

AD-A239 761 **ATION PAGE**Form Approved
OPM No. 0704-0188Public rep
needed, a
Headquar
Managemeponse, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE		3. REPORT TYPE AND DATES COVERED Final: 14 June 1991 to 01 June 1993	
4. TITLE AND SUBTITLE Tartan Inc., Tartan Ada VMS/680X0 Version 4.1, VAXstation 3100 (Host) to Motorola MVM134 VMS 5.2 (Target), 91061311.11171				5. FUNDING NUMBERS	
6. AUTHOR(S) IABG-AVF Ottobrunn, Federal Republic of Germany					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) IABG-AVF, Industrieanlagen-Betriebsgesellschaft Dept. SZT/ Einsteinstrasse 20 D-8012 Ottobrunn FEDERAL REPUBLIC OF GERMANY				8. PERFORMING ORGANIZATION REPORT NUMBER IABG-VSR 096	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081				10. 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/680X0, Version 4.1, Ottobrunn Germany, VAXstation 3100 VMS 5.2(Host) to Motorola MVME134 (MC68020)(bare machine)(Target) ACVC 1.11.					
DTIC ELECTE AUG 26 1991 S B D 91-08765 					
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 UNCLASSIFIED		18. SECURITY CLASSIFICATION UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	
20. LIMITATION OF ABSTRACT					

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on June 13, 1991.

Compiler Name and Version: Tartan Ada VMS/680X0 version 4.1

Host Computer System: VAXstation 3100 VMS 5.2

Target Computer System: Motorola MVME134 (MC68020) (bare machine)

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

As a result of this validation effort, Validation Certificate 910613I1.11171 is awarded to Tartan Inc. This certificate expires on 1 March, 1993.

This report has been reviewed and is approved.

Michael Tonndorf

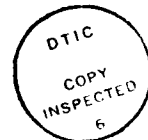
IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

[Signature]

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

John P. 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	

AVF Control Number: IABG-VSR 096
14 June 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910613I1.11171
Tartan Inc.
Tartan Ada VMS/680X0 version 4.1
VAXstation 3100 => Motorola MVME134
VMS 5.2 (MC68020) (bare machine)

== based on TEMPLATE Version 91-05-08 ==

Prepared By:
IABG mbh, Abt. ITE
Einsteinstrasse 20
W-8012 Ottobrunn
Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on June 13, 1991.

Compiler Name and Version: Tartan Ada VMS/680X0 version 4.1

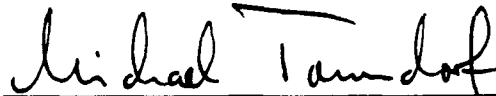
Host Computer System: VAXstation 3100 VMS 5.2

Target Computer System: Motorola MVME134 (MC68020) (bare machine)

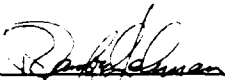
See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 910613I1.11171 is awarded to Tartan Inc. This certificate expires on 1 March, 1993.

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

Customer: Tartan, Inc.

Certificate Awardee: Tartan, Inc.

Ada Validation Facility: IABG

ACVC Version: 1.11

Ada Implementation:

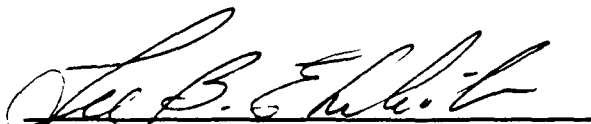
Ada Compiler Name and Version: Tartan Ada VMS/680X0 Version 4.1

Host Compiler System: VAXstation 3100 VMS 5.2

Target Computer System: Motorola MVME134 (MC68020)(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

Date:

16 June 91

TABLE OF CONTENTS

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

CHAPTER 1

INTRODUCTION

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

1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

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

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

1.2 REFERENCES

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

1.3 ACVC TEST CLASSES

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

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

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

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

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

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see Section 2.1) and, possibly some inapplicable tests (see Section 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 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 May 3, 1991.

E28005C	B28006C	C34006D	C35508I	C35508J	C35508M
C35508N	C35702A	C35702B	B41308B	C43004A	C45114A
C45346A	C45612A	C45612B	C45612C	C45651A	C46022A
B49008A	B49008B	A74006A	C74308A	B83022B	B83022H
B83025B	B83025D	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 ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

IMPLEMENTATION DEPENDENCIES

The following 201 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

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

C35404C	C45231C	C45304C	C45411C	C45412
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

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

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

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

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`.

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.

