

AD-A260 064

ION PAGE

Form Approved
OPM No. 0704-0188

2

Public reporting
needed, and if
Headquarters
Managementuse, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
rate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
is Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENCY USE ONLY (Leave Blank)

E

3. REPORT TYPE AND DATES COVERED

Final: 04 Nov 92

4. TITLE AND SUBTITLE

Validation Summary Report: Tartan Inc., Tartan Ada VMS/C40 v4.2.1, DEC
VAXstation 4000/VMS (Host) to Texas Instruments TMS320C40 (Target),
92103011.11296

5. FUNDING NUMBERS

6. AUTHOR(S)

Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Ada Validation Facility, Language Control Facility ASD/SCCL
Bldg. 676, Rm 135
Wright-Patterson AFB, Dayton, OH 454338. PERFORMING ORGANIZATION
REPORT NUMBER

IABG-VSR 113

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-308110. SPONSORING/MONITORING AGENCY
REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Tartan Inc., Tartan Ada VMS/C40 v4.2.1, DEC VAXstation 4000/VMS (Host) to Texas Instruments TMS320C40 (Target),
ACVC 1.11.DTIC
SELECTE
JAN 27 1993
S B D

14. SUBJECT TERMS

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

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION
OF REPORT
UNCLASSIFIED18. SECURITY CLASSIFICATION
UNCLASSIFIED19. SECURITY CLASSIFICATION
OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 92103011.11296
Tartan Inc.
Tartan Ada VMS/C40 v4.2.1
DEC VAXstation 4000/VMS Host
Texas Instruments TMS320C40 Target
(bare machine)

93-01433



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

98 1 26 059

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on October 30, 1992.

Compiler Name and Version: Tartan Ada VMS/C40 v4.2.1
Host Computer System: Digital VAXstation 4000 Model 60
under VAX/VMS version 5.5
Target Computer System: Texas Instruments TMS320C40
Parallel Processing Development System
(bare machine)

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

As a result of this validation effort, Validation Certificate 921030I1.11296 is awarded to Tartan Inc. This certificate expires 24 months after ANSI approval of ANSI/MIL-STD-1815B.

This report has been reviewed and is approved.

Michael Tonndorf

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

John Solomond

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

DTIC QUALITY INSPECTED 8

John Solomond

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

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

UNCLASSIFIED

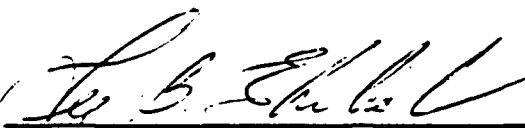
Appendix A.
Declaration of Conformance

Customer: Tartan, Inc.
Certificate Awardee: Tartan, Inc.
Ada Validation Facility: IABG mbH
ACVC Version: 1.11

Ada Implementation:
Ada Compiler Name and Version: Tartan Ada VMS/C40 v4.2.1
Host Computer System: Digital VAXstation 4000 Model 60
under VAX/VMS version 5.5
Target Computer System: Texas Instruments TMS320C40
Parallel Processing Development System
(bare machine)

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, FIPS 119 as tested in this validation and documented in the Validation Summary Report.


Lee B. Ehrlichman
Tartan, Inc.
President and Chief Executive Officer

11/2/92
Date

(Same)
Certificate Awardee Signature

(Same)
Date

Note: If the Customer and the Certificate Awardee are the same, only the customer signature is needed.

UNCLASSIFIED

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 [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 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
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

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 SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

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

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

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

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

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 August 02, 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 285 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

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

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

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.

A35801E checks that `FLOAT'FIRST..FLOAT'LAST` may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (See section 2.3.)

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.

C45536A, C46013B, C46031B, C46033B, and C46034B contain length clauses that specify values for `'SMALL` that are not powers of two or ten; this implementation does not support such values for `'SMALL`.

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

D64005G uses 17 levels of recursive procedure calls nesting; this 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.

CA2009A, CA2009C..D (2 tests), CA2009F, and BC3009C check whether a generic unit can be instantiated before its body (and any of its subunits) is compiled;

IMPLEMENTATION DEPENDENCIES

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.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

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.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

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 expect that NAME_ERROR is raised when an attempt is made to create a file with an illegal name; this implementation does not support the creation of external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 102 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	B24007A	B24009A	B25002B	B32201A	B33204A
B33205A	B35701A	B36171A	B36201A	B37101A	B37102A
B37201A	B37202A	B37203A	B37302A	B38003A	B38003B
B38008A	B38008B	B38009A	B38009B	B38103A	B38103B
B38103C	B38103D	B38103E	B43202C	B44002A	B48002A
B48002B	B48002D	B48002E	B48002G	B48003E	B49003A
B49005A	B49006A	B49006B	B49007A	B49007B	B49009A
B4A010C	B54A20A	B54A25A	B58002A	B58002B	B59001A
B59001C	B59001I	B62006C	B67001A	B67001B	B67001C
B67001D	B74103E	B74104A	B74307B	B83E01A	B85007C
B85008G	B85008H	B91004A	B91005A	B95003A	B95007B
B95031A	B95074E	BA1001A	BC1002A	BC1109A	BC1109C
BC1206A	BC2001E	BC3005B	BC3009C	BD2A06A	BD2B03A
BD2D03A	BD4003A	BD4006A	BD8003A		

E28002B was graded passed by Evaluation and Test Modification as directed by the AVO. This test checks that pragmas may have unresolvable arguments, and it includes a check that pragma LIST has the required effect; but for this implementation, pragma LIST has no effect if the compilation results in errors or warnings, which is the case when the test is processed without modification. This test was also processed with the pragmas at lines 46, 58, 70 and 71 commented out so that pragma LIST had effect.

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO. The compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. LRM 3.5.7:12).

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.

B83E01B was graded passed by Evaluation Modification as directed by the AVO. This test checks that a generic subprogram's formal parameter names (i.e. both generic and subprogram formal parameter names) must be distinct; the duplicated names within the generic declarations are marked as errors, whereas their recurrences in the subprogram bodies are marked as "optional" errors--except for the case at line 122, which is marked as an error. This implementation does not additionally flag the errors in the bodies and thus the expected error at line 122 is not flagged. The AVO ruled that the implementation's behavior was acceptable and that the test need not be split (such a split would simply duplicate the case in B83E01A at line 15).

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C 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

IMPLEMENTATION DEPENDENCIES

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:

BC3204C: C0, C1, C2, C3M, C4, C5, C6, C3M

BC3205D: D0, D1M, D2, D1M

BC3204D and BC3205C were graded passed by Test Modification as directed by the AVO. These tests are similar to BC3204C and BC3205D above, except that all compilation units are contained in a single compilation. For these two tests, a copy of the main procedure (which later units make obsolete) was appended to the tests; all expected errors were then detected.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-10 value as 'SMALL for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal 'SMALLs may be omitted.

AD9001B and AD9004A were graded passed by Processing Modification as directed by the AVO. These tests check that various subprograms may be interfaced to external routines (and hence have no Ada bodies). This implementation requires that a file specification exists for the foreign subprogram bodies. The following commands were issued to the Librarian to inform it that the foreign bodies will be supplied at link time (as the bodies are not actually needed by the programs, these commands alone are sufficient:

ALC40 interface/system AD9001B

ALC40 interface/system AD9004A

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:

Mr Ken Butler
Vice President, Product Development
Tartan Inc.
300, Oxford Drive
Monroeville, PA 15146 USA
Tel. (412) 856-3600

For sales information about this Ada implementation, contact:

Ms Marlyse Bennet
Tartan Inc.
12110 Sunset Hills Road
Suite 450
Reston, VA 22090 USA.
Tel. (703) 715-3044

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	3440	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	86	
d) Non-Processed I/O Tests	285	
e) Non-Processed Floating-Point Precision Tests	264	
f) Total Number of Inapplicable Tests	635	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

A TK50 cartridge containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the TK50 cartridge 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 host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link, an RS232 Interface, 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:

Options used for compiling:

/C40	Invoke the C40-targeted cross compiler. This qualifier is mandatory to invoke the C40-targeted compiler.
/REPLACE	Forces the compiler to accept an attempt to compile a unit imported from another library which is normally prohibited.
/NOSAVE_SOURCE	Suppresses the creation of a registered copy of the source code in the library directory for use by the REMAKE and MAKE subcommands to ALC40.

No explicit linker options were used.

Test output, compiler and linker listings, and job logs were captured on a TK50 cartridge 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	240 -- 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 lists all of the other macro parameters and their respective values.

MACRO PARAMETERS

Macro Parameter	Macro Value
\$ACC_SIZE	32
\$ALIGNMENT	1
\$COUNT_LAST	2147483646
\$DEFAULT_MEM_SIZE	16#FFFFFFFF#
\$DEFAULT_STOR_UNIT	32
\$DEFAULT_SYS_NAME	TI320C40
\$DELTA_DOC	2#1.0#E-31
\$ENTRY_ADDRESS	SYSTEM.ADDRESS'(16#2FF803#)
\$ENTRY_ADDRESS1	SYSTEM.ADDRESS'(16#2FF804#)
\$ENTRY_ADDRESS2	SYSTEM.ADDRESS'(16#2FF805#)
\$FIELD_LAST	240
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_TYPE
\$FLOAT_NAME	NO_SUCH_TYPE
\$FORM_STRING	" "
\$FORM_STRING2	"CANNOT RESTRICT_FILE_CAPACITY"
\$GREATER_THAN_DURATION	100_000.0
\$GREATER_THAN_DURATION BASE LAST	131_073.0
\$GREATER_THAN_FLOAT_BASE LAST	3.50282E+38
\$GREATER_THAN_FLOAT_SAFE LARGE	1.0E+38
\$GREATER_THAN_SHORT_FLOAT_SAFE LARGE	1.0E+38
\$HIGH_PRIORITY	100
\$ILLEGAL_EXTERNAL_FILE NAME1	ILLEGAL_EXTERNAL_FILE_NAME1
\$ILLEGAL_EXTERNAL_FILE NAME2	ILLEGAL_EXTERNAL_FILE_NAME2
\$INAPPROPRIATE_LINE_LENGTH	-1
\$INAPPROPRIATE_PAGE_LENGTH	-1
\$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")

MACRO PARAMETERS

```

$INCLUDE_PRAGMA2      PRAGMA INCLUDE ("B28006F1.TST")
$INTEGER_FIRST        -2147483648
$INTEGER_LAST         2147483647
$INTEGER_LAST_PLUS_1  2147483648
$INTERFACE_LANGUAGE   TI_C
$LESS_THAN_DURATION   -100_000.0
$LESS_THAN_DURATION_BASE_FIRST
                      -131_073.0
$LINE_TERMINATOR      ' '
$LOW_PRIORITY         10
$MACHINE_CODE_STATEMENT
                      Two_Opnds' (LDI, (Imm, 5), (Reg, R0));
$MACHINE_CODE_TYPE     Instruction_Mnemonic
$MANTISSA_DOC          31
$MAX_DIGITS           9
$MAX_INT              2147483647
$MAX_INT_PLUS_1       2147483648
$MIN_INT              -2147483648
$NAME                 NO_SUCH_TYPE_AVAILABLE
$NAME_LIST            TI320C40
$NAME_SPECIFICATION1  DUA2: {ACVC11.C30.TESTBED}X2120A.;1
$NAME_SPECIFICATION2  DUA2: {ACVC11.C30.TESTBED}X2120B.;1
$NAME_SPECIFICATION3  DUA2: {ACVC11.C30.TESTBED}X3119A.;1
$NEG_BASED_INT        16#FFFFFFFFE#
$NEW_MEM_SIZE         16#FFFFFFFF#
$NEW_STOR_UNIT        32
$NEW_SYS_NAME         TI320C40
$PAGE_TERMINATOR      ' '
$RECORD_DEFINITION    record Operation: Instruction_Mnemonic;
                      Operand_1: Operand; Operand_2: Operand;
                      end record;
$RECORD_NAME          Two_Opnds
$TASK_SIZE            32
$TASK_STORAGE_SIZE    4096
$TICK                 0.00006103515625

```

MACRO PARAMETERS

\$VARIABLE_ADDRESS	SYSTEM.ADDRESS'(16#2FF800#)
\$VARIABLE_ADDRESS1	SYSTEM.ADDRESS'(16#2FF801#)
\$VARIABLE_ADDRESS2	SYSTEM.ADDRESS'(16#2FF802#)
\$YOUR_PRAGMA	NO_SUCH_PRAGMA

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

Compiling Ada Programs

The TADA/C40 command is used to compile and assemble Ada compilation units.

