NTATION PAGE

Form Approved
OPM No.

EPORT

3. REPORT TYPE AND DATES
Final: 18 Mar 1992 tp 01 Jun 1993

4. TITLE AND
Validation Summary Report: Tartan, Inc., Tartan Ada SPARC 960mc Version 4.2, Sun SPARCstation/ELC(Host) to Intel EXV80960MC board (Target), 920313I1.11247

5. FUNDING

(2)

6.
IABG-AVF
Ottobrunn, Federal Republic of Germany

7. PERFORMING ORGANIZATION NAME(S) AND
IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION
IABG-VSR 86

9. SPONSORING/MONITORING AGENCY NAME(S) AND
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY

11. SUPPLEMENTARY

12a. DISTRIBUTION/AVAILABILITY
Approved for public release; distribution unlimited.

12b. DISTRIBUTION

13. *(Maximum 200*
Tartan, Inc., Tartan Ada SPARC 960mc Version 4.2, Sun SPARCstation/ELC (Host) to Intel EXV80960MC board (Target), ACVC 1.11.

92 0 [illegible]

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

15. NUMBER OF

16. PRICE

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

NSN

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

Certificate Information

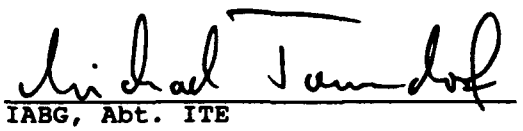
The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 13 March, 1992.

Compiler Name and Version:	**Tartan Ada SPARC 960mc version 4.2**
Host Computer System:	**Sun SPARCstation/ELC under SunOS Version 4.1.1**
Target Computer System:	**Intel EXV80960MC board (bare machine)**

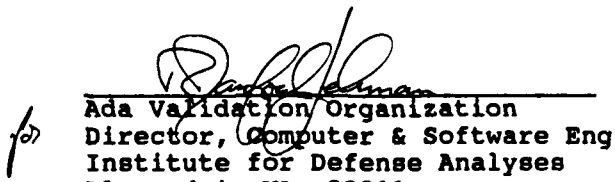
See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 920313I1.11247 is awarded to Tartan, Inc. This certificate expires 24 months after ANSI approval of MIL-STD 1815B.

This report has been reviewed and is approved.



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



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

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

AVF Control Number: IABG-VSR 86
18 March, 1992

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 920313I1.11247
Tartan, Inc.
Tartan Ada SPARC 960mc version 4.2
Sun SPARCstation/ELC =>
Intel EXV80960MC board

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

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 13 March, 1992.

Compiler Name and Version: **Tartan Ada SPARC 960mc version 4.2**

Host Computer System: **Sun SPARCstation/ELC under SunOS Version 4.1.1**

Target Computer System: **Intel EXV80960MC board (bare machine)**

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

As a result of this validation effort, Validation Certificate 920313I1.11247 is awarded to Tartan, Inc. This certificate expires 24 months after ANSI approval of MIL-STD 1815B.

This report has been reviewed and is approved.

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

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

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

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: Tartan, Inc.

Certificate Awardee: Tartan, Inc.

Ada Validation Facility: IABG mbH

ACVC Version: 1.11

Ada Implementation:

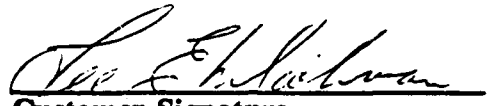
Ada Compiler Name and Version: Tartan Ada SPARC 960mc version 4.2

Host Computer System: SPARC Station/ELC SunOS version 4.1.1

Target Computer System: Intel EXV80960MC board (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 in the implementation listed above.



Customer Signature

March 16, 1992

Date

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

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

1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

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

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

1.2 REFERENCES

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

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

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

1.3 ACVC TEST CLASSES

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

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

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

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

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

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG89]).

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

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint	The part of the certification body which provides policy and

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

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

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 02 August 1991.

