AD-A239 677

TION PAGE

Form Approved
OPM No. 0704-0188

Public rep... needed, a... Headquar... Managem...

...esponse, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data ...stimate 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: 11 Feb 1991 to 01Jun 1993

4. TITLE AND SUBTITLE
TeleSoft, TeleGen2 Ada Cross Development System, Version 4.1 for SUN-3 to 68k, Sun-3/480 Workstation (Host) to Motorola MVME 135-1 (MC68020, bare machine)(Target), 910125I1.11126

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, Federal Republic of Germany

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

8. PERFORMING ORGANIZATION REPORT NUMBER
IABG-VSR 090

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)*
TeleSoft, TeleGen Ada Cross Development System, Version 4.1 for SUN-3 to 68k, Ottobrunn, Germany, Sun Microsystems Sun-3/480 under Sun UNIX, Release 4.1 (Host) to Motorola MVME 135-1 (MC68020)(bare machine), ACVC 1.11.

14. SUBJECT TERMS	15. NUMBER OF PAGES
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.	16. PRICE CODE

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

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-01-25.

Compiler Name and Version: TeleGen2™ Ada Cross Development System, Version 4.1, for SUN-3 to 68k

Host Computer System: Sun Microsystems Sun-3/480 under Sun UNIX, Release 4.1

Target Computer System: Motorola MVME 135-1 (MC68020) (bare machine)

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

As a result of this validation effort, Validation Certificate #910125I1.11126 is awarded to TeleSoft. This certificate expires on 01 March 1993.

This report has been reviewed and is approved.

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

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

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

AVF Control Number: IABG-VSR 090
11 February 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910125I1.11126
TeleSoft
TeleGen2™ Ada Cross Development System
Version 4.1 for SUN-3 to 68k
Sun-3/480 Workstation =>
Motorola MVME 135-1 (MC68020, bare machine)

== based on TEMPLATE Version 91-01-10 ==

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

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-01-25.

Compiler Name and Version:	**TeleGen2™ Ada Cross Development System, Version 4.1, for SUN-3 to 68k**
Host Computer System:	**Sun Microsystems Sun-3/480 under Sun UNIX, Release 4.1**
Target Computer System:	**Motorola MVME 135-1 (MC68020) (bare machine)**

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

As a result of this validation effort, Validation Certificate #910125I1.11126 is awarded to TeleSoft. This certificate expires on 01 March 1993.

This report has been reviewed and is approved.

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

Ada Validation Organization
for 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:	TeleSoft 5959 Cornerstone Court West San Diego CA USA 92121
Ada Validation Facility:	IABG, Dept. ITE W-8012 Ottobrunn Germany
ACVC Version:	1.11
Ada Implementation:	
Ada Compiler Name and Version:	TeleGen2™ Ada Cross Development System, Version 4.1, for SUN-3 to 68K
Host Computer System:	Sun Microsystems Sun-3/480 Workstation, Sun UNIX, Release 4.1
Target Computer System:	Motorola MVME 135-1 (MC68020) (bare machine)

Customer's Declaration

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

Date: 1/23/91

TELESOFT
Raymond A. Parra
Vice President
General Counsel

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

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

1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

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

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

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

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

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

1.3 ACVC TEST CLASSES

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

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

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

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

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

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

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

1.4 DEFINITION OF TERMS

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

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [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 91-01-09.

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

2.2 INAPPLICABLE TESTS

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

The following 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)

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

C35508I..J and C35508M..N (4 tests) include enumeration representation clauses for Boolean types in which the specified values are other than (FALSE => 0, TRUE => 1); this implementation does not support a change in representation for Boolean types. (See section 2.3.)

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types; for this implementation, MACHINE_OVERFLOWS is TRUE.

C86001F recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete. For this implementation, the package TEXT_IO is dependent upon package SYSTEM.

B86001Y checks for a predefined fixed-point type other than DURATION.

CA2009C, CA2009F, BC3204C, and BC3205D check whether a generic unit can be instantiated BEFORE its generic body (and any of its subunits) is compiled. This implementation creates a dependence on generic units as allowed by AI-00408 and AI-00530 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3)

LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests) check for pragma INLINE for procedures and functions.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

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

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2107B..E (4 tests), CE2107L, CE2110B and CE2111D attempt to associate multiple internal files with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

CE2107G..H (2 tests), CE2110D, and CE2111H attempt to associate multiple internal files with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A attempt to associate multiple internal files with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page

number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 22 tests.

C35508I..J and C35508M..N (4 tests) include enumeration representation clauses for Boolean types in which the specified values are other than (FALSE => 0, TRUE => 1); this implementation does not support a change in representation for Boolean types. (See section 2.3.)

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.

BA1001A	BA2001C	BA2001E	BA3006A	BA3006B
BA3007B	BA3008A	BA3008B	BA3013A	

CA2009C, CA2009F, BC3204C, and BC3205D were graded inapplicable by Evaluation Modification as directed by the AVO. Because the implementation makes the units with instantiations obsolete (see section 2.2), the Class C tests were rejected at link time and the Class B tests were compiled without error.