IMPLEMENTATION DEPENDENCIES

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 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

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 what the length clause specified, 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 require NAME_ERROR to be raised when an attempt is made to create a file with an illegal name; this implementation does not support external files and so raises USE_ERROR. (see 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see Section 1.3) were required for 110 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	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..K (11 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.

IMPLEMENTATION DEPENDENCIES

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 separate 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, 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

IMPLEMENTATION DEPENDENCIES

needed by the program, this command alone is sufficient):

AL68 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
Acting 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:

Mr Bill Geese
Director of Sales
Tartan Inc.
300, Oxford Drive
Monroeville, PA 15146
USA
Tel. (412) 856-3600

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	3553	
b) Total Number of Withdrawn Tests	94	
c) Processed Inapplicable Tests	58	
d) Non-Processed I/O Tests	264	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	523	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

The above number of I/O tests were not processed because this implementation does not support a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in Section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

A Magnetic Tape Reel containing the customized test suite (see Section 1.3) was taken on-site by the validation team for processing. The contents of the tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the 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:

/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.
/list=always	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 a Magnetic Tape Reel 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	1_000_000
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	MC68020
\$DELTA_DOC	2#1.0#E-31
\$ENTRY_ADDRESS	SYSTEM.ADDRESS' (16#F0#)
\$ENTRY_ADDRESS1	SYSTEM.ADDRESS' (16#F1#)
\$ENTRY_ADDRESS2	SYSTEM.ADDRESS' (16#F2#)
\$FIELD_LAST	240
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	THERE_IS_NO_SUCH_FLOAT_NAME
\$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	1.8E+39
\$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E+38

```

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
    1.0E+38

$HIGH_PRIORITY      200

$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 C

$LESS_THAN_DURATION -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
    -100_000_000.0

$LINE_TERMINATOR    ' '

$LOW_PRIORITY       10

$MACHINE_CODE_STATEMENT
    Two_Opnds' (MOVE_L, (IMM, 2), (ARIDEC, al));

$MACHINE_CODE_TYPE   Address_Mode

$MANTISSA_DOC        31

$MAX_DIGITS          15

$MAX_INT             2147483647

$MAX_INT_PLUS_1      2147483648

$MIN_INT             -2147483648

```

MACRO PARAMETERS

\$NAME	BYTE_INTEGER
\$NAME_LIST	MC68020
\$NAME_SPECIFICATION1	DUA2:[ACVC11.68K.TESTBED]X2120A.;1
\$NAME_SPECIFICATION2	DUA2:[ACVC11.68K.TESTBED]X2120B.;1
\$NAME_SPECIFICATION3	DUA2:[ACVC11.68K.TESTBED]X3119A.;1
\$NEG_BASED_INT	8#777777777776#
\$NEW_MEM_SIZE	1_000_000
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	MC68020
\$PAGE_TERMINATOR	' '
\$RECORD_DEFINITION	record Operation: Instruction_Mnemonic; Operand_1: Operand; end record;
\$RECORD_NAME	One_Opnds
\$TASK_SIZE	96
\$TASK_STORAGE_SIZE	4096
\$TICK	0.01
\$VARIABLE_ADDRESS	SYSTEM.ADDRESS' (16#C0000#)
\$VARIABLE_ADDRESS1	SYSTEM.ADDRESS' (16#C0004#)
\$VARIABLE_ADDRESS2	SYSTEM.ADDRESS' (16#C0008#)

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.

Supported compilation switches for Tartan Ada VMS 680X0. June 13, 1991.

/CALLS[=option] Controls whether the compiler generates 16-bit instead of 32-bit PC-relative address modes in call instructions when addressing objects whose distance from the current location cannot be determined at compile time. By using this qualifier, the user asserts that the program space for the final program will be small enough for all calls to use the 16-bit PC-relative address modes in call instructions. If this assertion is incorrect, erroneous code could result. The available options are:

SHORT Generate all short calls in the compiled code. Inappropriate use of this switch will cause a failure at link time.

MIXED The default option which generates short calls within the application code and long calls from applications to runtime routines.

/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.

/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.

/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`

Stop compilation and produce 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:

<code>QUIET</code>	The compiler attempts to generate extra instructions to fix incorrect address modes in the array aggregates operand field.
<code>WARN</code>	The compiler attempts to generate extra instructions to fix incorrect address modes. A warning message is issued if such a correction is required.
<code>NONE</code>	The compiler does not 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.

`/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:

<code>ALWAYS</code>	Always produce a listing file
<code>NEVER</code>	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.

/OPTIMIZE=option

Controls the level of optimization performed by the compiler according to the following options: minimum, low, standard, and time. 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 subexpression 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 - default option. 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 INLINE 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.