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

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. 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.

The following 159 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C241130..Y (11 tests)	C357050..Y (11 tests)
C357060..Y (11 tests)	C357070..Y (11 tests)
C357080..Y (11 tests)	C358020..Z (12 tests)
C452410..Y (11 tests)	C453210..Y (11 tests)
C454210..Y (11 tests)	C455210..Z (12 tests)
C455240..Z (12 tests)	C456210..Z (12 tests)
C456410..Y (11 tests)	C460120..Z (12 tests)

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

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

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_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

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 instantiate generic units before their bodies are compiled; this implementation creates a dependence on generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (see 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.

AE2101C and AE2101H use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults; these instantiations are rejected by this compiler.

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 114 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	BD2A06A	BD2B03A	BD2D03A
BD4003A	BD4006A	BD8003A			

E28002B was graded inapplicable 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.

Tests C45524A..N (14 tests) were graded passed by Test Modification as directed by the AVO. These tests expect that a repeated division will result in zero; but the Ada standard only requires that the result lie in the smallest safe interval. Thus, the tests were modified to check that the result was within the smallest safe interval by adding the following code after line 141; the modified tests were passed:

```
ELSIF VAL <= F'SAFE_SMALL THEN COMMENT ("UNDERFLOW SEEMS GRADUAL");
```

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 instantiate generic units before those units' bodies are compiled; this implementation creates dependences as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete, and the objectives of these tests cannot be met.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 & 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 compiling the seperate files in the following order (to allow re-compilation of obsolete units), and 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-ten 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 command was 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 program, this command alone is sufficient):

```
interface -sys -L=library ad9001b & 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 Ron Duursma
Director of Ada Products
Tartan, Inc.
300 Oxford Drive
Monroeville, PA 15146
USA
Tel. (412) 856-3600

For sales information about this Ada implementation, contact:

Ms. Marlyse Bennett
Director of Sales
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 [Pro90].

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

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn

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

a) Total Number of Applicable Tests	3614	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	38	
d) Non-Processed I/O Tests	264	
e) Non-Processed Floating-Point Precision Tests	159	
f) Total Number of Inapplicable Tests	461	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

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

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 for compiling:

-f	forces the compiler to accept an attempt to compile a unit imported from another library which is normally prohibited.
-c	suppresses the creation of a registered copy of the source code in the library directory for use by the REMAKE and MAKE subcommands.
-La	forces a listing to be produced, default is to only produce a listing when an error occurs.

No explicit Linker options were used.

Test output, compiler and linker listings, and job logs were captured on magnetic data 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
$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 Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2147483646
$DEFAULT_MEM_SIZE	2097152
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	I960MC
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.ADDRESS'(16#0000_00C8#)
$ENTRY_ADDRESS1	SYSTEM.ADDRESS'(16#0000_00C9#)
$ENTRY_ADDRESS2	SYSTEM.ADDRESS'(16#0000_00CA#)
$FIELD_LAST	240
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	EXTENDED_FLOAT
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	100_000_000.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E+38
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E+38
$HIGH_PRIORITY	17
$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")

```
$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006F1.TST")

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     Use_Call

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -100_000_000.0

$LINE_TERMINATOR        ' '

$LOW_PRIORITY           2

$MACHINE_CODE_STATEMENT
                        Two_Opnds'(MOV,(Reg_Lit,5),(Reg,R5));

$MACHINE_CODE_TYPE      Mnemonic

$MANTISSA_DOC           31

$MAX_DIGITS             18

$MAX_INT                9223372036854775807

$MAX_INT_PLUS_1         9223372036854775808

$MIN_INT                -9223372036854775808

$NAME                   BYTE_INTEGER

$NAME_LIST              I960MC

$NEG_BASED_INT          16#FFFFFFFFFFFFFFFE#

$NEW_MEM_SIZE           2097152

$NEW_STOR_UNIT          8

$NEW_SYS_NAME           I960MC

$PAGE_TERMINATOR        ' '

