).

F.4.2: Enumeration Representation Clauses

Enumeration representation clauses are supported with the restriction that the values of the internal codes must be in the range of **MIN_INT .. MAX_INT**.

F.4.3: Record Representation Clauses

Record representation clauses are supported with the following restrictions:

- Allowed values in the alignment clause are 1 (byte-aligned), 2 (half-word aligned), 4 (full-word aligned) and 8 (double-word aligned).

- In the component clause, the storage unit offset (the *static_simple_expression* part) must be a word offset (i.e. 0 or a positive multiple of 4). The range of bits specified has the following restrictions: if the starting bit is 0, there is no limit on the value for the ending bit; if the starting bit is greater than 0, then the ending bit must be less than or equal to 31.

The actual size of the record object (including its use as a component of a record or array type) will always be a multiple of words (32 bits) with padding added to the end of the record, if necessary. User-specified ranges must contain at least the minimal number of bits required to represent a (bit-packed) object of the corresponding type; e.g. to represent an integer type with a range of 0..15, at least 4 bits must be specified in the record representation specification range. For more information about record layout, refer to the RISCAE Software Programmer's Reference Manual.

F.4.4: Address Clauses

An address clause may be supplied for an object (whether constant or variable), a subprogram, or a task entry, but not for a package or task unit. If an address clause is supplied for a subprogram, a body is not allowed for that subprogram.

An interrupt entry (address clause for an entry) may not have parameters.

F.5: Implementation Dependent Components

There are no implementation-generated names denoting implementation-dependent (record) components.

F.6: Interpretation of Expression in Address Clauses

This section describes the interpretation of expressions that appear in address clauses, including those for interrupts. `System.Address` is declared to be new `INTEGER`, hence, takes values from -2^{31} to $2^{31} - 1$. For address clauses on objects or subprograms, these values will be interpreted as (virtual) addresses in target memory as follows:

```
address >= 0  implies  (virtual) address == address
address < 0   implies  (virtual) address == (2**32) + address
```

For an object:

The meaning of the value given by an address clause for an object is the (virtual) address in the target memory assigned to that object.

For a subprogram:

The meaning of the value given by an address clause for a subprogram is the (virtual) address in target memory to which the program will branch when the user program makes a call to the subprogram. The user must supply the code to be executed and ensure that it is located at the indicated address.

For an entry:

The TRW RH32 SCU provides 12 interrupt levels, 0 to 11 with level 0 the highest priority, of which 8 are external interrupts, 3 are timer interrupts, and 1 is reserved by the hardware. If the value given by an address clause for a task entry is in the range 0..11, the entry will be called when the interrupt corresponding to that interrupt level is signaled. Two of the 3 timer interrupts, interrupt levels 2 and 4, are reserved for use by the RISCAE kernel in the implementation of package `CALENDAR` and to support Ada delay statements. If an interrupt entry attempts to use either of these interrupt levels, `PROGRAM_ERROR` will be raised during activation of the task containing the interrupt entry.

The RISCAE kernel also provides an `INTERFACE` which allows an application to cause a software interrupt to occur. Values in the range 12..19 are provided for software interrupts.

Any value outside the range of 0 .. 19 will cause PROGRAM_ERROR to be raised during activation of the task containing the interrupt entry.

F.7: UNCHECKED CONVERSION

There are no restrictions on the use of UNCHECKED_CONVERSION. Conversions between objects whose sizes do not confirm may result in storage areas with undefined values.

F.8: Input-Output

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

- The RISCAE Ada run time provides no support for external files nor for STANDARD_INPUT. The predefined exception USE_ERROR will be raised if an attempt is made to create or open any external file. The predefined exception END_ERROR will be raised if an attempt is made to read STANDARD_INPUT. Support for STANDARD_OUTPUT is implemented assuming the presence of a console I/O device which accepts output characters. The RISCAE simulator provides the effect of a console I/O device which is used to implement STANDARD_OUTPUT. (Implementation of STANDARD_OUTPUT for the RH32 ADM is TBD).
- Line terminator is ASCII.LF (line feed); page terminator is ASCII.FF (form feed).
- The packages SEQUENTIAL_IO and DIRECT_IO cannot be instantiated with unconstrained composite types or record types with discriminants without defaults.
- Package LOW_LEVEL_IO is not provided.

F.9: Tasking

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

F.9.1: Scheduling of Ada tasks

The scheduler of the Ada run-time tasking system runs tasks of equal priority in the order that they became eligible to run and allows them to run until blocked or until interrupted by the eligibility of a task of higher priority. A task whose priority is higher than the task currently running may be made eligible to run by an interrupt or by the expiration of a delay statement. Such an event will cause the currently running task to be immediately blocked so that the higher priority task may run.

F.9.2: Implementation-Dependent Termination of Library Unit Tasks

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

F.9.3: Implementation of Calendar

Support for implementation of Ada delay statements and for the function CLOCK in package CALENDAR are provided by the RISCAE kernel. The kernel implementation uses RTC (Real Time Clock) and interrupt levels 2 and 4 in the SCU of the TRW RH32 processor.

F.10: Other Matters

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

Restrictions on main program:

Any parameterless procedure which is a library unit may be a main program (LRM 10.1:8).

Order of compilation of generic bodies and subunits (LRM 10.3:9):

Body and subunits of generic must be in the same compilation as the specification if instantiations precede them (see AI-00257/02).