/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 9.2.5 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.

/REPLACE

Forces the compiler to accept an attempt to compile a unit imported from another library which is normally prohibited.

/REMAKE

Data on this switch is provided for information only. This switch is used exclusively by AdaLib to notify the compiler that the source undergoing compilation is an internal source file. The switch cause the compiler to retain old external source file information. This switch should be used only by AdaLib and command files created by AdaLib.

/SAVE_SOURCE [Default]

/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 AL68.

/SUPPRESS[=(option, ...)]

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 /SUPPRESS qualifier has the same effect as a global pragma SUPPRESS applied to the source file. If the source program also contains a pragma SUPPRESS, then a given check is suppressed if either the pragma or the qualifier specifies it; that is, the effect of a pragma SUPPRESS cannot be negated with the command line qualifier. The /SUPPRESS qualifier cannot be negated.

The available options are:

ALL	Suppress all checks. This is the default when no option is supplied.
ACCESS_CHECK	As specified in the Ada LRM, Section 11.7.
CONSTRAINT_CHECK	Equivalent of all the follow-

ing:
 ACCESS_CHECK, INDEX_CHECK,
 DISCRIMINANT_CHECK,
 RANGE_CHECK.

DISCRIMINANT_CHECK	As specified in the Ada LRM, Section 11.7.
DIVISION_CHECK	Will suppress compile-time checks for division by zero, but the hardware does not permit efficient runtime checks, so none are done.
ELABORATION_CHECK	As specified in the Ada LRM, Section 11.7.
INDEX_CHECK	As specified in the Ada LRM, Section 11.7.
LENGTH_CHECK	As specified in the Ada LRM, Section 11.7.
OVERFLOW_CHECK	Will suppress compile-time checks for overflow, but the hardware does not permit efficient runtime checks, so none are done.
RANGE_CHECK	As specified in the Ada LRM, Section 11.7.
STORAGE_CHECK	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.

/SYNTAX_ONLY Examines units for syntax errors, then stops compilation without entering a unit in the library.

/TARGET_680X0=(MC68020, MC68030, MC68040)
This switch allows you to select the target from the MC680X0 family.

/WARNINGS [Default]
/NOWARNINGS 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 Error, Fatal_Error.

Linker switches for Tartan Ada VMS 680X0.

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.

/CONTROL=file	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.
/OUTPUT=file	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.
/ALLOCATIONS	Produce a link map showing the section allocations.
/UNUSED	Produce a link map showing the unused sections.
/SYMBOLS	Produce a link map showing global and external symbols.
/RESOLVEMODULES	This qualifier causes the linker to not perform unused section elimination. Specifying this option will generally make your program larger, since unreferenced data within object files will not be eliminated. Refer to Sections 2.6.2 and 2.4.3.2 for information on the way that unused section elimination works.
/MAP	Produce a link map containing all information except the unused section listings.

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. 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.

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 INTEGER is range -2147483648 .. 2147483647;

type BYTE_INTEGER is range -128 .. 127;

type SHORT_INTEGER is range -32768 .. 32767;

type FLOAT is digits 6 range
-16#0.FFFFFFFF#e+32 .. 16#0.FFFFFFFF#e+32;

type LONG_FLOAT is digits 9 range
-16#0.FFFFFFFFFFFFFFFF8#e+256 .. 16#0.FFFFFFFFFFFFFFFF8#e+256;

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. It is assumed that the foreign code interfaced adheres to Tartan Ada calling conventions as well as Tartan Ada parameter passing mechanisms. Any other Language_Name will be accepted, but ignored, and the default 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 mis-spelled 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.

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

- 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 AL68 `FOREIGN` command described in Sections 3.3.3 and 9.5.6. 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 6.4). Subprograms called by tasks should be reentrant.

Accepted *language_name* for the 680X0 compiler include Ada, Assembly, and C.