CD1009A, CD1009I, CD1C03A, CD2A21C, CD2A22J, CD2A24A, and CD2A31A..C (3 tests) use instantiations of the support procedure Length_Check, which uses Unchecked_Conversion according to the interpretation given in AI-00590.
The AVO ruled that this interpretation is not binding under ACVC 1.11; the tests are ruled to be passed if they produce Failed messages only from the instantiations of Length_Check--i.e., the allowed Report.Failed messages have the general form:

" * CHECK ON REPRESENTATION FOR <TYPE_ID> FAILED."

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 a point of contact for both technical and sales information about this Ada implementation system, see:

TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121, USA
(619) 457-2700

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.

a) Total Number of Applicable Tests	3801	
b) Total Number of Withdrawn Tests	84	
c) Processed Inapplicable Tests	84	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	285	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

All I/O tests of the test suite were processed because this implementation supports 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

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 285 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic data cartidge with the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the data cartridge were loaded to a Sun computer from which they were copied via ethernet to the host computer running the ACVC tests.

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 a serial communications link, and run. The results were captured from the host computer system onto a magnetic tape.

Test output, compiler and linker listings, and job logs were captured on a magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

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 are given on the next page, which was supplied by the customer.

SUN-3/E68 Compiler Option Information

B TESTS:

ada -v -L 'testfilename'

option	description
ada	invoke TeleGen2 Ada cross compiler
-v	verbose mode
-L	generate interspersed error listing
'testfilename'	the filename being compiled

NON_B TESTS:

VME135 board:

ada -v 'testfilename'
ald -v -a 'options filename' 'mainname'

option	description
ada	invoke TeleGen2 Ada cross compiler
ald	invoke TeleGen2 Ada linker
-v	verbose mode
-a 'options filename'	use additional options from the named options file
'testfilename'	the filename being compiled
'mainname'	the name of the main compilation unit

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	200 -- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	"CCCCCCC10CCCCCCCC20CCCCCCCC30CCCCCCCC40 CCCCCCCC50CCCCCCCC60CCCCCCCC70CCCCCCCC80 CCCCCCCC90CCCCCCC100CCCCCCC110CCCCCCC120 CCCCCCC130CCCCCCC140CCCCCCC150CCCCCCC160 CCCCCCC170CCCCCCC180CCCCCCC190CCCCCC199"

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

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_646
$DEFAULT_MEM_SIZE	2147483647
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	TELEGEN2
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	ENT_ADDRESS
$ENTRY_ADDRESS1	ENT_ADDRESS1
$ENTRY_ADDRESS2	ENT_ADDRESS2
$FIELD_LAST	1000
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	3.40283E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	4.25354E+38
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	0.0
$HIGH_PRIORITY	63

```
$ILLEGAL_EXTERNAL_FILE_NAME1
                        BADCHAR*^/%

$ILLEGAL_EXTERNAL_FILE_NAME2
                        /NONAME/DIRECTORY

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.ADA")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006D1.ADA")

$INTEGER_FIRST          -32768

$INTEGER_LAST           32767

$INTEGER_LAST_PLUS_1    32768

$INTERFACE_LANGUAGE     C

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -131_073.0

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        MCI'(OP => NOP);

$MACHINE_CODE_TYPE      Opcodes

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2_147_483_648

$MIN_INT                -2147483648

$NAME                   NO_SUCH_TYPE_AVAILABLE

$NAME_LIST              TELEGEN2

$NAME_SPECIFICATION1    X2120A

$NAME_SPECIFICATION2    X2120B
```

$NAME_SPECIFICATION3	X3119A
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483647
$NEW_SYS_NAME	TELEGEN2
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	RECORD NULL; END RECORD;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	2048
$TICK	0.01
$VARIABLE_ADDRESS	VAR_ADDRESS
$VARIABLE_ADDRESS1	VAR_ADDRESS1
$VARIABLE_ADDRESS2	VAR_ADDRESS2

APPENDIX B

COMPILATION SYSTEM AND LINKER 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.

TELESOFT

TeleGen2 Ada Development System for Sun-3 to Embedded MC680X0 Targets

Compiler Command Options

NOTE-1767N-V1.1(SUN.E68) 15JAN91

Version 4.01

TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-9819
(619) 457-2700
(Contractor)

Table of Contents

Compiler Command Summary

This document describes the options available for invoking the TeleGen2 compiler (via the *ada* command) and the TeleGen2 linker (via the *ald* command).

1.1. ada (Ada Compiler)

The TeleGen2 Ada Compiler is invoked by the *ada* command. Unless you specify otherwise, the front end, middle pass, and code generator are executed each time the compiler is invoked.

Before you can compile, you must (1) make sure you have access to TeleGen2, (2) have a library file available, and (3) create a sublibrary.

The syntax of the command to invoke the Ada compiler is shown below.

```
ada [<option>...] <input_spec>
```

`<option>` One of the options available with the command. Compiler options fall into five categories.