4.1. THE TADA/C40 COMMAND

Format: The TADA/C40 command has this format:

```
S TADA/C40 [/qualifier [(option, ...) ...]] file-spec [/qualifier [(option, ...) ...]]
```

The parameter *file-spec* is a source file name. Since the source files need not reside in the directory in which the compilation takes place, *file-spec* must include sufficient directory information to locate the file. If no extension is supplied with the file name, a default extension of .ADA will be supplied by the compiler.

The source file may contain more than one compilation unit, but it is considered good practice to place only one compilation unit in a file.

Output: The compiler sequentially processes all compilation units in the file. Upon successful compilation of a unit,

- The Ada program library, LIBRARY.DB, is updated to reflect the new compilation time and any new dependencies.
- One or more separate compilation files and/or object files are generated.

If no errors are detected in a compilation unit, the compiler produces an object module and updates the library. If any error is detected, no object code file is produced, a source listing is produced, and no library entry is made for that compilation unit. If warnings are generated, both an object code file and a source listing are produced, and the library is updated.

Refinements: The compiler is capable of limiting the number of library units that become obsolete in the following manner. A library unit is a *refinement* of its previously compiled version if the only changes that were made are:

- Adding or deleting of comments
- Adding subprogram specifications after the last declarative item in the previous version

A qualifier is required to cause the compiler to detect refinements. When a refinement is detected by the compiler, dependent units are not marked as obsolete. If a unit is a refinement of its previous compilation, no other unit dependent on it becomes obsolete because of this recompilation. The exception to this rule is, the body of the specification is still obsolete for the case where a new declaration was added.

4.2. COMMAND QUALIFIERS

Command qualifiers indicate special actions to be performed by the compiler or special output file properties. A qualifier identifying the target-code format *must* be used to invoke the C40-targeted compiler. The following qualifiers are available:

/C40 Invoke the C40-targeted cross compiler. This qualifier is mandatory to invoke the C40-targeted compiler.

/CALLS=option Allows the user to specify the size of the address space into which the linked application code will fit. The compiler will generate code, based on the user's assertion. The available options are:

LONG The user asserts that the linked application code will *not* fit within a 23-bit ($2^{23} - 1$) address space.

SHORT The user asserts that the linked application code will fit within a 15-bit ($2^{15} - 1$) address space.

If the **/CALLS=SHORT** assertion is incorrect, the linker will produce error messages at link-time. If the **/CALLS=LONG** switch is given and the code fits within a 23-bit ($2^{23} - 1$) space, no error is given by the linker, because the code is still correct; however, it is less efficient. By default, the compiler will generate code assuming that the linked application code fits within a 23-bit ($2^{23} - 1$) address space.

/CROSS_REFERENCE
/NOCROSS_REFERENCE [Default]

Controls whether the compiler generates cross-reference information in the object code file to be used by the TXREF tool (see Section 4.5). This qualifier may be used only with the Tartan Tool Set.

/DATA_PAGE_IS_ROM

Limit data-page references to compile- and link-time constants. Statically allocated variables normally accessed using data page addressing will be reached via "long" references. This option must be used on all compilation units of an application if it is used on any one.

/DEBUG
/NODEBUG [Default]

Controls whether debugging information is included in the object code file. It is not necessary for all object modules to include debugging information to obtain a linkable image, but use of this qualifier is encouraged for all compilations. No significant execution-time penalty is incurred with this qualifier.

/DELAYED_BRANCHES [Default]
/NODELAYED_BRANCHES

Controls whether the compiler generates delayed branch instructions (detailed in Section 11.4).

/ENUMERATION_IMAGES [Default]
/NOENUMERATION_IMAGES

Causes the compiler to omit data segments with the text of enumeration literals. This text is normally produced for exported enumeration types in order to support the text attributes ('IMAGE', 'VALUE' and 'WIDTH').

You should use `/NOENUMERATION_IMAGES` only when you can guarantee that no unit that will import the enumeration type will use any of its text attributes. However, if you are compiling a unit with an enumeration type that is not visible to other compilation units, this qualifier is not needed. The compiler can recognize when the text attributes are not used and will not generate the supporting strings.

`/ERROR_LIMIT=n`

Stops compilation and produces a listing after *n* errors are encountered, where *n* is in the range 0 .. 255. The default value for *n* is 255. The `/ERROR_LIMIT` qualifier cannot be negated.

`/FIXUP[=option]`

When package `Machine_Code` is used, controls whether the compiler attempts to alter operand address modes when those address modes are used incorrectly. The available options are:

QUIET	The compiler attempts to generate extra instructions to fix incorrect address modes in the array aggregates operand field.
WARN	The compiler attempts to generate extra instructions to fix incorrect address modes. A warning message is issued if such a correction is required.
NONE	The compiler does <i>not</i> attempt to fix any machine code insertion that has incorrect address modes. An error message is issued for any machine code insertion that is incorrect.

When no form of this qualifier is supplied in the command line, the default condition is `/FIXUP=QUIET`. For more information on machine code insertions, refer to Section 5.10 of this manual.

`/HUGE_LOOPS [Default]`
`/NOHUGE_LOOPS`

When the `NOHUGE_LOOPS` qualifier is specified, the user is asserting that no loops will iterate more than 2^{23} times. This limit includes non-user specific loops, such as those loops generated by the compiler to operate on large objects. Erroneous code will be generated if this assertion is false.

`/LIBRARY=library-name`

Specifies the library into which the file is to be compiled. The compiler still reads any `ADALIB.INI` files in the default directory and will report any associated error, but this qualifier will override the `ADALIB.INI`.

`/LIST[=option]`
`/NOLIST`

Controls whether a listing file is produced. If produced, the file has the source file name and a `.LIS` extension. The available options are:

ALWAYS	Always produce a listing file
NEVER	Never produce a listing file, equivalent to <code>/NOLIST</code>
ERROR	Produce a listing file only if a compilation error or warning occurs

When no form of this qualifier is supplied in the command line, the default condition is `/LIST=ERROR`. When the `LIST` qualifier is supplied without an option, the default option is `ALWAYS`.

`/MACHINE_CODE[=option]`

Controls whether the compiler produces an assembly code file in addition to an object file, which is always generated. The assembly code file is not intended to be input to an assembler, but serves as documentation only. The available options are:

NONE	Do not produce an assembly code file.
INTERLEAVE	Produce an assembly code file which interleaves source code with the machine code. Ada source appears as assembly language comments.
NOINTERLEAVE	Produce an assembly code file without interleaving.

When no form of this qualifier is supplied in the command line, the default option is `NONE`. Specifying the `/MACHINE_CODE` qualifier without an option is equivalent to supplying `/MACHINE_CODE=NOINTERLEAVE`.

`/MAX_RPTS_COUNT=n`

Controls the maximum iteration count for a loop using the `RPTS` instruction, where n is an integer in the range $-1 \dots 2^{31} - 1$. Since an `RPTS` loop is non-interruptible, this qualifier allows control over the interrupt latency time. The default value is 32. A value of minus one (-1) specifies no limit. A value of zero (0) specifies that no `RPTS` instructions will be generated. Any positive value sets the maximum iteration count. If a value in the range $0 \dots 31$ is used, it will be necessary to customize the runtimes. Please contact Tartan for information on how to perform these customizations.

`/OPTIMIZE=option`

Controls the level of optimization performed by the compiler according to the following options: `MINIMUM`, `LOW`, `STANDARD`, `TIME`, and `SPACE`. The results of the options are:

MINIMUM	Performs context determination, constant folding, algebraic manipulation, and short circuit analysis. Inlines are not expanded.
LOW	Performs MINIMUM optimizations plus common sub-expression elimination and equivalence propagation within basic blocks. It also optimizes evaluation order. Inlines are not expanded. AdaScope performs best when compiled at this level.
STANDARD	(Best tradeoff for space/time) - <i>default option</i> . Performs LOW optimizations plus flow analysis which is used for common subexpression elimination and equivalence propagation across basic blocks. It also performs invariant expression hoisting, dead code elimination, and assignment killing. With STANDARD optimization, lifetime analysis is performed to improve register allocation and if possible, inline expansion of subprogram calls indicated by pragma <code>INLINE</code> are performed.

TIME	Performs STANDARD optimizations plus inline expansion of subprogram calls which the optimizer decides are profitable to expand (from an execution time perspective). Other optimizations which improve execution time at a cost to image size are performed only at this level.
SPACE	Performs those optimizations which usually produce the smallest code, often at the expense of speed. Please note that this optimization level may not always produce the smallest code. Under certain conditions, another level may produce smaller code.
/PARSE /NOPARSE	Extracts syntactically correct compilation unit source from the parsed file and loads this file into the library as a parsed unit. Parsed units are, by definition, inconsistent. This switch allows users to load units into the library without regard to correct compilation order. The command REMAKE is used subsequently to reorder the compilation units in the correct sequence. See Section 13.2.4.3 for a more complete description of this command.
/PHASES /NOPHASES [Default]	Controls whether the compiler announces each phase of processing as it occurs. These phases indicate progress of the compilation. If there is an error in compilation, the error message will direct users to a specific location.
/REFINE /NOREFINE [Default]	Controls whether the compiler, when compiling a library unit, determines whether the unit is a refinement of its previous version and, if so, does not make dependent units obsolete. The default is /NOREFINE.
/REMAKE	<i>Data on this switch is provided for information only.</i> This switch is used exclusively by the Tartan Ada Librarian to notify the compiler that the source undergoing compilation is an internal source file. The switch causes the compiler to retain old external source file information. This switch should be used only by the librarian and command files created by the librarian. See Section 3.6.1.
/REPLACE /NOREPLACE [Default]	Forces the compiler to accept an attempt to compile a unit imported from another library which is normally prohibited.
/REV= <i>n</i>	Revision of the C40 silicon, where <i>n</i> is an integer in the range 1 .. 2. The default value for <i>n</i> is 2.
/SAVE_SOURCE [Default]	