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. Pragas 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. If pragma LINKAGE_NAME is not used, the cross-reference qualifier, /CROSS_REFERENCE, (see Section 4.2) should be used when invoking the compiler and the resulting cross-reference table of linknames inspected to identify the linknames assigned by the compiler and determine that there are no conflicting linknames (see also Section 4.5). 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
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); if the module is written in assembly code, for example, using the oasys_to_toff utility (See *Object File Utilities*, Chapter 4)
4. Issue the command

```
AL68 FOREIGN_BODY math_functions MATH.tof
```

5. Issue the command

```
AL68 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 AL68 FOREIGN (see Sections 3.3.3 and 9.5.6) 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 68K target in package `SYSTEM` [LRM 13.7.1 and Annex C] are:

```
package SYSTEM is
  type ADDRESS is new INTEGER;
  type NAME is (MC68020);
  SYSTEM_NAME : constant NAME := MC68020;
  STORAGE_UNIT : constant := 8;
  MEMORY_SIZE : constant := 1_000_000;
  MAX_INT : constant := 2_147_483_647;
  MIN_INT : constant := -MAX_INT - 1;
  MAX_DIGITS : constant := 15;

  MAX_MANTISSA : constant := 31;
  FINE_DELTA : constant := 2#1.0#e-31;
  TICK : constant := 0.01;
  subtype PRIORITY is INTEGER range 10 .. 200;
  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; that is, 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; that is, 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 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 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 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; the specification of 'SMALL' must then be accommodatable 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 to not provide a representation clause that merely duplicates the default mapping of enumeration types, which assigns consecutive numbers in ascending order starting with 0, since 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 run time.
- 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 up to a page boundary. 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, then 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.

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

1. The array type, the pragma has no effect, since such a length clause already uniquely determines the array packing method.
2. 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, when applied to character arrays, this pragma cannot be used to achieve denser packing than is the default for the target: 1 character per 8-bit 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 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 Motorola 680x0 specification provides 256 interrupts that may be associated with task entries. These interrupts are identified by an integer in the range 0..255, corresponding to the interrupt vector numbers in Section 6.2.1 of the *MC68020 32-Bit Microprocessor User's Manual*. When you specify an interrupt address clause, the *intID* argument is interpreted as follows:

- If the argument is in the range 0..255, a full support interrupt association is made between the interrupt specified by the argument and the task entry. That is, the runtimes make no assumptions about the task in question. This is the slower method.
- If the argument is in the range 256..511, a fast interrupt association is made between the interrupt number (argument-256) and the task entry. This method provides faster execution because the runtimes can depend upon the assumptions previously described.

For the difference between full support and fast interrupt handling, refer to Section 8.5.11.

5.7. RESTRICTIONS ON UNCHECKED CONVERSIONS

Tartan supports `UNCHECKED_CONVERSION` with a restriction that requires the sizes of both source and target types to be known at compile time. The sizes need not be the same. 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 target computer 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 AL68 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 as part of the same compilation. A recompilation of the body of a generic unit will casue any units that instantiated this generic unit to become obsolete.

5.9.3. Implementation-Defined Characteristics in Package STANDARD

The implementation-dependent characteristics in package STANDARD [Annex C] are:

```
package STANDARD is
...
type BYTE_INTEGER is range -128 .. 127;
type SHORT_INTEGER is range -32768 .. 32767;
type INTEGER is range -2_147_483_648 .. 2_147_483_647;
type FLOAT is digits 6 range -16#0.FFFFFFFF#E+32 .. 16#0.FFFFFFFF#E+32

type LONG_FLOAT is digits 9 range -16#0.FFFFFFFFFFFFFFFF8#E+256 ..
    16#0.FFFFFFFFFFFFFFFF8#E+256 ;
type DURATION is delta 0.0001 range -86400.0 .. 86400.0;
...
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 types INTEGER, SHORT_INTEGER and BYTE_INTEGER. The range bounds of these predefined types are:

Attribute	Value
INTEGER' FIRST	-2**31
INTEGER' LAST	2**31-1
SHORT_INTEGER' FIRST	-2**15
SHORT_INTEGER' LAST	2**15-1
BYTE_INTEGER' FIRST	-128
BYTE_INTEGER' LAST	127

The range bounds for subtypes declared in package TEXT_IO are:

Attribute	Value
COUNT' FIRST	0
COUNT' LAST	INTEGER' LAST - 1
POSITIVE_COUNT' FIRST	1
POSITIVE_COUNT' LAST	INTEGER' LAST - 1
FIELD' FIRST	0
FIELD' LAST	240

The range bounds for subtypes declared in packages DIRECT_IO are:

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

5.9.6. Values of Floating-Point Attributes

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

Attribute	Value for FLOAT
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	16#0.1000_00#E-4 (approximately 9.53674E-07)
SMALL	16#0.8000_00#E-21 (approximately 2.58494E-26)
LARGE	16#0.FFFF_F8#E+21 (approximately 1.93428E+25)
SAFE_EMAX	126
SAFE_SMALL	16#0.2000_000#E-31 (approximately 5.87747E-39)
SAFE_LARGE	16#0.3FFF_FE0#E+32 (approximately 8.50706E+37)
FIRST	-16#0.FFFFFFFF#E+32 (approximately -3.40282E+38)
LAST	16#0.FFFFFFFF#E+32 (approximately 3.40282E+38)
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	128
MACHINE_EMIN	-125
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

Attribute	Value for LONG_FLOAT
DIGITS	15
MANTISSA	53
EMAX	204
EPSILON	16#0.4000_0000_0000_000#E-12 (approximately 8.8817841970013E-16)
SMALL	16#0.8000_0000_0000_000#E-51 (approximately 1.9446922743316E-62)
LARGE	16#0.FFFF_FFFF_FFFF_E00#E+51 (approximately 2.5711008708143E+61)
SAFE_EMAX	1022
SAFE_SMALL	16#0.2000_0000_0000_000#E-255 (approximately 1.1125369292536-308)
SAFE_LARGE	16#0.3FFF_FFFF_FFFF_F80#E+256 (approximately 4.4942328371557E+307)
FIRST	-16#0.FFFFFFFFFFFFFFFF8#E+256 (approximately -1.79769313486232E+308)
LAST	16#0.FFFFFFFFFFFFFFFF8#E+256 (approximately -1.79769313486232E+308)
MACHINE_RADIX	2
MACHINE_MANTISSA	53
MACHINE_EMAX	1024
MACHINE_EMIN	-1021
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

5.10. SUPPORT FOR PACKAGE MACHINE_CODE

Package MACHINE_CODE provides the programmer with an interface to request the generation of any instruction that is available on the MC68020, MC68881, MC68030 and MC68040 processors. 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 A 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' RECORDAggregate, 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 6 operands. These records are adequate for the expression of all instructions provided by the 680X0.

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*. The address modes provided in package MACHINE_CODE provide access to all address modes supported by the 680X0.

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 (for example, a branch target). *Symbolic_Value* should be used when the operand is actually a value (for example, one of the source operands of an ADD 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' (MOVEQ, (Imm, 3), (DR, D0))
```