$RECORD_DEFINITION      record Operation: Mnemonic;
                        Operand_1: Operand;
                        Operand_2: Operand;
                        end record;

$RECORD_NAME            Two_Format

$TASK_SIZE              32

$TASK_STORAGE_SIZE      4096

$TICK                   0.015625

$VARIABLE_ADDRESS       SYSTEM.ADDRESS'(16#0000_1000#)

$VARIABLE_ADDRESS1      SYSTEM.ADDRESS'(16#0000_1004#)

$VARIABLE_ADDRESS2      SYSTEM.ADDRESS'(16#0000_1008#)
```

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and 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 compiler documentation and not to this report.

Chapter 4
Compiling Ada Programs

The `tada960mc` command is used to compile and assemble Ada compilation units.

4.1. *THE* `tada960mc` *COMMAND FORMAT*

The `tada960mc` command has this format:

% `tada960mc` [*option...*] *file...* [*option...*]

Arguments that start with a hyphen are interpreted as options; otherwise, they represent filenames. There must be at least one filename, but there need not be any options. Options and filenames may appear in any order, and all options apply to all filenames. For an explanation of the available options, see Section 4.2.

If a source file does not reside in the directory in which the compilation takes place, the *file* must include a path sufficient to locate the file. It is recommended that only one compilation unit be placed in a file.

If no extension is supplied with the file name, a default extension of `.ada` will be supplied by the compiler.

Files are processed in the order in which they appear on the command line. The compiler sequentially processes all compilation units in each file. Upon successful compilation of a unit:

- The library database, `Library.Db`, is updated with 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, `tada960mc` 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. For further details about the process of updating the library, files generated, replacement of existing files, and possible error conditions, see Sections 4.3 through 4.5.

The output from `tada960mc` is a file of type `.stof` or `.tof`, for a specification or a body unit respectively, containing object code. Some other files are generated as well. See Section 4.4 for a list of extensions of files that may be generated.

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

- Addition or deletion of comments.
- Addition of subprogram specifications after the last declarative item in the previous version.

An option is required to cause the compiler to detect refinements. When a refinement is detected by the compiler, dependent units are not marked as obsolete.

`-Me` When package `Machine_Code` is used, controls whether the compiler attempts to alter operand address modes when those address modes are used incorrectly. With this option, the compiler does not attempt to fix any machine code insertion that has incorrect address modes. An error message is issued for any incorrect machine code insertion. With the default, the compiler attempts to generate extra instructions to fix incorrect address modes in the array aggregates operand field.

`-Mw` The compiler attempts to generate extra instructions to fix incorrect address modes. A warning message is issued is such a *fixup* is required. With the default, the compiler attempts to generate extra instructions to fix incorrect address modes in the array aggregates operand field.

`-NBl` Suppress the `CALLJ` optimization. This optimization enables the linker to choose `CALL` or `BAL` calling instructions as appropriate. This switch should always be used when compiling the runtime sources. See Chapter 6.5.3 for details.

`-Op`*n* Control the level of optimization performed by the compiler, requested by *n*. The optimization levels available are:

- *n* = 0 **Minimum** - Performs context determination, constant folding, algebraic manipulation, and short circuit analysis.
- *n* = 1 **Low** - Performs level 0 optimizations plus common subexpression elimination and equivalence propagation within basic blocks. It also optimizes evaluation order.
- *n* = 2 **Best tradeoff for space/time** - *the default level*. Performs level 1 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. Level 2 also performs lifetime analysis which is used to improve register allocation. It also performs inline expansion of subprogram calls indicated by pragma `INLINE`, if possible.
- *n* = 3 **Time** - Performs level 2 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.
- *n* =4 **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.

`-p` 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 `remakecu` is used subsequently to reorder the compilation units in the correct sequence. See Section 13.2.5 for a more complete description of this command.

`-RS` Cause the compiler to accept non-Ada input, necessary to replace package `System`. This option should not be used for compiling user-defined packages with illegal code. Any changes of package `System` must conform to the requirements stated in ARM 4-5, 13.7, and 13.7.1, and must not change the given definition of type `ADDRESS`, in order to preserve validatability of the Ada system.

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, are outlined below for convenience.