<code>/NOSAVE_SOURCE</code>	Suppresses the creation of a registered copy of the source code in the library directory for use by the <code>REMAKE</code> and <code>MAKE</code> subcommands to <code>ALC40</code> .																						
<code>/SUPPRESS[=(option, ...)]</code>	Suppresses the specific checks identified by the options supplied. The parentheses may be omitted if only one option is supplied. Invoking this option will not remove all checks if the resulting code without checks will be less efficient. The <code>/SUPPRESS</code> qualifier has the same effect as a global pragma <code>SUPPRESS</code> applied to the source file. If the source program also contains a pragma <code>SUPPRESS</code>, a given check is suppressed if either the pragma or the qualifier specifies it; that is, the effect of a pragma <code>SUPPRESS</code> cannot be negated with the command line qualifier. The <code>/SUPPRESS</code> qualifier cannot be negated. The available options are: <table> <tr> <td><code>ALL</code></td><td>Suppress all checks. This option is the default when no option is supplied.</td></tr> <tr> <td><code>ACCESS_CHECK</code></td><td>As specified in the Ada LRM, Section 11.7.</td></tr> <tr> <td><code>CONSTRAINT_CHECK</code></td><td>Equivalent of all the following: <code>ACCESS_CHECK</code>, <code>INDEX_CHECK</code>, <code>DISCRIMINANT_CHECK</code>, <code>LENGTH_CHECK</code>, <code>RANGE_CHECK</code>.</td></tr> <tr> <td><code>DISCRIMINANT_CHECK</code></td><td>As specified in the Ada LRM, Section 11.7.</td></tr> <tr> <td><code>DIVISION_CHECK</code></td><td>Will suppress compile-time checks for division by zero, but the hardware does not permit efficient runtime checks, so none are done.</td></tr> <tr> <td><code>ELABORATION_CHECK</code></td><td>As specified in the Ada LRM, Section 11.7.</td></tr> <tr> <td><code>INDEX_CHECK</code></td><td>As specified in the Ada LRM, Section 11.7.</td></tr> <tr> <td><code>LENGTH_CHECK</code></td><td>As specified in the Ada LRM, Section 11.7.</td></tr> <tr> <td><code>OVERFLOW_CHECK</code></td><td>Will suppress compile-time checks for overflow, but the hardware does not permit efficient runtime checks, so none are done.</td></tr> <tr> <td><code>RANGE_CHECK</code></td><td>As specified in the Ada LRM, Section 11.7.</td></tr> <tr> <td><code>STORAGE_CHECK</code></td><td>As specified in the Ada LRM, Section 11.7. Suppresses only stack checks in generated code, not the checks made by the allocator as a result of a new operation.</td></tr> </table>	<code>ALL</code>	Suppress all checks. This option is the default when no option is supplied.	<code>ACCESS_CHECK</code>	As specified in the Ada LRM, Section 11.7.	<code>CONSTRAINT_CHECK</code>	Equivalent of all the following: <code>ACCESS_CHECK</code> , <code>INDEX_CHECK</code> , <code>DISCRIMINANT_CHECK</code> , <code>LENGTH_CHECK</code> , <code>RANGE_CHECK</code> .	<code>DISCRIMINANT_CHECK</code>	As specified in the Ada LRM, Section 11.7.	<code>DIVISION_CHECK</code>	Will suppress compile-time checks for division by zero, but the hardware does not permit efficient runtime checks, so none are done.	<code>ELABORATION_CHECK</code>	As specified in the Ada LRM, Section 11.7.	<code>INDEX_CHECK</code>	As specified in the Ada LRM, Section 11.7.	<code>LENGTH_CHECK</code>	As specified in the Ada LRM, Section 11.7.	<code>OVERFLOW_CHECK</code>	Will suppress compile-time checks for overflow, but the hardware does not permit efficient runtime checks, so none are done.	<code>RANGE_CHECK</code>	As specified in the Ada LRM, Section 11.7.	<code>STORAGE_CHECK</code>	As specified in the Ada LRM, Section 11.7. Suppresses only stack checks in generated code, not the checks made by the allocator as a result of a new operation.
<code>ALL</code>	Suppress all checks. This option is the default when no option is supplied.																						
<code>ACCESS_CHECK</code>	As specified in the Ada LRM, Section 11.7.																						
<code>CONSTRAINT_CHECK</code>	Equivalent of all the following: <code>ACCESS_CHECK</code> , <code>INDEX_CHECK</code> , <code>DISCRIMINANT_CHECK</code> , <code>LENGTH_CHECK</code> , <code>RANGE_CHECK</code> .																						
<code>DISCRIMINANT_CHECK</code>	As specified in the Ada LRM, Section 11.7.																						
<code>DIVISION_CHECK</code>	Will suppress compile-time checks for division by zero, but the hardware does not permit efficient runtime checks, so none are done.																						
<code>ELABORATION_CHECK</code>	As specified in the Ada LRM, Section 11.7.																						
<code>INDEX_CHECK</code>	As specified in the Ada LRM, Section 11.7.																						
<code>LENGTH_CHECK</code>	As specified in the Ada LRM, Section 11.7.																						
<code>OVERFLOW_CHECK</code>	Will suppress compile-time checks for overflow, but the hardware does not permit efficient runtime checks, so none are done.																						
<code>RANGE_CHECK</code>	As specified in the Ada LRM, Section 11.7.																						
<code>STORAGE_CHECK</code>	As specified in the Ada LRM, Section 11.7. Suppresses only stack checks in generated code, not the checks made by the allocator as a result of a new operation.																						
<code>/SYNTAX_ONLY</code> <code>/NOSYNTAX_ONLY [Default]</code>	Examines units for syntax errors, then stops compilation without entering a unit in the library.																						
<code>/WARNINGS [Default]</code> <code>/NOWARNINGS</code>	Controls whether the warning messages generated by the compiler are displayed to the user at the terminal and in a listing file, if produced. While suppressing warning messages also halts display of informational messages, it does not suppress <code>Error</code> , <code>Fatal_Error</code> .																						

WAIT_STATES=*n*
 WAIT_STATES=*option:n* [, *option:n* ...]

The wait states command line qualifier accepts either a single numeric digit or a list of one or more *options*.

A single numeric digit specifies the number of wait states to use for program code, data page, heap, and stack, where *n* is an integer in the range 0 .. 7. The default value for *n* is 2 or the maximum wait state of program code, data page, heap, or stack.

The following *options* may be specified:

- | | |
|-----------------|--|
| CODE: <i>n</i> | Specifies the number of wait states for the block of memory in which program code will be executed, where <i>n</i> is an integer in the range 0 .. 7. The default value for <i>n</i> is 2. |
| DATA: <i>n</i> | Specifies the number of wait states for the block of memory where the data page for the current compilation unit resides, where <i>n</i> is an integer in the range 0 .. 7. The default value for <i>n</i> is 2. |
| HEAP: <i>n</i> | Specifies the number of wait states for the block of memory where the heap resides, where <i>n</i> is an integer in the range 0 .. 7. The default value for <i>n</i> is 2. |
| STACK: <i>n</i> | Specifies the number of wait states for the block of memory where the stack resides, where <i>n</i> is an integer in the range 0 .. 7. The default value for <i>n</i> is 2. |

Examples:

```
TADA/C40/WAIT_STATES=4
TADA/C40/WAIT_STATES=(CODE:4, DATA:2)
TADA/C40/WAIT=(C:1,D:2,H:3,S:4)
TADA/C40/WAIT_STATES=(HEAP=1, STACK=7)
TADA/C40/WAIT_STATES:2
TADA/C40/WAIT:CODE:1
```

Note that "=" and ":" are interchangeable. A keyword can be specified by using only enough characters to make it unique. When more than one *option* is specified, the list of *options* must be enclosed in parentheses.

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.

2.2. INVOKING THE LINKER

The linker may be invoked either through the Ada librarian, or directly by the user. Invocation through the librarian ensures that all Ada language consistency and dependency requirements are met. Use of the librarian is the most common way of linking an Ada program. Direct invocation of the linker is used mostly for link operations that are outside the boundaries of an Ada program. Examples of the latter are construction of assembly language boot images, combining multiple Ada programs into a single image, or creating patches for an existing program.

2.2.1. Using the Tartan Librarian to Invoke the Linker

Normally, Ada programs are linked using the Ada librarian. The command for linking through the librarian is

```
$ ALC40 LINK [/qualifier...] main_unit
```

The parameter *main_unit* must be supplied. It specifies the unit in the library to be made the main program.

The use of the Ada librarian commands to link a program is described in the *Compilation System Manual*, Chapters 3 and 9. Here we will describe the interface between the librarian and the linker, and the manner in which the necessary information is passed.

The Ada librarian's LINK subcommand checks that the unit within the library specified by the user has the legal form for a main unit, checks all its dependencies, finds all required object files, and invokes the linker. The librarian creates two files that are used as input to the link process. These files are listed by name below; the name *main_unit* refers to the name of the main program unit.

<i>main_unit</i> .CTL	This file contains the list of object files that are to be included in the link. The file is written as a list of linker WITH commands, each specifying a file to be included.
<i>main_unit</i> .etof	This object file contains a procedure to perform elaboration of the Ada program. The code in this file is included in the executable and will ultimately be invoked by the Ada runtimes when the program is executed.

These files are normally deleted by the Ada librarian at the completion of the link step. They may be retained by use of the qualifier /KEEP, described in the *Compilation System Manual*, Section 13.5.10. If no changes are made to the program that would invalidate the dependency and closure information contained in these files, subsequent links may be performed by invoking the linker directly.

After writing the necessary files, the Ada librarian invokes the linker. The invocation performed is equivalent to the following user-level command:

```
$ TLINKC40 /CONTROL=linker_control_file /OUTPUT=main_unit.XTOF
```

In this example, *linker_control_file* is the name of a file containing linker control commands. This control file describes the details of how the program image is to be constructed for the particular target system. The user may specify to the Ada librarian which linker control file to use. If no file is specified, the librarian uses a default control file located in the TADAHOME directory and named TLINKC40.LCF. Note that the default control file expects the qualifier /OUTPUT= to be specified in the command line. It also expects the file *main_unit*.CTL to exist.

2.2.2. Direct Invocation of the Linker

The linker is controlled by command qualifiers and by commands in a linker control file. Command qualifiers are used to specify things that vary according to the particular link being performed. Examples are the name of the output file, whether or not to produce a link map, and whether to eliminate unused code. The control file is used to specify in general how to build a program for the particular target system and hardware. The command qualifiers may vary with each link, but the control file is usually fixed for the system at hand.

The general format of the invocation of the linker is

```
$ TLINKC40 /CONTROL=linker_control_file [/qualifier...] filespec...
```

Some command qualifiers specify particular file names (for example, the name of the linker control file). Other file names may also appear in the command line; these names are interpreted as the names of object files to be included as input to the link process. Input files may also be specified within the linker control file.

When the linker is invoked directly, the qualifier `/CONTROL=linker_control_file` must be supplied in the command line. (When invoked by ALC40, it is the librarian, not the linker, that supplies a default linker control file.) This command qualifier directs the linker to the control file that specifies how to perform the link. Only one such file may be specified. A user who does not have a special control file may use the default file used by the Ada librarian. Refer to the previous section for a description of the link process used by the Ada librarian.

The additional arguments to the linker depend upon the convention used by the control file you specify. For example, input object files may be specified on the command line, in the control file, or in another linker control file. Specification on the command line is convenient if a small number is involved. For larger numbers of input files, the `WITH` command (Section 2.6.5) may be used inside the linker control file, or in another linker control file included with the `CONTROL` file command.

If your control file uses the same convention as the default one, the command line will look like:

```
S TLINKC40 /CONTROL=linker_control_file /OUTPUT=outfile.xtcf
```

The control file would then expect that the file `outfile.ct1` contains the list of input file `WITH` commands. A `CONTROL` command in the linker control file causes this additional file to be read. A derived file specification (see Section 2.6.9) is used in the `CONTROL` command to allow the linker to infer the file name from the specified output file name.

The convention used by the default control file is only one way in which arguments could be specified. Your own linker control file can be set up to expect the input and output file on the command line, or to derive the output filename from a specified input file, or to specify both in the control file. The convention used in special-purpose control files can be adjusted to fit the circumstances at hand. Section 2.5.2 introduces the `WITH`, `OUTPUT`, and `CONTROL` commands that are used to set up customized conventions.