and more complex inserts such as

```
Two_Opnds' (ADDI_L,
            (Imm, 10),
            (Symbolic_Value, Array_Var(X, Y, 27)'ADDRESS))
```

In the first example, the compiler will emit the instruction MOVEQ 3, D0. In the second example, the compiler will first emit whatever instructions are needed to form the address of Array_Var(X, Y, 27) and then emit the ADDI_L instruction. 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`: `/FIXUP=NONE`, `/FIXUP=WARN`, and `/FIXUP=QUIET`. These modes of operation determine whether corrections are attempted and how much information about the necessary corrections is provided to the user. `/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, 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 an address register 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 the value found in the address register indicated into a data register, use the data register in the `ADDI` instruction, and then store from that data register back to the desired address register.

```
Two_Opnds' (ADDI_L, (Imm, 3), (AR, A1))
```

will produce a code sequence like

```
mov.l    a1,d0
addi.l   #3,d0
mov.l    d0,a1
```

In `/FIXUP=WARN` mode, the compiler will perform the same level of correction as in the `/FIXUP=QUIET` mode. 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 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 and source/destination 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 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 680X0. 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 680X0.
- `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

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.

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 (that is D0, D1, A0, A1, Fp0, Fp1).

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 explained 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.

5.10.8. Data Directives

Four special instructions are included in package Machine_Code to allow the user to place data into the code stream. These four instructions are DATA8, DATA16, DATA32 and DATA64. Each of these instructions can have from 1 to 6 operands.

DATA8 and DATA16 are used to place 8-bit and 16-bit integer data items into the code stream.

DATA32 is used to place 32-bit data into the code stream. The value of an integer, a floating point literal, or the address of a label or a routine are the legal operands (i.e. operands whose address mode is either Imm, Float_Lit_Single, or Symbolic_Address of an Ada object).

```
<< L1 >>
Three_Opnds' (DATA32, (Symbolic_Address, L1'Address),
                  (Float_Lit_Single, 2.0),
                  (Imm, 99));
```

will produce a code sequence like

```
L1:      .long L1
        .long 1073741824      | 0.2e1
        .long 99
```

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

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 programmer control through the use of pragma INLINE, or with optimization for *time* 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. 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.