Library search	-l(ibfile, -t(emplib
Execution control	Abort after errors: -E(rror_abort Compile, then link: -m(ain Perform minimal recompilation: -R(ecomp_min Update library for multiple files: -u(pdate_invoke
Output control	Bind but do not link: -b(ind_only Run front end only: -e(rrors_only Enable debugging: -d(ebug Suppress checks: -i(nhibit Keep intermediates: -k(eep_intermediates Keep source: -K(eep_source Inline code: -O(ptimize, -G(raph, -I(nline Include execution profile: -x(ecution_profile
Listing control	Output source plus errors: -L(ist Output errors: -F(ile_only_errs, -j(oin Error context: -C(ontext Output assembly: -S("asm_listing"
Other	-q(uiet, -V(space_size, -v(erbose

<input_spec> The Ada source file(s) to be compiled. It may be:

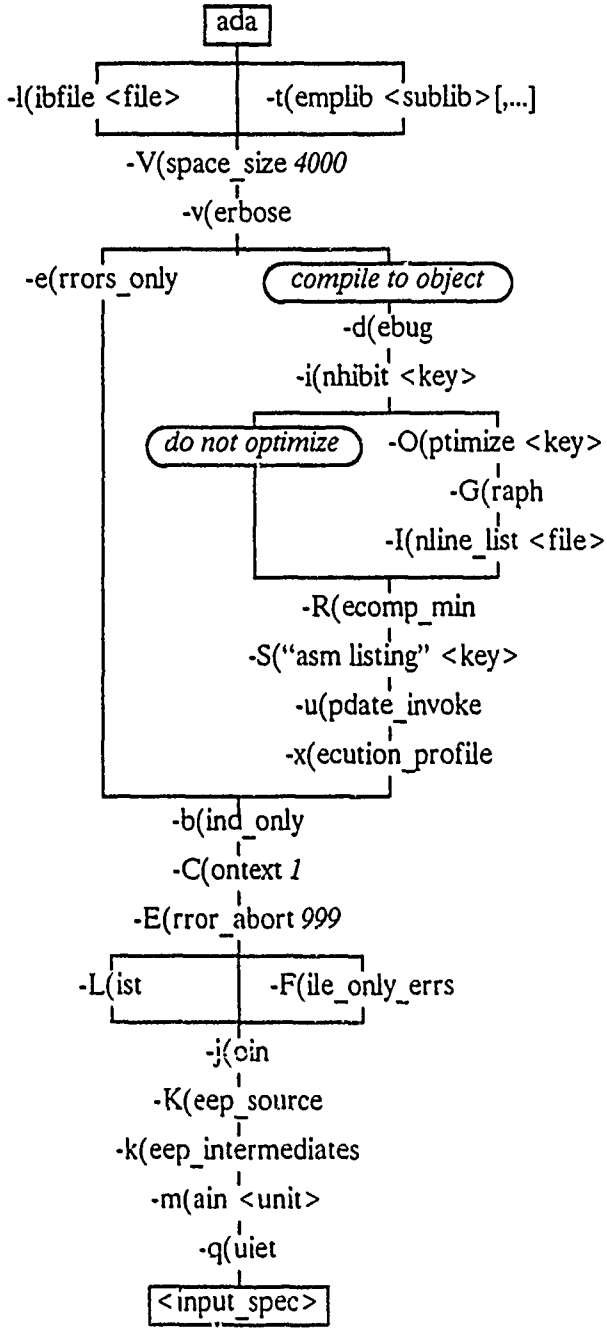
- One or more Ada source files, for example:

```
/user/john/example
Prog_A.text
ciosrc/calc_mem.ada
calcio.ada  myprog.ada
*.ada
```

- A file containing names of files to be compiled. Such a file must have the extension ".ilf".
- A combination of the above.

Compiler defaults. Compiler defaults are set for your convenience. In most cases you will not need to use additional options; a simple "ada <input_spec>" is sufficient. However, options are included to provide added flexibility. You can, for example, have the compiler quickly check the source for syntax and semantic errors but not produce object code [-e(rrors_only] or you can compile, bind, and link a main program with a single compiler invocation [-m(ain]. Other options are provided for other purposes.

The options available with the *ada* command, and the relationships among them, are illustrated in the following figure.


ada
-l(ibfile <file>
-t(emplib <sublib>[,...]
-V(space_size 4000
-v(erbose
-e(rrors_only
compile to object
-d(ebug
-i(nhibit <key>
do not optimize
-O(ptimize <key>
-G(raph
-I(nline_list <file>
-R(ecomp_min
-S("asm listing" <key>
-u(pdate_invoke
-x(ecution_profile
-b(ind_only
-C(ontext 1
-E(rror_abort 999
-L(ist
-F(ile_only_errs
-j(oin
-K(eep_source
-k(eep_intermediates
-m(ain <unit>
-q(uiet
<input_spec>

Below are some basic examples that show how the command is used.

1. (No options) The following command compiles the file *sample.ada*, producing object code that is stored in the working sublibrary.

   ```
   ada sample.ada
   ```

   In this example, the working sublibrary is the first sublibrary listed in *liblst.alb*. No listings are produced, no progress messages are output, no intermediate forms are retained, and so forth. In other words, it's the simplest example of compilation.

2. The following command compiles *sample.ada* as above, but because we used the *-L* option, a listing file, *sample.l*, is output to the working directory. The listing file shows the source code, errors (if any), the number of lines compiled, plus other information.

   ```
   ada -L -v sample.ada
   ```

   Progress messages are output during compilation because we used the *-v* option.

The options available with *ada* are presented below in alphabetical order.