A user who is simply relinking a program will need to know only a couple of command qualifiers and the convention established by the system-specific linker control file used for the system. A user who needs to alter the program layout for a specific target system will require the wider spectrum of commands available in the linker control file. Command qualifiers are described in the next section; the linker control file commands are described in Section 2.5.

2.3. COMMAND QUALIFIERS

This section describes the command qualifiers available to a user who directly invokes the linker. The qualifier names can be abbreviated to unique prefixes; the first letter is sufficient for all current qualifier names. The qualifier names are not case sensitive.

<code>/CONTROL=linker_control_file</code>	The specified file contains linker control commands. Only one such file may be specified, but it can include other files using the CONTROL command. Every invocation of the linker must specify a control file.
<code>/OUTPUT=filename</code>	The specified file is the name of the first output object file. The module name for this file will be null. Only one output file may be specified in this manner. Additional output files may be specified in the linker control file.
<code>/MAP</code>	Produce a link map containing all information except the unused section listings. When /MAP is specified without a file name, the name of the file containing the link map is specified by the LIST command in the linker control file. If your control file does not specify a name and you request a listing, the listing will be written to the default output stream.
<code>/MAP=filename</code>	Produce a link map containing all information except the unused section listings. The map is written to the specified file.
<code>/ALLOCATIONS</code>	Produce a link map showing the section allocations.
<code>/UNUSED</code>	Produce a link map showing the unused sections.
<code>/SYMBOLS</code>	Produce a link map showing global and external symbols.
<code>/LOCALS=filename</code>	Causes the linker to retain local symbol definitions in the output file specified.
<code>/RESOLVEMODULES</code>	This qualifier causes the linker to <i>not</i> perform unused section elimination. Specifying this option will generally make your program larger, since un-referenced data within object files will not be eliminated. Refer to Sections 2.6.3 and 2.4.3.2 for information on the way that unused section elimination works.

Note that several listing options are permitted because link maps for real systems can become rather large, and writing them consumes a significant fraction of the total link time. Options specifying the contents of the link map can be combined, in which case the resulting map will contain all the information specified by any of the switches.

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 -16#0.1000_00#E+33 .. 16#0.FFFF_FF#E+32;

type LONG_FLOAT is digits 9 range -16#0.1000_000_0#E+33 ..
                                16#0.FFFF_FFFF_0#E+32;
type DURATION is delta 0.0001 range -86400.0 .. 86400.0;

-- DURATION'SMALL = 2#1.0#E-14 (that is, 6.103516E-5 sec)
```

.....

end STANDARD;

Chapter 5

Appendix F to MIL-STD-1815A

This chapter contains the required Appendix F to the LRM, which is *Military Standard, Ada Programming Language*, ANSI/MIL-STD-1815A (American National Standards Institute, Inc., February 17, 1983).

5.1. PRAGMAS

5.1.1. Predefined Pragmas

This section summarizes the effects of and restrictions on predefined pragmas.

- Access collections are not subject to automatic storage reclamation so pragma CONTROLLED has no effect. Space deallocated by means of UNCHECKED_DEALLOCATION will be reused by the allocation of new objects.
- Pragma ELABORATE is supported.
- Pragma INLINE is supported.
- Pragma INTERFACE is supported. The LANGUAGE_NAME TI_C is used to make calls to subprograms (written in the Texas Instruments C language) from Tartan Ada. Any other LANGUAGE_NAME will be accepted, but ignored, and the default language, Ada, will be used.
- Pragma LIST is supported but has the intended effect only if the command qualifier /LIST=ALWAYS was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma MEMORY_SIZE is supported. See Section 5.1.3.
- Pragma OPTIMIZE is supported except when at the outer level (that is, in a package specification or body).
- Pragma PACK is supported.
- Pragma PAGE is supported but has the intended effect only if the command qualifier /LIST=ALWAYS was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma PRIORITY is supported.
- Pragma STORAGE_UNIT is accepted but no value other than that specified in package SYSTEM (Section 5.3) is allowed.
- Pragma SHARED is not supported.
- Pragma SUPPRESS is supported.
- Pragma SYSTEM_NAME is accepted but no value other than that specified in package SYSTEM (Section 5.3) is allowed.

5.1.2. Implementation-Defined Pragmas

Implementation-defined pragmas provided by Tartan are described in the following sections.

5.1.2.1. *Pragma LINKAGE_NAME*

The pragma `LINKAGE_NAME` associates an Ada entity with a string that is meaningful externally; for example, to a linkage editor. It takes the form

```
pragma LINKAGE_NAME (Ada-simple-name, string-constant)
```

The *Ada-simple-name* must be the name of an Ada entity declared in a package specification. This entity must be one that has a runtime representation; for example, a subprogram, exception or object. It may not be a named number or string constant. The pragma must appear after the declaration of the entity in the same package specification.

The effect of the pragma is to cause the *string-constant* to be used in the generated assembly code as an external name for the associated Ada entity. It is the responsibility of the user to guarantee that this string constant is meaningful to the linkage editor and that no illegal linkname clashes arise.

This pragma has no effect when applied to a subprogram or to a *renames* declaration; in the latter case, no warning message is given.

When determining the maximum allowable length for the external linkage name, keep in mind that the compiler will generate names for elaboration flags simply by appending the suffix `*GOTO`. Therefore, the external linkage name has 5 fewer significant characters than the lower limit of other tools that need to process the name (for example, 40 in the case of the Tartan Linker).

Note: Names used as pragma `LINKAGE_NAME` are case sensitive. For example, `aNy_Old_LINKname` is not equivalent to `ANY_OLD_LINKNAME`. Therefore, a misspelled linkname will cause the link to fail.

5.1.2.2. *Pragma FOREIGN_BODY*

In addition to pragma `INTERFACE`, Tartan Ada supplies pragma `FOREIGN_BODY` as a way to access subprograms in other languages.

Unlike pragma `INTERFACE`, pragma `FOREIGN_BODY` allows access to objects and exceptions (in addition to subprograms) to and from other languages.

There are some restrictions on pragma `FOREIGN_BODY` that are not applicable to pragma `INTERFACE`:

- Pragma `FOREIGN_BODY` must appear in a non-generic library package.
- All objects, exceptions and subprograms in such a package must be supplied by a foreign object module.
- Types may not be declared in such a package.

Use of the pragma `FOREIGN_BODY` dictates that all subprograms, exceptions and objects in the package are provided by means of a foreign object module. In order to successfully link a program including a foreign body, the object module for that body must be provided to the library using the `ALC40 FOREIGN_BODY` command described in Section 3.3.3. The pragma is of the form:

```
pragma FOREIGN_BODY (Language_name [, elaboration_routine_name])
```

The parameter *Language_name* is a string intended to allow the compiler to identify the calling convention used by the foreign module (but this functionality is not yet in operation). Currently, the programmer must ensure that the calling convention and data representation of the foreign body procedures are compatible with those used by the Tartan Ada Compiler (see Section 6.5). Subprograms called by tasks should be reentrant.

The optional *elaboration_routine_name* string argument is a linkage name identifying a routine to initialize the package. The routine specified as the *elaboration_routine_name*, which will be called for the elaboration of this package body, must be a global routine in the object module provided by the user.

A specification that uses this pragma may contain only subprogram declarations, object declarations that use an unconstrained type mark, and number declarations. Pragma may also appear in the package. The type mark for an object cannot be a task type, and the object declaration must not have an initial value expression. The pragma must be given prior to any declarations within the package specification. If the pragma is not located before the first declaration, or any restriction on the declarations is violated, the pragma is ignored and a warning is generated.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma FOREIGN_BODY. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

Pragma LINKAGE_NAME should be used for all declarations in the package, including any declarations in a nested package specification to be sure that there are no conflicting linknames. If pragma LINKAGE_NAME is not used, the cross-reference qualifier, /CROSS_REFERENCE, (see Section 4.2) should be used when invoking the compiler. The resulting cross-reference table of linknames should then be inspected to determine that no conflicting linknames have been assigned by the compiler (see also Section 4.5). In the following example, we want to call a function plmn which computes polynomials and is written in assembly.

```
package MATH_FUNCTIONS is
  pragma FOREIGN_BODY ("assembly");
  function POLYNOMIAL (X:INTEGER) return INTEGER;
  -- Ada spec matching the assembly routine
  pragma LINKAGE_NAME (POLYNOMIAL, "plmn");
  -- Force compiler to use name "plmn" when referring to this
  -- function
end MATH_FUNCTIONS;

with MATH_FUNCTIONS; use MATH_FUNCTIONS;
procedure MAIN is
  X:INTEGER := POLYNOMIAL(10);
  -- Will generate a call to "plmn"
  begin ...
end MAIN;
```

To compile, link and run the above program, you must:

1. Compile MATH_FUNCTIONS
2. Compile MAIN
3. Provide the object module (for example, math.TOF) containing the assembled code for plmn
4. Issue the command:

```
$ ALC40 FOREIGN MATH_FUNCTIONS MATH.TOF
```

5. Issue the command:

```
$ ALC40 LINK MAIN
```

Without Step 4, an attempt to link will produce an error message informing you of a missing package body for MATH_FUNCTIONS.

Using an Ada body from another Ada program library. The user may compile a body written in Ada for a specification into the library, regardless of the language specified in the pragma contained in the specification. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a foreign body with an Ada body without recompiling the specification.

The user can either compile an Ada body into the library, or use the command ALC40 FOREIGN_BODY (see Section 3.3.3) to use an Ada body from another library. The Ada body from another library must have been compiled under an identical specification. The pragma LINKAGE_NAME must have been applied to all entities declared in the specification. The only way to specify the linkname for the elaboration routine of an Ada body is with the pragma FOREIGN_BODY.

5.1.3. *Pragma* MEMORY_SIZE

This section details the procedure for compilation of a new unit, such as `pragma MEMORY_SIZE`, with a system pragma. The new unit must be compiled into a library that contains package `SYSTEM`. For most users, the `STANDARD_PACKAGES` library will be the library that also includes package `SYSTEM`.

1. Thaw `STANDARD_PACKAGES.SPEC`.
2. Compile this unit into `STANDARD_PACKAGES.ROOT`. This step updates package `SYSTEM`.
3. Freeze `STANDARD_PACKAGES.SPEC`.

Following these steps will allow you to modify the maximum address space.

5.2. IMPLEMENTATION-DEPENDENT ATTRIBUTES

No implementation-dependent attributes are currently supported.

5.3. SPECIFICATION OF THE PACKAGE SYSTEM

The parameter values specified for the Texas Instruments C40 processor family target in package SYSTEM (LRM 13.7.1 and Annex C) are:

```

package SYSTEM is
  type ADDRESS is new INTEGER;
  type NAME is (TI320C40);
  SYSTEM_NAME : constant NAME := TI320C40;
  STORAGE_UNIT : constant := 32;
  MEMORY_SIZE : constant := 16#FFFFFFFF#;
  MAX_INT : constant := 2_147_483_647;
  MIN_INT : constant := -MAX_INT - 1;
  MAX_DIGITS : constant := 9;