- Do not assume that a machine code insert routine has its own set of local registers. This may not be true if the routine is inline expanded into another routine. Explicitly save and restore any registers which are not volatile across calls. If you wish to guarantee that a routine will never be inline expanded, you should use an Ada `separate` body for the routine or compile at an optimization level lower than the default. Then make sure there is no pragma `INLINE` for the routine.
- 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 `D0`, you should write

```
Two_Opnds' (ADD_L, (Symbolic_Value, X'ADDRESS), (DR, D0));
```

but to add the *address* of `X` to `D0`, you should write

```
Two_Opnds' (ADD_L, (Symbolic_Address, X'ADDRESS), (DR, D0));
```

- The compiler *will not* generate call site code for you if you emit a call instruction. You must save and restore any volatile registers (`D0`, `D1`, `A0`, `A1`, `Fp0`, `Fp1`) which currently have values in them. 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, 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 attempt to move multiple Ada objects with a single long instruction such as `MOVE`. Although the objects may be contiguous under the current circumstances, there is no guarantee that later changes will permit them to remain contiguous. If the objects are parameters, it is virtually certain that they will not be contiguous if the routine is inline expanded into the body of another routine. In the case of locals, globals, and own variables, the compiler does not guarantee that objects which are declared textually "next" to each other will be contiguous in memory. If the source code is changed such that it declares additional objects, this may change the storage allocation such that objects which were previously adjacent are no longer adjacent.

5.10.11. 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 in the insert:

```
Two_Opnds' (ADD_B, (Symbolic_Value, Integer_Variable'ADDRESS), (DR, D0))
```

the compiler will assume that `Integer_Variable` is 8 bits, when in fact it is stored in 32 bits of memory.

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.

- Absolute Short Address Mode with a symbolic operand is not supported. For example, the following operand is illegal:

```
(Abs_Short, Some_Variable'ADDRESS)
```

- In Address Modes in which two displacements are allowed only base displacement can be represented by a symbolic address. Outer displacement must be an integer. For example, this operand is legal:

```
(MEMPOST2, Bd_MEMPOST2    => Some_Routine'ADDRESS,  -- base displacement
  An_MEMPOST2              => A0,
  Xn_MEMPOST2              => D0,
  Xn_Size_MEMPOST2         => Long,
  Scale_MEMPOST2           => One,
  Od_MEMPOST2              => 16)  -- outer Displacement
```

while the following operand is illegal:

```
(MEMPOST2, Bd_MEMPOST2    => Routine_1'ADDRESS,  --base displacement
  An_MEMPOST2              => A0,
  Xn_MEMPOST2              => D0,
  Xn_Size_MEMPOST2         => Long,
  Scale_MEMPOST2           => One,
  Od_MEMPOST2              => Routine_2'ADDRESS)  --outer Displacement
```

- PC-relative Address Modes with a suppressed base register field can sometimes be handled incorrectly by the current implementation of the compiler.
- Extended precision floating point literals are not supported.

5.10.12. Address_Mode Usage

- Addressing modes that accept 16 or 32-bit displacements are represented by two entries in package Machine_Code's Address_Mode enumeration: one that accepts an integer, and one that accepts a symbolic address. For example, Memory Indirect Pre-Indexed addressing mode is represented by MEMPRE and MEMPRE2 Address Modes.
- DARI (Data or Address Register Indirect) Address_Mode is provided exclusively for use with operands five and six of the CAS2 instruction.
- ARIDX (Address Register Indirect with Index and Displacement) Address_Mode represents both the 8-bit displacement (Section 2.8.3.1 of MC68020 UM) and the base displacement (Section 2.8.3.2 of MC68020 UM) sub-modes of the Address Register Indirect with Index addressing mode. The compiler will pick the most economical form.
- PCIDX (Program Counter Indirect with Index and Displacement) Address_Mode represents both the 8-bit displacement (Section 2.8.6.1 of MC68020 UM) and the base displacement (Section 2.8.6.2 of MC68020 UM) sub-modes of the Program Counter Indirect with Index addressing mode. The compiler will pick the most economical form.

5.10.13. Instruction_Mnemonic Usage

- Instruction_Mnemonic names in package Machine_Code are formed by concatenating the base instruction name with a suffix representing the size of the instruction. For example, CMP_B, CMP_W, and CMP_L are package Machine_Code entries for the MC68020 CMP instruction. If the instruction exists in a single size only, it is represented by two entries in package Machine_Code: one with and one without a suffix. For example, the MC68020 LEA instruction is represented by LEA and LEA_L. Unsized instructions are represented by their base names with no suffix.
- For instructions that operate on control registers the control register operand needs to be explicitly supplied in the machine code insert:

```
Two_Opnds' (ANDItCCR, (Imm, 3), (CR, CCR));
```
- For Conditional Branch, Branch Always, and Branch to Subroutine instructions an unsized entry (for example, BEQ) lets the compiler pick the instruction of the optimal size.