-b(ind_only

The -b(ind_only option instructs the compiler to invoke the binder but not the linker when -m(ain is used on the *ada* command line. (The -main option calls *ald* when compilation is complete.) The *-b* option is useful when you have adapted your own linker and want to use the adapted linker instead of the linker provided.

-C(ontext

When an error message is sent to *stderr*, it is helpful to include the lines of the source program that surround the line containing the error. These lines provide a context for the error in the source program and help to clarify the nature of the error. The *-C* option controls the number of source lines that surround the the error. The format of the option is

```
-C <n>
```

where <n> is the number of source context lines output for each error. The default for <n> is 1. This parameter specifies the total number of lines output for each error (including the source line that contains the error). The first context line is the one immediately before the line in error; other context lines are distributed before and after the line in error.

-d(ebug

To use the debugger, you must compile and link with the -d(ebug option. This is to make sure that a link map and debugging information are put in the Ada library for use by the debugger. Using -d(ebug ensures that the intermediate forms needed for debugging and the debugging information for secondary units are not deleted.

Performance note. While the compilation time overhead generated by the use of -d(ebug is minimal, retaining this optional information in the Ada library increases the space overhead. To see if a unit has been compiled with the -d(ebug option, use the *als* command with the -X(tended option. Debugger information exists for the unit if the "dbg_info" attribute appears in the listing for that unit.

-E(rror_abort

The -E(rror_abort option allows you to set the maximum number of errors (syntax errors and semantic errors) that the compiler can encounter before it aborts. This option can be used with all other compiler options.

The format of the option is

```
-E <n>
```

where <n> is the maximum number of errors allowed (combined counts of syntax errors and semantic errors). The default is 999; the minimum is 1. If the number of errors becomes too great during a compilation, you may want to abort the compilation by typing <control>-C.

-e(rrors_only

The -e(rrors_only option instructs the compiler to perform syntactic and semantic analysis of the source program without generating Low Form and object code. That is, it calls the front end only, not the middle pass and code generator. This means that only front end errors are detected and that only the High Form intermediates are generated. Unless you use the -k(eep_intermediates option along with *-e*, the High Form intermediates are deleted at the end of compilation; in other words, the library is not updated.

The -e(rrors_only option is typically used during early code development where execution is not required and speed of compilation is important.

Since only the front end of the compiler is invoked when *-e* is used, *-e* is incompatible with *ada* options that require processing beyond the front end phase of compilation. Such options include, for example, -O(ptimize and -d(ebug. If *-e* is not used (the default situation), the source is compiled to object code (providing no errors are found). Object code is generated for the specification and body and inserted into the working sublibrary.

-F(ile_only_errs

The -F(ile_only_errs option is one of the five listing control options available with the *ada* command (the others are -C, -L, -j, and -S). The *-F* option is used to produce a listing containing only the errors generated during compilation; source is not included. The output is sent to <file>.l, where <file> is the base name of the input file. If input to the *ada* command is an input-list file (<file>.ilf), a separate listing file is generated for each source file listed in the input file. Each resulting listing file has the same name as the parent file, except that the extension ".l" is appended. *-F* is incompatible with *-L*.

-G(raph

The -G(raph option is valid only with -O(ptimize.

This option generates a call graph for the unit being optimized. The graph is a file containing a textual representation of the call graph for the unit being optimized. For each subprogram, a list is generated that shows every subprogram called by that subprogram. By default, no graph is generated.

The graph is output to a file named <unit>.grf, where <unit> is the name of the unit being optimized.

-I(nline_list

The -I(nline_list option is valid only with -O(ptimize.

This option allows you to inline subprograms selectively. The format of the option is

```
-I <file>
```

where <file> is a file that contains subprogram names. The format of the option is shown below.

- All visible-subprogram names, each separated by a comma or line feed *then*
- A semicolon or a blank line *then*
- All hidden-subprogram names, each separated by a comma or line feed

Tabs and comments are not allowed. If there is no semicolon or blank line, the subprograms are considered to be visible. If you have no visible units to inline, use a semicolon to mark the beginning of the hidden-subprogram list. Inline lists are commonly set up with one name per line.

Each subprogram name in the list is in the form shown below.

```
[<unit>.]<subprogram>
```

The unit name indicates the location of the subprogram declaration, not the location of its body. If a unit name is not supplied, any matching subprogram name (regardless of the location of its declaration) will be affected. For

example, the list

```
test; testing.test
```

indicates that all subprograms named Test should be marked for inlining except for those declared in either the specification or the body of the compilation unit Testing.

The first list of subprograms will be processed as if there had been a pragma Inline in the source for them. The second list of subprograms will negate any Inline pragmas (including those applied by the first list) and will also prevent any listed subprograms from being automatically inlined (see **A/a** suboption pair, in the discussion of -O(ptimize).

The ability to exempt otherwise qualified subprograms from automatic inlining gives you greater control over optimization. For example, a large procedure called from only one place within a case statement might overflow the branch offset limitation if it were inlined automatically. Including that subprogram's name in the second list in the list file prevents the problem and still allows other subprograms to be inlined.

Since the Low Form contains no generic templates, pragma Inline must appear in the source in order to affect all instantiations. However, specific instantiations can be affected by the inline lists. The processing of the names is case insensitive.

If you do not use *-I*, the optimizer automatically inlines any subprogram that is: (1) called from only one place, (2) considered small by the optimizer, or (3) tail recursive.