  MAX_MANTISSA : constant := 31;
  FINE_DELTA : constant := 2#1.0#e-31;
  TICK : constant := 0.00006103515625 -- 2**(-14)
  subtype PRIORITY is INTEGER range 10 .. 100;
  DEFAULT_PRIORITY : constant PRIORITY := PRIORITY'FIRST;
  RUNTIME_ERROR : exception;
end SYSTEM;

```

5.4. RESTRICTIONS ON REPRESENTATION CLAUSES

The following sections explain the basic restrictions for representation specifications followed by additional restrictions applying to specific kinds of clauses.

5.4.1. Basic Restriction

The basic restriction on representation specifications (LRM 13.1) is that they may be given only for types declared in terms of a type definition, excluding a `GENERIC_TYPE_DEFINITION` (LRM 12.1) and a `PRIVATE_TYPE_DEFINITION` (LRM 7.4). Any representation clause in violation of these rules is not obeyed by the compiler; an error message is issued.

Further restrictions are explained in the following sections. Any representation clauses violating those restrictions cause compilation to stop and a diagnostic message to be issued.

5.4.2. Length Clauses

Length clauses (LRM 13.2) are, in general, supported. The following sections detail use and restrictions.

5.4.2.1. Size Specifications for Types

The rules and restrictions for size specifications applied to types of various classes are described below.

The following principle rules apply:

1. The size is specified in bits and must be given by a static expression.
2. The specified size is taken as a mandate to store objects of the type in the given size wherever feasible. No attempt is made to store values of the type in a smaller size, even if possible. The following rules apply with regard to feasibility:

- An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine (i.e., no attempt is made to create objects of non-referable size on the stack). If such stack compression is desired, it can be achieved by the user by combining multiple stack variables in a composite object; for example:

```
type MY_ENUM is (A,B);
for MY_ENUM'SIZE use 1;
V,W: MY_ENUM; -- will occupy two storage
               -- units on the stack
               -- (if allocated at all)

type REC is record
  V,W: MY_ENUM;
end record;
pragma PACK(REC);
O: REC;      -- will occupy one storage unit
```

- A formal parameter of the type is sized according to calling conventions rather than size specifications of the type. Appropriate size conversions upon parameter passing take place automatically and are transparent to the user.
- Adjacent bits to an object that is a component of a composite object, but whose size is non-referable, may be affected by assignments to the object, unless these bits are occupied by other components of the composite object (i.e., whenever possible, a component of non-referable size is made referable).

In all cases, the compiler generates correct code for all operations on objects of the type, even if they are stored with differing representational sizes in different contexts.

Note: A size specification cannot be used to force a certain size in value operations of the type; for example:

```

type MY_INT is range 0..65535;
for MY_INT'SIZE use 16; -- o.k.
A,B: MY_INT;
...A + B... -- this operation will generally be
              -- executed on 32-bit values

```

3. A size specification for a type specifies the size for objects of this type and of all its subtypes. For components of composite types, whose subtype would allow a shorter representation of the component, no attempt is made to take advantage of such shorter representations. In contrast, for types without a length clause, such components may be represented in a lesser number of bits than the number of bits required to represent all values of the type. For example:

```

type MY_INT is range 0..2**15-1;
for MY_INT'SIZE use 16; -- (1)
subtype SMALL_MY_INT is MY_INT range 0..255;
type R is record
    ...
    X: SMALL_MY_INT;
    ...
end record;

```

the component R.X will occupy 16 bits. In the absence of the length clause at (1), R.X may be represented in 32 bits.

Size specifications for access types must coincide with the default size chosen by the compiler for the type.

Size specifications are not supported for floating-point types or task types.

No useful effect can be achieved by using size specifications for these types.

5.4.2.2. Size Specification for Scalar Types

The specified size must accommodate all possible values of the type including the value 0, even if 0 is not in the range of the values of the type. For numeric types with negative values, the number of bits must account for the sign bit. No skewing of the representation is attempted. Thus,

```

type MY_INT is range 100..101;

```

requires at least 7 bits, although it has only two values, while

```

type MY_INT is range -101..-100;

```

requires 8 bits to account for the sign bit.

A size specification for a real type does not affect the accuracy of operations on the type. Such influence should be exerted via the ACCURACY_DEFINITION of the type (LRM 3.5.7, 3.5.9).

A size specification for a scalar type may not specify a size larger than the largest operation size supported by the target architecture for the respective class of values of the type.

5.4.2.3. Size Specification for Array Types

A size specification for an array type must be large enough to accommodate all components of the array under the densest packing strategy. Any alignment constraints on the component type (see Section 5.4.7) must be met.

The size of the component type cannot be influenced by a length clause for an array. Within the limits of representing all possible values of the component subtype (but not necessarily of its type), the representation of components may, however, be reduced to the minimum number of bits, unless the component type carries a size specification.

If there is a size specification for the component type, but not for the array type, the component size is rounded up to a referable size, unless pragma PACK is given. This rule applies even to boolean types or other types that require only a single bit for the representation of all values.

5.4.2.4. Size Specification for Record Types

A size specification for a record type does not influence the default type mapping of a record type. The size must be at least as large as the number of bits determined by type mapping. Influence over packing of components can be exerted by means of (partial) record representation clauses or by pragma PACK.

Neither the size of component types, nor the representation of component subtypes can be influenced by a length clause for a record.

The only implementation-dependent components allocated by Tartan Ada in records contain either dope information for arrays whose bounds depend on discriminants of the record or relative offsets of components within a record layout for record components of dynamic size. These implementation-dependent components cannot be named or sized by the user.

A size specification cannot be applied to a record type with components of dynamically determined size.

Note: Size specifications for records can be used only to widen the representation accomplished by padding at the beginning or end of the record. Any narrowing of the representation over default type mapping must be accomplished by representation clauses or pragma PACK.

5.4.2.5. Specification of Collection Sizes

The specification of a collection size causes the collection to be allocated with the specified size. It is expressed in storage units and need not be static; refer to package SYSTEM for the meaning of storage units.

Any attempt to allocate more objects than the collection can hold causes a STORAGE_ERROR exception to be raised. Dynamically sized records or arrays may carry hidden administrative storage requirements that must be accounted for as part of the collection size. Moreover, alignment constraints on the type of the allocated objects may make it impossible to use all memory locations of the allocated collection. No matter what the requested object size, the allocator must allocate a minimum of 2 words per object. This lower limit is necessary for administrative overhead in the allocator. For example, a request of 5 words results in an allocation of 5 words; a request of one (1) word results in an allocation of 2 words.

In the absence of a specification of a collection size, the collection is extended automatically if more objects are allocated than possible in the collection originally allocated with the compiler-established default size. In this case, STORAGE_ERROR is raised only when the available target memory is exhausted. If a collection size of zero is specified, no access collection is allocated.

5.4.2.6. Specification of Task Activation Size

The specification of a task activation size causes the task activation to be allocated with the specified size. It is expressed in storage units; refer to package SYSTEM for the meaning of storage units.

Any attempt to exceed the activation size during execution causes a STORAGE_ERROR exception to be raised. Unlike collections, there is no extension of task activations.

5.4.2.7. Specification of 'SMALL

Only powers of 2 are allowed for 'SMALL.

The length of the representation may be affected by this specification. If a size specification is also given for the type, the size specification takes precedence; it must then be possible to accommodate the specification of 'SMALL within the specified size.

5.4.3. Enumeration Representation Clauses

For enumeration representation clauses (LRM 13.3), the following restrictions apply:

- The internal codes specified for the literals of the enumeration type may be any integer value between INTEGER'FIRST and INTEGER'LAST. It is strongly advised that you do not provide a representation clause that merely duplicates the default mapping of enumeration types which assigns consecutive numbers

in ascending order starting with zero (0). Unnecessary runtime cost is incurred by such duplication. It should be noted that the use of attributes on enumeration types with user-specified encodings is costly at runtime.

- Array types, whose index type is an enumeration type with non-contiguous value encodings, consist of a contiguous sequence of components. Indexing into the array involves a runtime translation of the index value into the corresponding position value of the enumeration type.

5.4.4. Record Representation Clauses

The alignment clause of record representation clauses (LRM 13.4) is observed.

Static objects may be aligned at powers of 2. The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record forced by the component allocation and the minimum alignment requirements of the components is already more stringent than the specified alignment.

The component clauses of record representation clauses are allowed only for components and discriminants of statically determinable size. Not all components need to be present. Component clauses for components of variant parts are allowed only if the size of the record type is statically determinable for every variant.

The size specified for each component must be sufficient to allocate all possible values of the component subtype, but not necessarily the component type. The location specified must be compatible with any alignment constraints of the component type; an alignment constraint on a component type may cause an implicit alignment constraint on the record type itself.

If some, but not all, discriminants and components of a record type are described by a component clause, the discriminants and components without component clauses are allocated after those with component clauses; no attempt is made to utilize gaps left by the user-provided allocation.

5.4.5. Address clauses

Address clauses (LRM 13.5) are supported with the following restrictions:

- When applied to an object, an address clause becomes a linker directive to allocate the object at the given address. For any object not declared immediately within a top-level library package, the address clause is meaningless.
- Address clauses applied to local packages are not supported by Tartan Ada. Address clauses applied to library packages are prohibited by the syntax; therefore, an address clause can be applied to a package only if it is a body stub.
- Address clauses applied to subprograms and tasks are implemented according to the LRM rules. When applied to an entry, the specified value identifies an interrupt in a manner customary for the target. Immediately after a task is created, a runtime call is made for each of its entries having an address clause, establishing the proper binding between the entry and the interrupt. A specified address must be an Ada static expression.

Note: Creating an overlay of two objects by means of address clauses is possible with Tartan Ada. However, such overlays (which are considered erroneous by the Ada LRM 13.5(8)) will not be recognized by the compiler as an aliasing that prevents certain optimizations. Therefore, problems may arise if reading and writing of the two overlaid objects are intermingled. For example, if variables A and B are overlaid by means of address clauses, the Ada code sequence:

```
A := 5;
B := 7;
if A = 5 then raise SURPRISE; end if;
```

may well raise the exception SURPRISE, since the compiler believes the value of A to be 5 even after the assignment to B.

5.4.6. *Pragma* PACK

Pragma PACK (LRM 13.1) is supported. For details, refer to the following sections.

5.4.6.1. *Pragma* PACK for Arrays

If *pragma* PACK is applied to an array, the densest possible representation is chosen. For details of packing, refer to the explanation of size specifications for arrays (Section 5.4.2.3).

If, in addition, a length clause is applied to the array type, the *pragma* has no effect, since such a length clause already uniquely determines the array packing method.

If a length clause is applied to the component type, the array is packed densely, observing the component's length clause. Note that the component length clause may have the effect of preventing the compiler from packing as densely as would be the default if *pragma* PACK is applied where there was no length clause given for the component type.

5.4.6.2. The Predefined Type STRING

Package STANDARD applies *pragma* PACK to the type STRING. However, because type character is determined to be 32 bits on the C40, this application results in one character per word.

5.4.6.3. *Pragma* PACK for Records

If *pragma* PACK is applied to a record, the densest possible representation is chosen that is compatible with the sizes and alignment constraints of the individual component types. *Pragma* PACK has an effect only if the sizes of some component types are specified explicitly by size specifications and are of non-referable nature. In the absence of *pragma* PACK, such components generally consume a referable amount of space.

It should be noted that the default type mapping for records maps components of boolean or other types that require only a single bit to a single bit in the record layout, if there are multiple such components in a record. Otherwise, it allocates a referable amount of storage to the component.

If *pragma* PACK is applied to a record for which a record representation clause has been given detailing the allocation of some but not all components, the *pragma* PACK affects only the components whose allocation has not been detailed. Moreover, the strategy of not utilizing gaps between explicitly allocated components still applies.

5.4.7. Minimal Alignment for Types

Certain alignment properties of values of certain types are enforced by the type mapping rules. Any representation specification that cannot be satisfied within these constraints is not obeyed by the compiler and is appropriately diagnosed.

Alignment constraints are caused by properties of the target architecture, most notably by the capability to extract non-aligned component values from composite values in a reasonably efficient manner. Typically, restrictions exist that make extraction of values that cross certain address boundaries very expensive, especially in contexts involving array indexing. Permitting data layouts that require such complicated extractions may impact code quality on a broader scale than merely in the local context of such extractions.

Instead of describing the precise algorithm of establishing the minimal alignment of types, we provide the general rule that is being enforced by the alignment rules:

- No object of scalar type (including components or subcomponents of a composite type) may span a target-dependent address boundary that would mandate an extraction of the object's value to be performed by two or more extractions.

5.5. IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record. These components cannot be named by the user.

5.6. INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES

Section 13.5.1 of the LRM describes a syntax for associating interrupts with task entries. Tartan Ada implements the address clause

for toentry use at intID;

by associating the interrupt specified by intID with the toentry entry of the task containing this address clause. The interpretation of intID is both machine and compiler dependent.

5.7. RESTRICTIONS ON UNCHECKED CONVERSIONS

Tartan supports UNCHECKED_CONVERSION as documented in Section 13.10 of the LRM. The sizes need not be the same, nor need they be known at compile time. If the value in the source is wider than that in the target, the source value will be truncated. If narrower, it will be zero-extended. Calls on instantiations of UNCHECKED_CONVERSION are made inline automatically.

5.8. IMPLEMENTATION-DEPENDENT ASPECTS OF INPUT-OUTPUT PACKAGES

Tartan Ada supplies the predefined input/output packages DIRECT_IO, SEQUENTIAL_IO, TEXT_IO, and LOW_LEVEL_IO as required by LRM Chapter 14. However, since the C40 chip is used in embedded applications lacking both standard I/O devices and file systems, the functionality of DIRECT_IO, SEQUENTIAL_IO, and TEXT_IO is limited.

DIRECT_IO and SEQUENTIAL_IO raise USE_ERROR if a file open or file access is attempted. TEXT_IO is supported to CURRENT_OUTPUT and from CURRENT_INPUT. A routine that takes explicit file names raises USE_ERROR.

5.9. OTHER IMPLEMENTATION CHARACTERISTICS

The following information is supplied in addition to that required by Appendix F to MIL-STD-1815A.

5.9.1. Definition of a Main Program

Any Ada library subprogram unit may be designated the main program for purposes of linking (using the Ada librarian's LINK subcommand) provided that the subprogram has no parameters.

Tasks initiated in imported library units follow the same rules for termination as other tasks [described in LRM 9.4 (6-10)]. Specifically, these tasks are not terminated simply because the main program has terminated. Terminate alternatives in selective wait statements in library tasks are therefore strongly recommended.

5.9.2. Implementation of Generic Units

All instantiations of generic units, except the predefined generic UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION subprograms, are implemented by code duplications. No attempt at sharing code by multiple instantiations is made in this release of Tartan Ada.

Tartan Ada enforces the restriction that the body of a generic unit must be compiled before the unit can be instantiated. It does not impose the restriction that the specification and body of a generic unit must be provided as part of the same compilation. A recompilation of the body of a generic unit will cause any units that instantiated this generic unit to become obsolete.

5.9.3. Implementation-Defined Characteristics in Package STANDARD

The implementation-dependent characteristics for C40 in package Standard (Annex C) are:

```
package STANDARD is
...
type INTEGER is range -2_147_483_648 .. 2_147_483_647;
type FLOAT is digits 6 range -16#0.1000_00#E+33 .. 16#0.FFFF_FF#E+32;

type LONG_FLOAT is digits 9 range -16#0.1000_000_0#E+33 ..
    16#0.FFFF_FFFF_0#E+32 ;
type DURATION is delta 0.0001 range -86400.0 .. 36400.0;
    -- DURATION'SMALL = 2#1.0#E-14 (that is, 6.103516E-5 sec)
...
end STANDARD;
```

5.9.4. Attributes of Type DURATION

The type DURATION is defined with the following characteristics:

Attribute	Value
DURATION' DELTA	0.0001 sec
DURATION' SMALL	6.103516E ⁻⁵ sec
DURATION' FIRST	-86400.0 sec
DURATION' LAST	86400.0 sec

5.9.5. Values of Integer Attributes

Tartan Ada supports the predefined integer type `INTEGER`. The range bounds of the predefined type `INTEGER` are:

Attribute	Value
<code>INTEGER' FIRST</code>	<code>-2**31</code>
<code>INTEGER' LAST</code>	<code>2**31-1</code>

The range bounds for subtypes declared in package `TEXT_IO` are:

Attribute	Value
<code>COUNT' FIRST</code>	<code>0</code>
<code>COUNT' LAST</code>	<code>INTEGER' LAST - 1</code>
<code>POSITIVE_COUNT' FIRST</code>	<code>1</code>
<code>POSITIVE_COUNT' LAST</code>	<code>INTEGER' LAST - 1</code>
<code>FIELD' FIRST</code>	<code>0</code>
<code>FIELD' LAST</code>	<code>240</code>

The range bounds for subtypes declared in package `DIRECT_IO` are:

Attribute	Value
<code>COUNT' FIRST</code>	<code>0</code>
<code>COUNT' LAST</code>	<code>INTEGER' LAST</code>
<code>POSITIVE_COUNT' FIRST</code>	<code>1</code>
<code>POSITIVE_COUNT' LAST</code>	<code>COUNT' LAST</code>

5.9.6. Values of Floating-Point Attributes

Tartan Ada supports the predefined floating-point types `Float` and `Long_Float`.

In addition, a set of standard library packages provides support for a non-Ada "double precision" 16-decimal digit float type, `Extended_Float`. Please refer to Section 8.2 for details. This type could not be supported as a predefined type due to Ada's "4*B" rule (LRM 3.5.7.7) that relates 'DIGITS' to the range of the machine exponent. Under this rule, the `Extended_Float` is indistinguishable from the `Long_Float` type.

Attribute	Value for <code>Float</code>
<code>DIGITS</code>	6
<code>MANTISSA</code>	21
<code>EMAX</code>	84
<code>EPSILON</code>	16#0.1000_00#E-4 (approximately 9.53674E-07)
<code>SMALL</code>	16#0.8000_00#E-21 (approximately 2.58494E-26)
<code>LARGE</code>	16#0.FFFF_FF#E+21 (approximately 1.93428E+25)
<code>SAFE_EMAX</code>	125
<code>SAFE_SMALL</code>	16#0.4000_00#E-31 (approximately 1.17549E-38)
<code>SAFE_LARGE</code>	16#0.1FFF_FF#E+32 (approximately 4.25353E+37)
<code>FIRST</code>	-16#0.1000_00#E+33 (approximately -3.40282E+38)
<code>LAST</code>	16#0.FFFF_FF#E+32 (approximately 3.40282E+38)
<code>MACHINE_RADIX</code>	2
<code>MACHINE_MANTISSA</code>	24
<code>MACHINE_EMAX</code>	128
<code>MACHINE_EMIN</code>	-125
<code>MACHINE_ROUNDS</code>	FALSE
<code>MACHINE_OVERFLOW</code>	TRUE

The Ada attributes are insufficient for completely describing floating point numbers, especially with non-symmetric machine exponent and machine mantissa ranges. For example, `MACHINE_EMAX` and `MACHINE_EMIN` are defined such that both the full mantissa range and the negative of any value must be supported in the floating format. This fails to document other, less restrictive exponent limits.

Additional (missing) properties are provided in the table below. The table is informational; there are *no* additional attributes corresponding to these values supplied by Tartan Ada. In the table, `POS_MACHINE_EMAX`, `NEG_MACHINE_EMAX`, `POS_MACHINE_EMIN`, `NEG_MACHINE_EMIN` are defined as the positive and negative floating value extremes for the machine exponent such that the full mantissa range is still supported (but no guarantees are made that the negative of any value is still representable). `POS_MACHINE_VERY_EMAX`, `NEG_MACHINE_VERY_EMAX`, `POS_MACHINE_VERY_EMIN`, `NEG_MACHINE_VERY_EMIN` are defined as the positive and negative floating value extremes for the machine exponent such that at least one floating value is still representable. `MOST_POSITIVE`, `LEAST_POSITIVE`, `MOST_NEGATIVE`, and `LEAST_NEGATIVE` are the absolute extremes possible in the floating format.

Property	Value for FLOAT
<code>POS_MACHINE_EMAX</code>	128
<code>NEG_MACHINE_EMAX</code>	128
<code>POS_MACHINE_EMIN</code>	-126
<code>NEG_MACHINE_EMIN</code>	-125
<code>POS_MACHINE_VERY_EMAX</code>	128
<code>NEG_MACHINE_VERY_EMAX</code>	129
<code>POS_MACHINE_VERY_EMIN</code>	-126
<code>NEG_MACHINE_VERY_EMIN</code>	-126
<code>MOST_POSITIVE</code>	16#0.FFFF_FF#E+32 (= 'LAST, approx 3.40282E+38)
<code>LEAST_POSITIVE</code>	16#2.0#E-32 (approx 5.87747E-39)
<code>MOST_NEGATIVE</code>	-16#0.1000_00#E+33 (= 'FIRST, approx -3.40282E+38)
<code>LEAST_NEGATIVE</code>	-16#2.0000_01#E-32 (approx -5.87747E-39)

Attribute	Value for LONG_FLOAT
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON	16#0.4000_0000_0#E-7 (approximately 9.31322575E-10)
SMALL	16#0.8000_0000_0#E-31 (approximately 2.35098870E-38)
LARGE	16#0.FFFF_FFFE_0#E+31 (approximately 2.12676479E+37)
SAFE_EMAX	125
SAFE_SMALL	16#0.4000_0000_0#E-31 (approximately 1.175494351E-38)
SAFE_LARGE	16#0.1FFF_FFFF_C#E+32 (approximately 4.253529587E+37)
FIRST	-16#0.1000_0000_0#E+33 (approximately -3.40282367E+38)
LAST	16#0.FFFF_FFFF_0#E+32 (approximately 3.40282367E+38)
MACHINE_RADIX	2
MACHINE_MANTISSA	32
MACHINE_EMAX	128
MACHINE_EMIN	-125
MACHINE_ROUNDS	FALSE
MACHINE_OVERFLOW	TRUE

The Ada attributes are insufficient for completely describing floating point numbers, especially with non-symmetric machine exponent and machine mantissa ranges. For example, MACHINE_EMAX and MACHINE_EMIN are defined such that both the full mantissa range and the negative of any value must be supported in the floating format. This fails to document other, less restrictive exponent limits.

Additional (missing) properties are provided in the table below. The table is informational; there are *no* additional attributes corresponding to these values supplied by Tartan Ada. In the table, POS_MACHINE_EMAX, NEG_MACHINE_EMAX, POS_MACHINE_EMIN, NEG_MACHINE_EMIN are defined as the positive and negative floating value extremes for the machine exponent such that the full mantissa range is still supported (but no guarantees are made that the negative of any value is still representable). POS_MACHINE_VERY_EMAX, NEG_MACHINE_VERY_EMAX, POS_MACHINE_VERY_EMIN, NEG_MACHINE_VERY_EMIN are defined as the positive and negative floating value extremes for the machine exponent such that at least one floating value is still representable. MOST_POSITIVE, LEAST_POSITIVE, MOST_NEGATIVE, and LEAST_NEGATIVE are the absolute extremes possible in the floating format.

Property	Value for LONG_FLOAT
POS_MACHINE_EMAX	128
NEG_MACHINE_EMAX	128
POS_MACHINE_EMIN	-126
NEG_MACHINE_EMIN	-125
POS_MACHINE_VERY_EMAX	128
NEG_MACHINE_VERY_EMAX	129
POS_MACHINE_VERY_EMIN	-126
NEG_MACHINE_VERY_EMIN	-126
MOST_POSITIVE	16#0.FFFF_FFFF#E+32 (= 'LAST, approx 3.40282367E+38)
LEAST_POSITIVE	16#2.0#E-32 (approx 5.87747175E-39)
MOST_NEGATIVE	-16#0.1000_0000#E+33 (= 'FIRST, approx -3.40282367E+38)
LEAST_NEGATIVE	-16#2.0000_0001#E-32 (approx -5.87747176E-39)

5.10. SUPPORT FOR PACKAGE MACHINE_CODE

Package MACHINE_CODE provides the user with an interface through which to request the generation of any instruction that is available on the C40. The implementation of package MACHINE_CODE is similar to that described in Section 13.8 of the Ada LRM, with several added features. Please refer to Appendix B for the package MACHINE_CODE specification.

5.10.1. Basic Information

As required by LRM, Section 13.8, a routine which contains machine code inserts may not have any other kind of statement, and may not contain an exception handler. The only allowed declarative item is a use clause. Comments and pragmas are allowed as usual.

5.10.2. Instructions

A machine code insert has the form TYPE_MARK' RECORD_AGGREGATE, where the type must be one of the records defined in package MACHINE_CODE. Package MACHINE_CODE defines seven types of records. Each has an opcode and zero to six operands. These records are adequate for the expression of all instructions provided by the C40.

5.10.3. Operands and Address Modes

An operand consists of a record aggregate which holds all the information to specify it to the compiler. All operands have an address mode and one or more other pieces of information. The operands correspond exactly to the operands of the instruction being generated.

Each operand in a machine code insert must have an *Address_Mode_Name*. The address modes provided in package MACHINE_CODE provide access to all address modes supported by the C40.

In addition, package MACHINE_CODE supplies the address modes SYMBOLIC_ADDRESS and SYMBOLIC_VALUE, which allow the user to refer to Ada objects by specifying OBJECT'ADDRESS as the value for the operand. Any Ada object which has the 'ADDRESS attribute may be used in a symbolic operand. SYMBOLIC_ADDRESS should be used when the operand is a true address (e.g., a branch target). SYMBOLIC_VALUE should be used when the operand is actually a value (i.e., one of the source operands of an ADDI instruction).

When an Ada object is used as a *source* operand in an instruction (that is, one from which a value is read), the compiler will generate code which fetches the *value* of the Ada object. When an Ada object is used as the destination operand of an instruction, the compiler will generate code which uses the *address* of the Ada object as the destination of the instruction.

5.10.4. Examples

The implementation of package MACHINE_CODE makes it possible to specify both simple machine code inserts such as