- BSRnoret and JSRnoret mnemonics are aliases for BSR and JSR respectively. Use them when called routine is known to never return.
- The digit in the mnemonics of the Co-Processor instructions (cp*) indicates the number of optional co-processor defined extension words.
- When using MC68881 unary instructions which operate on a single floating point register, the register operand needs to be supplied as both the source and the destination operand:

```
Two_Opnds' (FCOSH_X, (FPR, Fp1), (FPR, Fp1));
```

The FTST instruction is the exception to this rule:

```
One_Opnds' (FTST_X, (FPR, Fp1));
```

- MC68881 instruction Move System Control Register, FMOVE is represented by several individual instructions, each of which requires an explicit control register operand:

```
Two_Opnds' (FMOVEtoFPCR, (DR, D0), (CR, FPCR));
```

- FSINCOS instruction returns the sine in its second operand and the cosine in its third operand.
- MC68881 Move Multiple Data Registers, FMOVEM_X and Move Multiple Control Registers, FMOVEM_L instructions, expect the register mask operand to be represented by an integer literal:

```
Two_Opnds' (FMOVEM_X, (Imm, 3), (ARI, A0));
```

5.10.14. Example

```
with machine_code; use machine_code;
with system; use system;

procedure Sincos (Source : in Long_Float;
                  Sin     : out Long_Float;
                  Cos     : out Long_Float) is
begin
  --
  -- Compute sine and cosine of Source and return them in
  -- parameters Sin and Cos, respectively
  --
  Three_Opnds' (FSINCOS_D, (Symbolic_Value, Source'ADDRESS),
                (FPR, Fp0),
                (FPR, Fp1));
  Two_Opnds' (FMOVE_D, (FPR, Fp0), (Symbolic_Address, Sin'address));
  Two_Opnds' (FMOVE_D, (FPR, Fp1), (Symbolic_Address, Cos'address));
end Sincos;
```

Assembly code output:

```
| TARTAN Ada Compiler, Version 4.1
| 1991

.data

.globl _a0fsincos

.text

_a0fsincos:
  link    a6, #0
  clr     a7@-

  fsincosd    a6@(8:w), fpl:fp0
  fmoved     fp0, a6@(16:w)
  fmoved     fp1, a6@(24:w)

  unlk     a6
```

```

rts

| Total bytes of code in the above routine = 28

.text

.even
| Total bytes of code = 28
| Total bytes of data = 0

```

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, then all calls that appear in other libraries will be out of line calls. The compiler will issue warning message #6601 to indicate 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, then the compiler will inline the subprogram. In the example above, if the body of `In_Pack` has been compiled before `uses_Inlined_Subp`, then 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, `EXPORT LIBRARY` command, then 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.

5.12. PACKAGE INTRINSICS

The Intrinsics package is provided as a means for the programmer to access certain hardware capabilities of the MC680X0 and MC68881 in an efficient manner.

The package declares generic functions which may be instantiated to create functions that have particularly efficient implementations. A call to such a function does not include a hardware subroutine call at all, but is implemented inline as a few MC680X0 or MC68881 instructions. Most of these functions are implemented as a single MC680X0 or MC68881 instruction.

5.12.1. Native Instructions

The following group of generic functions allows specific MC68020 instructions to be applied to Ada entities. The user must instantiate the generic function for the types that will be used as the operand(s) and result of the operation. These generic functions have been given the same name as the assembler's name for the corresponding instruction. In some cases this convention leads to a conflict with an Ada reserved word. This conflict is resolved by using the instruction name with an "i" appended to it.

For details of the operation applied by calling an instance of one of these generic functions see the *MC68020 User's Manual*. Examples of their use are given in Figures 5-1 and 5-2. Refer to Appendix B for the signatures of all intrinsics. The available operations are shown in the following table.