```
package STANDARD is

      ...

   type BYTE_INTEGER  is range -128 .. 127;
   type SHORT_INTEGER is range -32768 .. 32767;
   type INTEGER       is range -2147483648 .. 2147483647;
   type LONG_INTEGER  is range -9223372036854775808 .. 9223372036854775807;

   type FLOAT is digits 6 range
-2#1.11111111111111111111111#e126 .. 2#1.11111111111111111111111#e126;

   type LONG_FLOAT is digits 15 range
-2#1.11111111111111111111111111111111111111111111111111111#e1022 ..
 2#1.11111111111111111111111111111111111111111111111111111#e1022;

   type EXTENDED_FLOAT is digits 18 range
-2#1.111111111111111111111111111111111111111111111111111111111111111#e16382 ..
 2#1.111111111111111111111111111111111111111111111111111111111111111#e16382;

   type DURATION is delta 0.0001 range -86400.0 .. 86400.0;

      ...

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. A particular Ada calling sequence is associated with a subprogram whose implementation is provided in the form of an object code module. `Language_Name` may be either `Use_Call`, `Use_Bal`, or `Intel_C` as described in Section 5.1.2.3. Any other `Language_Name` will be accepted, but ignored, and the default, `Use_Call` will be used.
- Pragma `LIST` is supported but has the intended effect only if the command line option `-La` 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, but on a subprogram basis only. It does not affect code at the block level.
- Pragma `PACK` is supported.
- Pragma `PAGE` is supported but has the intended effect only if the command line option `-La` 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 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.

A specification that uses this pragma may contain only subprogram declarations, object declarations that use an unconstrained type mark, and number declarations. Pragmas 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 link names.

In the following example, we want to call a function plmn which computes polynomials and is written in C.

```
package Math_Functions is
  pragma FOREIGN_BODY ("C");
  function POLYNOMIAL  (X:INTEGER) return INTEGER;
        -- Ada spec matching the C routine
  pragma LINKAGE_NAME (POLYNOMIAL, "plmn");
        -- Force compiler to use name "plmn" when referring to this
        -- function
        -- Note: The linkage name "plmn" may need to be "_plmn",
        --       if the C compiler produces leading underscores
        --       for external symbols.
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 do the following steps:

1. Compile Math_Functions
2. Compile MAIN
3. Provide the object module (for example, math.TOF) containing the compiled "C" code for plmn, converted to Tartan Object File Format (TOFF) using the COFF_TO_TOFF converter. Be sure to refer to the *C Common Symbols* section in the *Object File Utilities Manual*, Chapter 4.
4. Issue the command:

```
% adalib960mc foreign Math_Functions math.TOF
```

5. Issue the command:

```
% adalib960mc 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.

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 value of `System_Tick` at compilation time.