-i(nhibit

The -i(nhibit option is equivalent to adding pragma Suppress to the beginning of the declarative part of each compilation unit in a file. Specifically, -i(nhibit allows you to suppress, within the generated object code, certain run-time checks, source line references, and subprogram name information. The format of the option is

```
-i <key>
```

where <key> is one or more of the single-letter suboptions listed below. When more than one suboption is used, the suboptions appear together with no separators. For example, "-i lnc".

l [line_info] Suppress source line information in object code.

By default, the compiler stores source line information in the object code. However, this introduces an overhead of 6 bytes for each line of source that causes code to be generated. Thus, a 1000-line package may have up to 6000 bytes of source line information.

When source line information is suppressed, exception tracebacks indicate the offset of the object code at which the exception occurs instead of the source line number.

n [name_info] Suppress subprogram name information in object code.

By default, the compiler stores subprogram name information in object code. For one compilation unit, the extra overhead (in bytes) for subprogram name information is the total length of all subprogram names in the unit (including middle pass-generated subprograms), plus the length of the compilation unit name. For space-critical applications, this extra space may be unacceptable.

When subprogram name information is suppressed, the traceback indicates the offsets of the subprogram calls in the calling chain instead of the subprogram names.

c [checks] Suppress run-time checks — elaboration, overflow, storage access, discriminant, division, index, length, and range checks.

While run-time checks are vital during development and are an important asset of the language, they introduce a substantial overhead. This overhead may be prohibitive in time-critical applications.

a [all] Suppress source line information, subprogram name information, and run-time checks. In other words, **a** (=inhibit all) is equivalent to **lnc**.

Below is a command that tells the compiler to inhibit the generation of source line information and run-time checks in the object code of the units in *sample.ada*.

```
ada -v -i lc sample.ada
```

-j(oin

The -j(oin option is one of five listing-control options available with the *ada* command (the others are -C, -L, -S, and -F). The *-j* option writes any errors that are generated during compilation back into the source file. The errors appear in the file as Ada comments. This option allows you to comment the source file with the errors that are generated at compile time. These comments can help facilitate debugging and commenting your code. Unlike *-L*, *-S*, and *-F*, the *-j* option does not produce a separate listing, since the information generated is written into the source file.

-K(eep_source

This option tells the compiler to take the source file and store it in the Ada library. When you need to retrieve your source file later, use the *axt* command.

-k(eep_intermediates

The -k(eep_intermediates option allows you to retain certain intermediate code forms that the compiler otherwise discards.

By default, the compiler deletes the High Form and Low Form intermediate representations of all compiled secondary units from the working sublibrary. Deletion of these intermediate forms can significantly decrease the size of sublibraries — typically 50% to 80% for multi-unit programs. On the other hand, some of the information within the intermediate forms may be required later. For example, High Form is required if the unit is to be referenced by the Ada cross-referencer (*axr*). In addition, information required by the debugger and the optimizer must be saved if these utilities are used.

To verify that a unit has been compiled with the -k(eep_intermediates option (has not been "squeezed"), use the *als* command with the -X(tended option. A listing will be generated that shows whether the intermediate forms for the unit exist. A unit has been compiled with -k(eep if the attributes high_form and low_form appear in the listing for that unit.

-L(ist

The -L(ist option is one of five listing control options available with the *ada* command (the others are -C, -F, -j, and -S). The *-L* option instructs the compiler to output a listing of the source being compiled, interspersed with error information (if any). The listing is output to <file>.l, where <file> is the name of the source file (minus the extension). If <file>.l already exists, it is overwritten.

If input to the *ada* command is an input-list file (<file>.ilf), a separate listing file is generated for each source file listed in the input file. Each resulting listing file has the same name as the parent file, except that the extension ".l" is appended. Errors are interspersed with the listing. If you do not use -L (the default situation), errors are sent to *stdout* only; no listing is produced. *-L* is incompatible with *-F*.

-l(ibfile

The -l(ibfile option is one of the two library-search options; the other is -t(emplib. Both of these options allow you to specify the name of a library file other than the default, *liblst.alb*. The two options are mutually exclusive.

The format of the -l(ibfile option is

```
-l <file>
```

where <file> is the name of a library file, which contains a list of sublibraries and optional comments. The file must have the extension ".alb". The first sublibrary is always the working sublibrary; the last sublibrary is generally the basic run-time sublibrary (rtl.sub). Note that comments may be included in a library file and that each sublibrary listed must have the extension ".sub".

-m(ain

This option tells the compiler that the unit specified with the option is to be used as a main program. After all files named in the input specification have been compiled, the compiler invokes the Ada Linker to bind and link the program with its extended family. By default an "execute form" (EF) load module named <unit>.ef is left in the current directory.

The format of the option is

```
-m <unit>
```

where <unit> is the name of the main unit for the program. If the main unit has already been compiled, make sure that the body of the main unit is in the current working directory.