Name	Meaning
IOR	Bitwise Inclusive Logical OR
ANDi	Bitwise Logical AND
EOR	Bitwise Exclusive Logical OR
NOTi	Bitwise Ones Complement
MULU	Unsigned Multiplication
DIVU	Unsigned Division
LSL	Logical Shift Left
LSR	Logical Shift Right
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
ROL	Rotate Left
ROR	Rotate Right

```
with Intrinsics; use Intrinsics;

function Logical_And (Left      : short_integer;
                     Right     : short_integer) return integer is

    function Log_And is new ANDi (Left_Operand_Type => short_integer,
                                Right_Operand_Type => short_integer,
                                Result_Type       => integer);

begin
    return Log_And (Left, Right);
end Logical_And;
```

Figure 5-1: ANDi Used To Define a Logical AND Routine

```

with Intrinsic; use Intrinsic;

function Shift_Left (Shift_Me      : integer;
                    Shift_Count    : positive;
                    Signed         : boolean) return integer is

    function Log_Shift_Left is new LSL (Source_Type => integer,
                                        Result_Type => integer);
    function Ari_Shift_Left is new ASL (Source_Type => integer,
                                        Result_Type => integer);

begin
    if Signed then
        return Ari_Shift_Left (Shift_Me, Shift_Count);
    else
        return Log_Shift_Left (Shift_Me, Shift_Count);
    end if;
end Shift_Left;

```

Figure 5-2: LSL and ASL Used To Define a Shift-Left Routine

5.12.2. No-Overflow Integer Arithmetic

Occasionally, it is desirable to generate code sequences using operations defined for two's complement integer arithmetic, but without the overflow checks that usually come with them. Such is the case when emulating unsigned 32-bit arithmetic, for example. Four functions are provided. All of them are generic on source and result types and can thus be instantiated for 8, 16 or 32-bit arithmetic.

Name	Meaning
No_Overflow_ADD	Two's complement add with no overflow detection. Result is always same as true mathematical answer truncated to 8, 16 or 32-bits.
No_Overflow_SUB	Two's complement subtract with no overflow detection. Result is always same as true mathematical answer truncated to 8, 16 or 32-bits.
No_Overflow_MUL	Two's complement multiply with no overflow detection. Result is always same as true mathematical answer truncated to 8, 16 or 32-bits.
No_Overflow_NEG	Two's complement negate with no overflow detection. Result is always same as true mathematical answer truncated to 8, 16 or 32-bits.

5.12.3. Conversion Operations

The following group of generic functions is provided as a means for converting operands between integer and floating point formats.

Name	Meaning
FLOATi	Conversion from integer to float.
FIXR	From float to integer rounded.
FIXT	From float to integer truncated.

5.12.4. MC68881/MC68040 Floating Point Instructions

The following group of generic functions allows specific MC68881/MC68040 floating point instructions to be applied to Ada entities. The user must instantiate the generic function for the floating point types that will be used as the operand(s) and result of the operation. These generic functions have been given the same name as the assembler's name for the corresponding instruction.

For details of the operation applied by calling an instance of one of these generic functions see the *MC68881 and MC68040 User's Manuals*. Examples of their use are given in Figure 5-3. Refer to Appendix B for the signatures of all intrinsics. The available operations are shown in the following table.

Name	Meaning
FABS	Absolute Value
FSABS	Absolute Value, Single Precision
FDABS	Absolute Value, Double Precision
FACOS	Arc Cosine
FADD	Add
FSADD	Add, Single Precision
FDADD	Add, Double Precision
FASIN	Arc Sine
FATAN	Arc Tangent
FATANH	Hyperbolic Arc Tangent
FCOS	Cosine
FCOSH	Hyperbolic Cosine
FDIV	Divide
FSDIV	Divide, Single Precision
FDDIV	Divide, Double Precision
FETOX	E to the Power X
FETOXM1	E to the Power (X-1)
FGETEXP	Get Exponent
FGETMAN	Get Mantissa
FLOG10	Log Base 10
FLOG2	Log Base 2
FLOGN	Log Base E
FLOGNP1	Log Base E of X + 1
FMOD	Modulo Remainder
FMUL	Multiply
FSMUL	Multiply, Single Precision
FDMUL	Multiply, Double Precision
FNEG	Negate
FSNEG	Negate, Single Precision
FDNEG	Negate, Double Precision

FREM	IEEE Remainder
FSCALE	Scale Exponent
FSGLDIV	Single Precision Divide
FSGLMUL	Single Precision Multiply
FSIN	Sine
FSINH	Hyperbolic Sine
FSQRT	Square Root
FSSQRT	Square Root, Single Precision
FDSQRT	Square Root, Double Precision
FSUB	Subtract
FSSUB	Subtract, Single Precision
FDSUB	Subtract, Double Precision
FTAN	Tangent
FTANH	Hyperbolic Tangent
FTENTOX	Ten to Source
FTWOTOX	Two to Source

```

with Intrinsic; use Intrinsic;

function Sqrt (Source : float) return float is
  function My_Fsqrt is new FSQRT (float, float);
begin
  return My_Fsqrt (Source);
end Sqrt;

```

Figure 5-3: MC68881 Square Root Function