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 i960MC in package `System` [LRM 13.7.1 and Appendix C] are:

```
package System is
   type ADDRESS is new Integer;
   type NAME is (I960MC);

   SYSTEM_NAME   : constant name := I960MC;
   STORAGE_UNIT : constant := 8;
   MEMORY_SIZE   : constant := 2_097_152;

   MAX_INT       : constant := 9_223_372_036_854_775_807;
   MIN_INT       : constant := -MAX_INT - 1;

   MAX_DIGITS    : constant := 18;

   MAX_MANTISSA : constant := 31;
   FINE_DELTA    : constant := 2#1.0#e-31;
   TICK          : constant := 0.015625;
   subtype PRIORITY is INTEGER range 2 .. 17;
   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:

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.

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 dope information for arrays whose bounds depend on discriminants of the record or contain 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

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 accepted but meaningless. Please refer to Section 8.10 for details on how address clauses relate to linking; refer to Section 12.2 for an example.
- 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. Refer to Section 11.6.3 for more details. A specified address must be an Ada static expression.
- Address clauses which are applied to objects, subprograms, packages, or task units specify virtual, not physical, addresses.
- When specifying absolute addresses, please note that the compiler will treat the argument as a `System.Address` type. The range of `System.Address` is `-16#8000_0000#` to `16#7fff_ffff#`. To represent a machine virtual address in the range `16#0000_0000#` to `16#7fff_ffff#`, use the corresponding `System.Address`. To represent a machine virtual address greater than `System.Address` `16#7fff_ffff#`, use the negated radix-complement of the desired machine virtual address. For example, to express machine virtual address `16#C000_0000`, specify instead `System.Address -16#4000_0000`.

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.

5.5. 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 Ada Language Reference Manual 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.

The Ada runtimes provide interrupts that may be associated with task entries. These interrupts are of type `System.Address` in the ranges 8..243, 252..255, 264..499, and 508..511.

5.7. RESTRICTIONS ON UNCHECKED CONVERSIONS

Tartan supports `UNCHECKED_CONVERSION` as documented in Section 13.10 of the Ada Language Reference Manual. 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 i960MC processor is used in embedded applications lacking both standard I/O devices and file systems, the functionality of packages `Direct_IO`, `Sequential_IO`, and `Text_IO` is limited.

Packages `Direct_IO` and `Sequential_IO` raise `USE_ERROR` if a file open or file access is attempted. Package `Text_IO` is supported to `CURRENT_OUTPUT` and from `CURRENT_INPUT`. A routine that takes explicit file names raises `USE_ERROR`. Package `Low_Level_IO` for the i960MC processor provides an interface by which the user may read and write from memory mapped devices. In both the `SEND_CONTROL` and `RECEIVE_CONTROL` procedures, the device parameter specifies a device address while the data parameter is a byte, halfword, word, or doubleword of data transferred.

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 `adalib960mc link` command) 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

Attribute	Value
COUNT'FIRST	0
COUNT'LAST	INTEGER'LAST