```
Two_Opnds' (LDI, (Imm, 5), (Reg, R0))
```

and more complex inserts such as

```
Three_Opnds' (ADDI3,
              (Imm, 10),
              (Symbolic_Value, Array_Var(X, Y, 27)'ADDRESS),
              (Symbolic_Address, Parameter_1'ADDRESS))
```

In the first example, the compiler will emit the instruction LDI 5, R0. In the second example, the compiler may first emit an instruction to load the immediate value 10 into a register (depending on whether the compiler decides to make the type1 or type2 three operand instruction), next emit whatever instructions are needed to form the address of ARRAY_VAR(X, Y, 27) and then emit the ADDI3 instruction. If PARAMETER_1 is not found in a register, the compiler will put the result of the addition in a temporary register and then store it to PARAMETER_1'ADDRESS. Note that the destination operand of the ADDI3 instruction is given as a

SYMBOLIC_ADDRESS. This holds true for all destination operands that are not also read as source operands by the instruction. **SYMBOLIC_VALUE** should be used if the operand is both a source and a destination as in the second operand of the **ADDI** instruction. The various error checks specified in the LRM will be performed on all compiler-generated code unless they are suppressed by the user (either through **pragma SUPPRESS**, or through command qualifiers).

5.10.5. Incorrect Operands

Under some circumstances, the compiler attempts to correct incorrect operands. Three modes of operation are supplied for package **MACHINE_CODE** to determine whether corrections are attempted and how much information about the necessary corrections is provided to the user. These modes of operation are **/FIXUP=NONE**, **/FIXUP=WARN**, and **/FIXUP=QUIET**. **/FIXUP=QUIET** is the default.

In **/FIXUP=NONE** mode, the specification of incorrect operands for an instruction is considered to be a fatal error. In this mode, the compiler will not generate any extra instructions to help you to make a machine code insertion. Note that it is still legal to use **'ADDRESS** constructs as long as the object which is used meets the requirements of the instruction.

In **/FIXUP=QUIET** mode, the compiler will do its best to correct the machine code if incorrect operands are specified. For example, although it is illegal to use a memory address as the destination of an **ADDI** instruction, the compiler will accept it and try to generate correct code. In this case, the compiler will load into a register the value found at the memory address indicated, use this register in the **ADDI** instruction, and then store from that register back to the desired memory location.

```
Two_Opnds'(ADDI, (Imm, 10), (ARI, AR1))
```

will produce a code sequence resembling

```
LDI      *AR1, R0
ADDI     10, R0
STI      R0, *AR1
```

The next example illustrates the correction required when the displacement is out of range for the first operand of an **ADDI3** instruction. The displacement is first loaded into one of the index registers.

```
Three_Opnds'(ADDI3, (IPDA, AR3, 32), (Reg, R0), (Reg, R1))
```

will produce a code sequence resembling

```
LDI      32, IRO
ADDI3    AR3(IRO), R0, R1
```

In **/FIXUP=WARN** mode, the compiler will also do its best to correct any incorrect operands for an instruction. However, a warning message is issued stating that the machine code insert required additional machine instructions to make its operands legal.

5.10.6. Assumptions Made in Correcting Operands

When compiling in **/FIXUP=QUIET** or **/FIXUP=WARN** modes, the compiler attempts to emit additional code to move "the right bits" from an incorrect operand to a register or place in memory which is a legal operand for the requested instruction. The compiler makes certain basic assumptions when performing these corrections. This section explains the assumptions made by the compiler and their implications for the generated code. Note that if you want a correction which is different from that performed by the compiler, you must make explicit machine code insertions to perform it.

For source operands:

- **SYMBOLIC_ADDRESS** means that the *address* specified by the **'ADDRESS** expression is used as the source bits. When the Ada object specified by the **'ADDRESS** instruction is bound to a register, it will cause a compile-time error message because it is not possible to "take the address" of a register.

- **SYMBOLIC_VALUE** means that the *value* found at the address specified by the 'ADDRESS expression will be used as the source bits. An Ada object which is bound to a register is correct here, because the contents of a register can be expressed on the C40.
- **PCREL** indicates that the *address* of the label will be used as the source bits.
- Any other non-register means that the *value* found at the address specified by the operand will be used as the source bits.

For destination operands:

- **SYMBOLIC_ADDRESS** means that the desired destination for the operation is the *address* specified by the 'ADDRESS expression. An Ada object which is bound to a register is correct here; a register is a legal destination on the C40.
- **SYMBOLIC_VALUE** means that the desired destination for the operations is found by fetching 32 bits from the address specified by the 'ADDRESS expression, and storing the result to the address represented by the fetched bits. This is equivalent to applying one extra indirection to the address used in the SYMBOLIC_ADDRESS case.
- All other operands are interpreted as directly specifying the destination for the operation.

5.10.7. Register Usage

Since the compiler may need to allocate registers as temporary storage in machine code routines, there are some restrictions placed on your register usage. The compiler will automatically free all registers which are volatile across a call for your use (R0..R3, bits 32-39 of R4..R5, bits 0-7 of R6..R7, bits 32-39 of R8, R9 .. R11, AR0..AR2, IR0, IR1, BK, ST, DIE, IIE, IIF, RS, RC, RE).

If you reference any other register, the compiler will reserve it for your use until the end of the machine code routine. The compiler will *not* save the register automatically if this routine is inline expanded. This means that the first reference to a register which is not volatile across calls should be an instruction which saves its value in a safe place. The value of the register should be restored at the end of the machine code routine. This rule will help ensure correct operation of your machine code insert even if it is inline expanded in another routine. However, the compiler will save the register automatically in the prolog code for the routine and restore it in the epilog code for the routine if the routine is *not* inline expanded.

As a result of freeing all volatile registers for the user, any parameters which were passed in registers will be moved to either a non-volatile register or to memory. References to **PARAMETER' ADDRESS** in a machine code insert will then produce code that uses this register or memory location. This means that there is a possibility of invalidating the value of some 'ADDRESS expression if the non-volatile register to which it is bound is used as a destination in some later machine code insert. In this case, any subsequent references to the 'ADDRESS expression will cause the compiler to issue a warning message.

The compiler may need several registers to generate code for operand corrections in machine code inserts. If you use all the registers, corrections will not be possible. In general, when more registers are available to the compiler, it is able to generate better code.

5.10.8. Data Directives

Two special instructions are included in package **MACHINE_CODE** to allow the user to place data into the code stream. These two instructions are **DATA32** and **DATA64**. Each of these instructions can have 1 to 6 operands.

DATA32 is used to place 32-bit data into the code stream. The value of an integer or 32-bit float, and the address of a label are the legal operands (i.e., operands whose address mode is either **IMM**, **FLOATIMM**, or **SYMBOLIC_ADDRESS** of an Ada label).

```

<< L1 >>
Three_Opnds' (DATA32, (Symbolic_Address, L2'Address),
                  (Symbolic_Address, L3'Address),
                  (Symbolic_Address, L4'Address));

<< L2 >>
<< L3 >>
<< L4 >>

```

will produce a code sequence like

```

L1:      .word L2
        .word L3
        .word L4

```

DATA64 is used to place a 64-bit piece of data into the code stream. The only legal operand is a floating literal (i.e., the operand whose address mode is FLOATIMM).

5.10.9. Inline Expansion

Routines which contain machine code inserts may be inline expanded into the bodies of other routines. This may happen under user control through the use of pragma INLINE, or with optimizations for *standard* and *timeoptimization* levels when the compiler selects that optimization as an appropriate action for the given situation. The compiler will treat the machine code insert as if it were a call. Volatile registers will be saved and restored around it and similar optimizing steps will be taken.

5.10.10. Move Macro Instructions

The C40 instruction set contains no single all-purpose move instruction, but instead supplies the set {LDI, LDF, STI, STF}. Each of these instructions defines a very specific kind of move with restrictions on data types and source/destination locations (memory vs. register). Unfortunately, when constructing data moves using package MACHINE_CODE, it is impossible to predict if an Ada object will be in memory or in a register, especially in the presence of inlining. For this reason, three "macro" instructions are supplied:

Name	Meaning
MOVI	Move a 32-bit integer from the first operand to the second, emitting some combination of LDI and STI's to do so.
MOVF32	Move a 32-bit float from the first operand to the second, emitting some combination of LDF and STF's to do so.
MOVF40	Move a 40-bit float from the first operand to the second, emitting some combination of LDF/LDI and STF/STI's to do so.

5.10.11. Using LAJ instructions

The code generated for a routine written in Ada has two entry points. One of the entry points is used by CALL instructions. The other entry point is used by LAJ instructions. The example below shows the two code sequences that the compiler can generate for a given routine:

```

_myfunc:      POP R11    ; CALL entry
_myfunc$LAJ:  ....      ; LAJ entry
             ....
             ....
             BU R11

_myfunc$LAJ:  PUSH R11   ; LAJ entry
_myfunc:      ....      ; CALL entry
             ....
             ....
             RETSU

```

Two groups of LAJ instructions are provided in package MACHINE_CODE. The special group of LAJ instructions that has the `_Ada` suffix should be used to call routines that are written in Ada. Using one of these special LAJ instructions tells the compiler that the target of this call should be the LAJ entry of the routine and not the CALL entry of the routine. If an LAJ without the `_Ada` suffix is used, the compiler will use the CALL entry of the routine as the target. The non `_Ada` version should be used to call routines that are not written in Ada (i.e., routines that are written in assembly, C, etc.).

Calling the Ada routine

```
One_Opnds' (LAJ_Ada, (Symbolic_Address, My_Ada_Function'Address));
```

will produce

```
LAJ  _myadafunc$LAJ
```

where the compiler uses the LAJ entry of `My_Ada_Function` as the target.

Calling the Assembly routine

```
One_Opnds' (LAJ, (Symbolic_Address, My_Assembly_Function'Address));
```

will produce

```
LAJ  _myassemblyfunc
```

and not

```
LAJ  _myassemblyfunc$LAJ
```

because the CALL entry of `MY_ASSEMBLY_FUNCTION` is used for the non `_Ada` LAJs.

If the source operand of a LAJCOND_ADA instruction is a register then the compiler cannot generate the LAJ entry for the routine. An error message is issued in this case. An Ada routine can be called by LAJCOND instructions whose source operand is a register if the non `_Ada` version is used. The following example shows how Ada routines can be called by LAJCOND instructions that have a register as the source operand.