Note: You may specify options that are specific to the linker on the *ada* command line if you use the -m(ain option. In other words, if you use *-m*, you may also use *-o*, *-a*, or any of the other *ald* options. For example, the command

```
ada -v -m welcome -o new.ef -a link.opt sample.ada
```

instructs the compiler to compile the Ada source file *sample.ada*, which contains the main program unit Welcome. After the file has been compiled, the compiler calls the Ada linker, passing to it the *-o* and *-a* options with their respective arguments. The *-a* option tells the linker to use the commands specified in the options file *link.opt* to direct the linking process; an options file is required for linking. The linker produces an "execute form" load module of of the unit, placing it in file *new.ef* as requested by the linker's *-o* option.

If you use an option with -m(ain that is common to both *ada* and *ald*, the option serves for both compiling and linking. For example, using *-S* with "ada -m" produces two assembly listings--one from compilation, one from elaboration.

-O(ptimize

The optimizer operates on Low Form, the intermediate code representation that is output by the middle pass of the compiler.

When used on the *ada* command line, -O(ptimize causes the compiler to

invoke the global optimizer, which optimizes the Low Form generated by the middle pass for the unit being compiled. The code generator takes the optimized Low Form as input and produces more efficient object code.

Note: We recommend that you do not attempt to compile with optimization until the code being compiled has been fully debugged and tested because using the optimizer increases compilation time.

The format of the option is

```
-O  <suboptions>
```

where <suboptions> is a string composed of one or more of the single-letter suboptions listed below. <suboptions> is required.

The suboptions may appear in any order (later suboptions supersede earlier suboptions). The suboption string must not contain any characters (including spaces or tabs) that are not valid suboptions. Examples of valid suboptions are:

```
-O pRiA
-O pa
```

Table of optimizer suboptions

P	[optimize with parallel tasks] Guarantees that none of subprograms being optimized will be called from parallel tasks. **P** allows data mapping optimizations to be made that could not be made if multiple instances of a subprogram were active at the same time.
p	[optimize without parallel tasks] Indicates that one or more of the subprograms being optimized might be called from parallel tasks. This is a "safe" suboption. DEFAULT
R	[optimize with external recursion] Guarantees that no interior subprogram will be called recursively by a subprogram exterior to the unit/collection being optimized. Subprograms may call themselves or be called recursively by other subprograms interior to the unit/collection being optimized.
r	[optimize without external recursion] Indicates that one or more of the subprograms interior to the unit/collection being optimized could be called recursively by an exterior subprogram. This is a "safe" suboption. DEFAULT
I	[enable inline expansion of subprograms] Enables inline expansion of those subprograms marked with an Inline pragma or introduced by the compiler. DEFAULT
i	[disable inline expansion] Disables all inlining.
A	[enable automatic inline expansion] If the **I** suboption is also in effect (**I** is the default), **A** enables automatic inline expansion of any subprogram not marked for inlining; that is, any subprogram that is (1) called from only one place, (2) considered to be small by the optimizer, or (3) tail recursive. If **i** is used as well, inlining is prohibited and **A** has no effect. DEFAULT
a	[disable automatic inline expansion] Disables automatic inlining. If **i** is used as well, inlining is prohibited and **a** has no effect.
M	[perform maximum optimization] Specifies the maximum level of optimization; it is equivalent to "**PRIA**". This suboption assumes that the program has no subprograms that are called recursively or by parallel tasks.
D	[perform safe optimizations] Specifies the default "safe" level of optimization; it is equivalent to "**prIA**". It represents a combination of optimizations that is safe for all compilation units, including those with subprograms that are called recursively or by parallel tasks.

Below are some examples showing the use of *ada* with -O(ptimize.

1. The command below compiles and optimizes a single unit in file *optimize.ada*.

```
ada -O D -v optimize.ada
```

   It uses "safe" optimization (**D**), since the unit may have subprograms called recursively or by parallel tasks.

2. The command below compiles and optimizes individually a series of units listed in the input list *protype1.ilf*.

```
ada -O PrIa -v protype1.ilf
```

   This command tells the compiler that the units have subprograms called recursively (**r**) but none called by parallel tasks (**P**). It also tells the compiler that pragma Inline marks subprograms to be inlined (**I**), but that automatic inlining is not desired (**a**).

3. The command below requests maximum optimization (**M**), because the one-unit program in *alpha_sort.ada* has no subprograms called recursively or by parallel tasks.

```
ada -O M -v alpha_sort.ada
```

-q(uiet

The -q(uiet option allows you to suppress information messages that are output as the command executes. The option is particularly useful during optimization, when a large number of such messages are likely to be output.

-R(ecomp_min

The -R(ecomp_min (recompilation minimization) option allows you to recompile source files without updating the library. Use of the option eliminates the need for compiling dependent units when you've made an insignificant change to another unit.

Examples of "insignificant" changes are:

- Changing a comment
- Changing the case of identifiers, including reserved words
- Changing the case of letters within numeric literals
- Reformatting code or comments by adding white space

To use the -R(ecomp_min option, you must have previously compiled the file with the -K(eep_source option. This is so the compiler can determine the type of changes that have been made.

Restriction

In the current release, if a unit contains inline or instantiation dependencies, all dependent units must be recompiled if the original unit is recompiled—even if the change to the original unit is insignificant.