POSITIVE_COUNT'FIRST	1
POSITIVE_COUNT'LAST	COUNT'LAST

5.9.5. Ordinal Types

Ordinal types are supported via two separate packages which are included with the standard packages. `Package Ordinal` provides support for unsigned arithmetic, including functions which convert between Integer and Ordinal types, and a complete set of Ordinal arithmetic operations. The specifications of package `Ordinal` may be found in Appendix D and `Intrinsic_Ordinal_Support` in Section E.2.3.

5.9.6. Values of Floating-Point Attributes

Tartan Ada supports the predefined floating-point types `FLOAT`, `LONG_FLOAT`, and `EXTENDED_FLOAT`.

Attribute	Value for `FLOAT`
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	16#0.1000_000#E-4 (approx. 9.536743E-07)
SMALL	16#0.8000_000#E-21 (approx. 2.58494E-26)
LARGE	16#0.FFFF_F80#E+21 (approx. 1.93428E+25)
SAFE_EMAX	126
SAFE_SMALL	16#0.2000_000#E-31 (approx. 5.87747EE-39)
SAFE_LARGE	16#0.3FFF_FE0#E+32 (approx. 8.50706+37)
FIRST	-16#0.7FFF_FFC#E+32 (approx. -1.70141E+38)
LAST	16#0.7FFF_FFC#E+32 (approx. 1.70141E+38)
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	126
MACHINE_EMIN	-126
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

Attribute	Value for EXTENDED_FLOAT
DIGITS	18
MANTISSA	61
EMAX	244
EPSILON	16#0.1000_0000_0000_0000_0#E-14 (approx. 8.67361737988403547E-19)
SMALL	16#0.8000_0000_0000_0000_0#E-61 (approx. 1.76868732008334226E-74)
LARGE	16#0.FFFF_FFFF_FFFF_FFF8_0#E+61 (approx. 2.82695530364541493E+73)
SAFE_EMAX	16382
SAFE_SMALL	16#0.2000_0000_0000_0000_0#E-4096 (approx. 1.68105157155604675E-4932)
SAFE_LARGE	16#0.3FFF_FFFF_FFFF_FFFF_0#E+4096 (approx. 2.97432873839307941E+4931)
FIRST	-16#0.7FFF_FFFF_FFFF_FFFF_8#E+4096 (approx. -5.94865747678615883E+4931)
LAST	16#0.7FFF_FFFF_FFFF_FFFF_8#E+4096 (approx. 5.94865747678615883E+4931)
MACHINE_RADIX	2
MACHINE_MANTISSA	63
MACHINE_EMAX	16382
MACHINE_EMIN	-16382
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

register and then store it to Parameter_1'ADDRESS. Note that the destination operand of the MULI instruction is given as a Symbolic_Address. This behavior holds true for all destination operands. The various error checks specified in the LRM will be performed on all compiler-generated code unless they are suppressed by the programmer, 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: -Me, -Mw and the default mode. These modes of operation determine whether corrections are attempted and how much information about the necessary corrections is provided to the user.

In -Me 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 default mode, if you specify incorrect operands for an instruction, the compiler will do its best to correct the machine code to provide the desired effect. For example, although it is illegal to use a memory address as the destination of an ADD instruction, the compiler will accept it and try to generate correct code. In this case, the compiler will allocate a temporary register to use as the destination of the ADD, and then store from that register to the desired location in memory.

In -Mw mode, the compiler will perform the same level of correction as in the default mode. However, a warning message is issued stating that the machine code insert required additional machine instructions to make its operands legal.

The compiler will *always* emit the instruction named in the machine code insert, even if it was necessary to correct all of its operands. In extreme cases this action can lead to surprising code sequences. Consider, for example, the machine code insert