```

Two_Opnds' (LDIU, (Symbolic_Address, My_Ada_Function'Address),
              (Reg, AR0));
One_Opnds' (LAJU, (Reg, AR0)); -- use the non_Ada version
Two_Opnds' (LDIU, (Imm, 1), (Reg, R0));
Two_Opnds' (LDIU, (Imm, 1), (Reg, R1));
One_Opnds' (PUSH, (Reg, R1)); -- return address is pushed on stack

```

will produce

```

LDIU    @DEF1, AR0
LAJU    AR0
LDIU    1, R0
LDIU    1, R1
PUSH    R11

....

DEF1: .word _myadafunc

```

In the above example, the target of the LAJU will be the CALL entry for MY_ADA_FUNCTION and not the LAJ entry. Since the target of the LAJU is the CALL entry, the return address must be pushed onto the runtime stack to simulate the semantics of a CALL instruction. This is done by making the PUSH R11 fill the last delay slot of the LAJU.

5.10.12. Unsafe Assumptions

There are a variety of assumptions which should *not* be made when writing machine code inserts. Violation of these assumptions may result in the generation of code which does not assemble or which may not function correctly.

- The compiler *will not* generate call site code for you if you emit a CALL or LAJ instruction. You must save and restore any volatile registers which currently have values in them, etc. If the routine you call has out parameters, a large function return result, or an unconstrained result, it is your responsibility to emit the necessary instructions to deal with these constructs as the compiler expects. In other words, when you emit a CALL or LAJ, you must follow the linkage conventions of the routine you are calling. For further details on call site code, see Sections 6.4, 6.5 and 6.6.
- Do not assume that the 'ADDRESS on SYMBOLIC_ADDRESS or SYMBOLIC_VALUE operands means that you are getting an ADDRESS to operate on. The Address- or Value-ness of an operand is determined by your choice of SYMBOLIC_ADDRESS or SYMBOLIC_VALUE. This means that to add the *contents* of X to AR0, you should write

```

Two_Opnds' (ADDI, (Symbolic_Value, X'ADDRESS),
              (Reg, AR0))

```

but to add the *address* of X to AR0, you should write

```

Two_Opnds' (ADDI, (Symbolic_Address, X'ADDRESS),
              (Reg, AR0));

```

5.10.13. Limitations

The current implementation of the compiler is unable to fully support automatic correction of certain kinds of operands. In particular, the compiler assumes that the size of a data object is the same as the number of bits which is operated on by the instruction chosen in the machine code insert. This means that the insert

```

Two_Opnds' (ADDF, (Symbolic_Value, Long_Float_Variable'ADDRESS),
              (Reg, R0))

```

will not generate correct code when LONG_FLOAT_VARIABLE is bound to memory. The compiler will assume that LONG_FLOAT_VARIABLE is 32 bits, when in fact it is stored in 64 bits of memory. If, on the other hand,

LONG_FLOAT_VARIABLE was bound to an extended-precision register, the insertion will function properly, as no correction is needed.

Note that the use of X'ADDRESS in a machine code insert *does not* guarantee that X will be bound to memory. This is a result of the use of 'ADDRESS to provide a "typeless" method for naming Ada objects in machine code inserts. For example, it is legal to say (SYMBOLIC_VALUE, X'ADDRESS) in an insert even when X is found in a register.

5.10.14. Example

```
with machine_code; use machine_code;
procedure mach_example is

    type ary_type is array(1..4) of integer;

    a: ary_type := (1,2,3,4);
    b: integer;

    procedure case_statement(a: in integer; b: in out integer) is
    begin
        -- implements case a is
        --      when 1 => b := 0;
        --      when 2 => b := b + 1;
        --      when 3 => b := b * b;
        --      when others => null
        --      end case;
        Three_Opnds'(SUBI3, (Imm, 1), (Symbolic_Value, a'Address), (Reg, IR0));
        Two_Opnds'(LDI, (Symbolic_Address, L1'Address), (Reg, Ar0));
        Two_Opnds'(LDI, (IPrIA, Ar0, IR0), (Reg, Ar1));
        One_Opnds'(case_jump, (Reg, Ar1));
        << L1 >>
        Three_Opnds'(DATA32, (Symbolic_Address, L2'Address),
                    (Symbolic_Address, L3'Address),
                    (Symbolic_Address, L4'Address));

        << L2 >>
        Two_Opnds'(LDI, (Imm, 0), (Symbolic_Address, b'Address));
        One_Opnds'(BU, (PcRel, L5'Address));
        << L3 >>
        Two_Opnds'(ADDI, (Imm, 1), (Symbolic_Value, b'Address));
        One_Opnds'(BU, (PcRel, L5'Address));
        << L4 >>
        Two_Opnds'(MPYI, (Symbolic_Value, b'Address), (Symbolic_Value, b'Address));
        << L5 >>
        Zero_Opnds'(NOP); -- since label can't be last statement in procedure
    end case_statement;

    pragma INLINE(case_statement);

begin
    if a(1) >= 0 then
        case_statement(a(3), b); -- will be inline expanded
    end if;

end mach_example;
```

Assembly code output:

```

.global mach_example

.global A00mchxmpl0009
.global A00mchxmpl0009SLAJ

.text

A00mchxmpl0009: POP      R11
A00mchxmpl0009SLAJ: ADDI    1,SP
                PUSH     AR3
                LDA      SP,AR3
                PUSH     AR3
                ADDI     4,SP
                PUSH     R5
                PUSH     R8
                LDA      @DEF1,AR0
                STI      AR0,*+AR3(1)

                LDA      @DEF2,AR0                ; line 7
                ADDI3    2,AR3,AR1
                LDIU     *AR0++(1),R1
                RPTS     2
                LDI      *AR0++(1),R1
                || STI   R1,*AR1++(1)
                STI      R1,*AR1
                CMPI3    0,*+AR3(2)                ; line 43
                BLT      L22
                LDIU     *+AR3(4),R0                ; line 44
                LDIU     R5,R8
                LDIU     R0,R5
                SUBI3    1,R5,IRO                ; line 18
                LDI      @DEF3,AR0                ; line 19
                LDI      *+AR0(IRO),AR1            ; line 20
                BU        AR1                ; line 21

L23:
L14:
                .word    L15
                ; line 23
                .word    L16
                .word    L17

L15:  LDI      0,R8                ; line 27
      BU      L18                ; line 28
L16:  ADDI     1,R8                ; line 30
      BU      L18                ; line 31
L17:  MPYI     R8,R8                ; line 33
L18:  NOP                      ; line 35
      LDIU     R8,R5                ; line 44

L22:
                ; line 40
      LDIU     *+AR3(6),R5
      LDIU     *+AR3(7),R8
      BUD      R11
      LDA      AR3,SP
      POP      AR3
      SUBI     1,SP

; Total words of code in the above routine = 41

.text ;assigned to "DEFAULT" data page
.sect "o:DEFAULT"
DEF3: .word    L14
DEF1: .word    L22

```

COMPILATION SYSTEM MANUAL

```
.text

casestatement$00:      PCP      R11
casestatement$00$LAJ:

      BU      R11

; Total words of code in the above routine = 2

      .text    ;assigned to "DEFAULT" data page
      .sect    "o:DEFAULT"

      .text    ;assigned to "DEFAULT" data page
      .sect    "o:DEFAULT"
DEFAULT: .word  0

      .text
      .text    ;assigned to "DEFAULT" data page
      .sect    "o:DEFAULT"
DEF2:   .word   DEF5
DEF5:   .word   1
      .word   2
      .word   3
      .word   4

; Total words of code = 43
; Total words of data = 7

.end
```

5.11. *INLINE GUIDELINES*

The following discussion on inlining is based on the next two examples. From these sample programs, general rules, procedures, and cautions are illustrated.

Consider a package with a subprogram that is to be inlined.

```
package IN_PACK is
  procedure I_WILL_BE_INLINED;
  pragma INLINE (I_WILL_BE_INLINED);
end IN_PACK;
```

Consider a procedure that makes a call to an inlined subprogram in the package.

```
with IN_PACK;
procedure USES_INLINED_SUBP is
begin
  I_WILL_BE_INLINED;
end;
```

After the package specification for IN_PACK has been compiled, it is possible to compile the unit USES_INLINED_SUBP that makes a call to the subprogram I_WILL_BE_INLINED. However, because the body of the subprogram is not yet available, the generated code will not have an inlined version of the subprogram. The generated code will use an out of line call for I_WILL_BE_INLINED. The compiler will issue warning message #2429 that the call was not inlined when USES_INLINED_SUBP was compiled.

If IN_PACK is used across libraries, it can be exported as part of a specification library after having compiled the package specification. Note that if only the specification is exported, there will be no inlined calls to IN_PACK in all units within libraries that import IN_PACK. If only the specification is exported, all calls that appear in other libraries will be out of line calls. The compiler will issue warning message #6601 to indicate the call was not inlined.

There is no warning at link-time that subprograms have not been inlined.

If the body for package IN_PACK has been compiled before the call to I_WILL_BE_INLINED is compiled, the compiler will inline the subprogram. In the example above, if the body of IN_PACK has been compiled before USES_INLINED_SUBP, the call will be inlined when USES_INLINED_SUBP is compiled.

Having an inlined call to a subprogram makes a unit dependent on the unit that contains the body of the subprogram. In the example, once USES_INLINED_SUBP has been compiled with an inlined call to I_WILL_BE_INLINED, the unit USES_INLINED_SUBP will have a dependency on the package body IN_PACK. Thus, if the body for package body IN_PACK is recompiled, USES_INLINED_SUBP will become obsolete, and must be recompiled before it can be linked.

It is possible to export the body for a library unit. If the body for package IN_PACK is added to the specification library using the Ada librarian subcommand EXPORT LIBRARY, other libraries that import package IN_PACK will be able to compile inlined calls across library units.

At optimization levels lower than the default, the compiler will not inline calls, even when pragma INLINE has been used and the body of the subprogram is in the library prior to the unit that makes the call. Lower optimization levels avoid any changes in flow of the code that causes movement of code sequences, as happens in a pragma INLINE. If the compiler is running at a low optimization level, the user will not be warned that inlining is not happening.

See Section 7.12 for a method to control inlining.