-S("asm_listing"

The *-S* option is one of five listing control option available with the *ada* command (the others are -C, -F, -j, and -L). The *-S* option instructs the compiler to generate an assembly listing intermixed with source code in the form of comments. The output is sent to <unit>.s (for a body) or <unit>_.s (for a specification) or both (for a main program), where <unit> is the name of the unit in the user-supplied source file. The assembly listings are put in the working directory. If more than one unit is in the file, separate listings are generated for each unit. The format of the option is

```
-S <key>
```

where <key> is either "e" or "a".

- **e** [extended] Generate a paginated, extended assembly listing that includes code offsets and object code.

- **a** [assembler] Generate a listing that can later be used as input to an assembler.

The argument of the *-S* option, <key>, is mandatory, so either "e" or "a" must be used with *-S*.

The listing generated consists of assembly code intermixed with source code as comments. If input to the *ada* command is an input-list file (<file>.ilf), a separate assembly listing file is generated for each unit contained in each source file listed in the input file. Since *-S* is also an *ald* option, if you use *-S* along with -m(ain, an assembly listing is also output during the binding process.

-t(emplib

The -t(emplib option is one of the two library-search options; the other is -l(ibfile. Both of these options allow you to select a set of sublibraries for use during the time in which the command is being executed. The two options are mutually exclusive.

The format of the -t(emplib option is

```
-t <sublib>[,<sublib>] ...
```

where <sublib> is the name of a sublibrary. The name must include the ".sub" extension; it must also be prefaced by a path name if the sublibrary is in a directory other than the current directory. The first sublibrary listed is the working sublibrary by definition. If more than one sublibrary is listed, the

names must be separated by a comma. Single or double quotes may be used as delimiters.

The argument string of the -t(emplib option is logically equivalent to the names of the sublibraries listed in a library file. So instead of using

```
-l worklib.alb
```

you could use -t(emplib and specify the names of the sublibraries listed in *worklib.alb* as the argument string.

-u(pdate_invoke

The -u(pdate_invoke (short for "update_after_invocation") option tells the compiler to update the working sublibrary only after all files submitted in that invocation of *ada* have compiled successfully. The option is therefore useful only when compiling multiple source files.

If the compiler encounters an error while *-u* is in effect, the library is not updated, even for files that compile successfully. Furthermore, all source files that follow the file in error are compiled for syntactic and semantic errors only.

If you do not use the -u(pdate_lib option, the library is updated each time one of the files submitted has compiled successfully. In other words, if the compiler encounters an error in any unit within a single source file, all changes to the working sublibrary for the erroneous unit and for all other units in that file are discarded. However, library updates for units in previous or remaining source files are unaffected.

Since using *-u* means that the library is updated only once, a successful compilation is faster with *-u* than without it. On the other hand, if the compiler finds an error when you've used *-u*, the library is not updated even when the other source files compile successfully. The implication is that it is better to avoid using *-u* unless your files are likely to be error free.

-V(space_size

The -V(space_size option allows you to specify the size of the working space for TeleGen2 components that operate on library contents. The format of the option is

```
-V <value>
```

where the option parameter is specified in 1-Kbyte units; it must be an integer value. The default value is 4000. The upper limit is 2,097,152. Larger values generally improve performance but increase physical memory requirements.

-v(erbose

The -v(erbose option is used to display messages that inform you of the progress of the command's execution. Such messages are prefaced by a banner that identifies the component being executed. If *-v* is not used, no progress messages are output.

-x(ecution_profile

The -x(ecution_profile option is used to obtain a profile of how a program executes. The option is available with *ada*, *ald*, and *aopt*.

Using *-x* with *ada* or *aopt* causes the code generator to insert special run-time code into the generated object. Using *-x* with *ald* causes the binder to link in the run-time support routines that will be needed during execution.

Important. If you have compiled any code in a program with the -x(ecution_profile option, you must also use *-x* when you bind and link the program.

1.2. ald (Ada Linker)

The TeleGen2 Ada Compiler produces object code from Ada source code and stores it in the Ada sublibrary. The TeleGen2 Ada Linker takes Ada object code and non-Ada imported object code from Ada sublibraries and creates either linked output modules or partially linked object form (OF) modules that can be used as input to subsequent linking operations. The terms "OF modules" and "OFMs" are interchangeable.

The linker operates in two phases: the binding phase and the linking phase. In the binding phase, the linker binds together all necessary Ada units, creating elaboration code that is stored in the sublibrary. In the linking phase, the linker combines the elaboration code, the appropriate Ada object modules, and any OFM, environment, or imported non-Ada object specified in the linker option file to produce either an executable load module or a new OFM.

The linker is invoked by the *ald* command; it can also be invoked with the -m(ain option of the *ada* command. In the latter case the compiler passes appropriate options to the linker to direct its operation. The syntax of the command is shown below.

```
ald [<option>...] unit
```

`<option>` One of the options available with the command.