```
Two_Format'(MOV,  (Reg_Ind,  G0),  (Reg_Ind_Disp,  G1, 128))
```

The MOV instruction requires two registers, but both operands are memory addresses. The compiler will generate a code sequence like

```
ld      (g0), g12
mov     g12, g13
st      g13, 128(g1)
```

Note that the MOV instruction is generated even though a LD ST combination would have been sufficient. As a result of always emitting the instruction specified by the programmer, the compiler will never optimize away instructions which it does not understand (such as SENDSERV), unless they are unreachable by ordinary control flow.

5.10.6. Assumptions Made in Correcting Operands

When compiling in -Mw and default modes, the compiler attempts to emit additional code to move "the right bits" from an incorrect operand to a place which is a legal operand for the requested instruction. The compiler makes certain basic assumptions when performing these corrections. This section explains the assumptions the compiler makes 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, this binding 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 960.

Inst	Opnd1	Opnd2	Opnd3
b	Error 2		
bx	Store to Memory 3		
bal	Error 2		
balx	Store to Memory 3		
bbc, bbs	Load to Register 1	Load to Register 1	Error 2
BRANCH IF	Error 2		
call	Error 2		
calls	Load to Register 1		
callx	Store to Memory 3		
chkbit	Load to Register 1	Load to Register 1	
classr, classrl	Load to Register 1		
clrbit	Load to Register 1	Load to Register 1	Store to Memory 2
cmpi, cmpo	Load to Register 1	Load to Register 1	
cmpdeci, cmpdeco	Load to Register 1	Load to Register 1	Store to Memory 2
cmpinci, cmpinco	Load to Register 1	Load to Register 1	Store to Memory 2
cmpor, cmporl	Load to Register 1	Load to Register 1	
cmpr, cmprl	Load to Register 1	Load to Register 1	
cmpstr	Load to Register 1	Load to Register 1	Load to Register 1
COMPARE AND BRANCH	Load to Register 1	Load to Register 1	Error 2
concmpi, concmpo	Load to Register 1	Load to Register 1	
condrec	Load to Register 1	Load to Register 1	
condwait	Load to Register 1		
cosr, cosrl	Load to Register 1	Store to Memory 2	
cpyrsre, cpysre	Load to Register 1	Load to Register 1	Store to Memory 2
cvtilr	Load to Register 1 (64 bits)	Store to Memory 2	
cvtir	Load to Register 1	Store to Memory 2	
cvtri	Load to Register 1	Store to Memory 2	
cvtril	Load to Register 1	Store to Memory 2 (64 bits)	
cvtzri	Load to Register 1	Store to Memory 2	
cvtzril	Load to Register 1	Store to Memory 2 (64 bits)	
daddc	Load to Register 1	Load to Register 1	Store to Memory 2
divo, divi, divr, divrl	Load to Register 1	Load to Register 1	Store to Memory 2

Table 5-1: Machine_Code Fixup Operations

Inst	Opnd1	Opnd2	Opnd3
notor	Load to Register 1	Load to Register 1	Store to Memory 2
or, ornot	Load to Register 1	Load to Register 1	Store to Memory 2
recieve	Load to Register 1	Load to Register 1	
remo, remi, remr, remrl	Load to Register 1	Load to Register 1	Store to Memory 2
resumprcs	Load to Register 1		
ret			
rotate	Load to Register 1	Load to Register 1	Store to Memory 2
roundr, roundrl	Load to Register 1	Store to Memory 2	
saveprcs			
scaler, scalerl	Load to Register 1	Load to Register 1	Store to Memory 2
scanbit	Load to Register 1	Store to Memory 2	
scanbyte	Load to Register 1	Load to Register 1	
schedprcs	Load to Register 1		
send	Load to Register 1	Load to Register 1	Load to Register 1
sendserv	Load to Register 1		
setbit	Load to Register 1	Load to Register 1	Store to Memory 2
SHIFT	Load to Register 1	Load to Register 1	Store to Memory 2
signal	Load to Register 1		
sinr, sinrl	Load to Register 1	Store to Memory 2	
spanbit	Load to Register 1	Store to Memory 2	
sqrtr, sqrtrl	Load to Register 1	Store to Memory 2	
STORE	Load to Register 1	Load To Register 2	
subo, subi, subc, subr, subrl	Load to Register 1	Load to Register 1	Store to Memory 2
syncf			
synld	Load to Register 1	Store to Memory 2	
synmov, synmovl, synmovq	Load To Register 1	Load to Register 1	
tanr, tanrl	Load to Register 1	Store to Memory 2	
TEST	Store to Memory 2		
wait	Load to Reg 1		
xnor, xor	Load to Register 1	Load to Register 1	Store to Memory 2

Table 5-1: Machine_Code Fixup Operations

```
Three_Format'(ADDI, (Symbolic_Value, X'ADDRESS),
                    (Reg, R3), (Reg, R3));
```

but to add the *address* of X to r3:

```
Three_Format'(ADDI, (Symbolic_Address, X'ADDRESS),
                    (Reg, R3), (Reg, R3));
```

- The compiler will not prevent writing to register r3, which is used to hold the address of the current exception handler. This feature provides the opportunity to make a custom exception handler. However, there is considerable danger in creating it. Knowledge of the details on the structure of exception handlers will help; see the *Tartan Ada Runtime Implementer's Guide, available only if you have purchased the Runtime Enhancement Package.*

5.10.10. Limitations

- When specifying absolute addresses in machine-code inserts, please note that the compiler will treat addresses as an `INTEGER` type and specifications of addresses may raise arithmetic overflow errors. Therefore, addresses must be in the range `INTEGER'FIRST..INTEGER'LAST`. To represent an address greater than `INTEGER'LAST`, use the negated radix-complement of the desired address. For example, to express address `16#C000_0000#`, specify instead `-16#4000_0000#`.
- 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 operated on by the instruction chosen in the machine-code insert. For example:

```
Three_Format'(ADDO, (Symbolic_Value, Byte_Variable'ADDRESS),
                    (Reg, R0), (Reg, R1))
```

  will not generate correct code when `Byte_Variable` is bound to memory. The compiler will assume that `Byte_Variable` is 32 bits, when in fact it is only 8, and will emit an `LD` instruction to load the value of `Byte_Variable` into a register. If, on the other hand, `Byte_Variable` was bound to a register, the insertion will function properly as no correction is needed.
- The compiler generates incorrect code when the `BAL` and `BALX` instructions are used with symbolic operands which are not of the form `Routine'ADDRESS`. To get the effect of an unconditional branch, use the `B` or `BX` instructions instead.
- Note that the use of `X'ADDRESS` in a machine code insert *does not* guarantee that `X` will be bound to memory because `'ADDRESS` provides a "typeless" method for naming Ada objects in machine code inserts. For example, it is legal to say to `(Symbolic_Value, X'ADDRESS)` in an insert even when `X` is a formal parameter of the machine code routine, and is thus found in a register.

```
procedure mtest1(first, second, third: in integer; fourth: out ary_type) is
begin

   -- Note the use of fourth(1)'ADDRESS as the destination of the MOV
   -- instruction.  The compiler will understand that the user "really
   -- wanted" something moved to fourth(1)'ADDRESS, and will make sure
   -- that the bits get there.  The compiler does NOT assume that it
   -- knows enough to second guess the user's choice of instructions.
   -- We generate the MOV, followed by a store to memory.
   Two_Format'(MOV,
               (Symbolic_Value, First'Address),
               (Symbolic_Address, fourth(1)'ADDRESS));
end mtest1;

procedure inline_into_me is
   array1 : ary_type := (1, 2, 3, 4);
begin
   if array1(3) >= 0 then
      -- note that mach_test is inline expanded
      mach_test(22, 41, array1(4), array1);
   else
      -- but mtest is not at Op=2 (No pragma INLINE)
      mtest1(1, 2, 3, array1);
   end if;
end inline_into_me;

end Mtest;
```

```
        faultg
        st      g12,-4(r8)[r4*4]
        st      g0,ADA.OWN
        lda     ADA.OWN,g14
        callx   xxmtest$inline_into_me$00
        b       .L19    #               line 76

.L17:   mov     1,g0    #               line 81

        mov     2,g1
        mov     3,g2
        lda     64(fp),g3
        bal     mtest1$00
.L19:

        ld      100(fp),g12
        ret

# Total bytes of code in the above routine = 216

        .align 4

mach_test$00:
        mov     0,r3
        st      sp,8(sp)
        addo    8,sp,sp
        st      g12,68(fp)
        lda     .L21,r3

        movl    g0,g6
        mov     g0,g6
        mov     g1,g7
        mov     g2,g8
        addi    g0,g8,g1
        muli    g7,g1,g12
        st      g12,64(fp)
        ld      64(fp),g13
        st      g13,(g3)
        ld      4(g3),g5
        xor     g13,g5,g12
        st      g12,8(g3)
        subi    1,g0,g13        #               line 46

        cmpo    g13,3
        faultg
        st      g12,-4(g3)[g0*4]
        st      g0,ADA.OWN
        lda     ADA.OWN,g14
        callx   xxmtest$inline_into_me$00
.L21:

        ld      68(fp),g12
        ret

# Total bytes of code in the above routine = 124

        .align 4

mtest1$00:
        mov     g14,g4
        mov     0,g14

        mov     g0,g13
        st      g13,(g3)

        bx      (g4)
```

5.11. *INLINE GUIDELINES*

The following discussion on inlining is based on the 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, that in all units in libraries that import `In_Pack` there will be no inlined calls to `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`, when `uses_Inlined_Subp` is compiled, the call will be inlined.

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, `exportlib` command, 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.