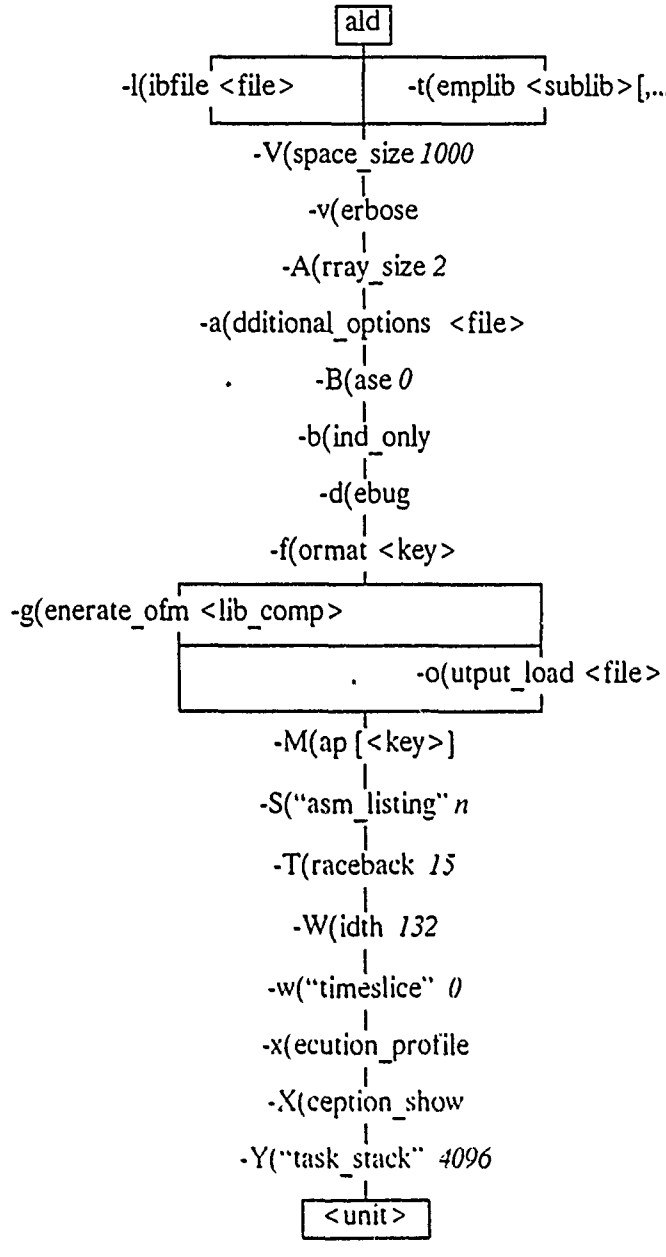
`<unit>` The name of the main unit of the Ada program to be linked. If the name of the unit is not provided on the command line, the unit is specified with the INPUT option in a linker option file. *Important:* When using the *ald* command, the body of the main unit to be prelinked must be in the working sublibrary.

In the simplest case, *ald* takes two arguments - the name of the main unit of the Ada program structure that is to be linked and the name of a linker option file - and produces one output file, the complete load module produced by the linking process. The load module is placed in the directory from which *ald* was executed, under the name of the main unit used as the argument to *ald*. For example, the command

```
ald -a link.opt main
```

links the object modules of all the units in the extended family of unit Main, including any user-specified modules in the linker option file *link.opt*. The resulting load module will be named "main.ef", which is in the TeleGen2 proprietary output format that can be used as input to the downloader (*adwn* command).

The options available with the command, and the relationships among them, are shown in the figure below.



Linker directives are communicated to the linker as options on the command line or as options entered via an option file. *Command-line options* are useful for controlling options that you are likely to change often. The default option settings are designed to allow for the simplest and most convenient use of the linker. Command-line options are discussed below. *Option-file options* are for specifying more complicated linker options, such as the specification of memory locations for

specific portions of the code or data for a program.

The options available with *ald* are presented below in alphabetical order.

-A(rray_size

This option specifies the amount of internal buffer space, in Kbytes, to be allocated for the linker. The format of the option is:

```
-A <value>
```

where <value> is a value between 1 and 10. The default is 2. Use this option only as recommended by Customer Support.

-a(dditional_options

The *-a* option specifies that the linker is to process additional options obtained from a linker option file. The format is

```
-a <file>
```

where <file> is a valid file specification and represents a file containing linker options. If no extension is given, the linker uses the default extension ".opt".

-B(ase

This option is used to specify the start location of the linked output. The linker will locate non-absolute control sections in consecutive memory locations. All control sections are word aligned on the MC680X0. The format is

```
-B <addr>
```

where <addr> is a valid MC680X0 address. The address can be specified as a decimal (%Ddecimal), a hexadecimal (%Xhex), or a hexadecimal-based literal in Ada syntax (16#hex#). The default is hexadecimal (%Xhex).

If you specify neither the *-B* option nor an option file LOCATE command and the link is complete, the linker uses the default location value of address 0.

The *-B* option governs the location for any code, constant, or data section not covered by an option file LOCATE command. This option does not supercede any LOCATE options. -B(ase is equivalent to a LOCATE option with no control section or component name specified.

-b(ind_only

The -b(ind_only option instructs the compiler to not invoke the link phase--in other words, to generate elaboration code only. This option is particularly useful when you have adapted your own linker and want to use it in place of the TeleGen2 linker.

-d(ebug

This option controls the generation of debug symbol information for use with the debugger. A program that is to be run with the debugger must be linked with the -d(ebug option. If supported by the chosen load module format, -d(ebug may also cause symbol information to be output in the load module. The option is ignored if -g(enerate_ofm is used. In the standard configuration of the TeleGen2 system, none of the outputs support symbol information in the load module.

The default situation is that no debug information is produced.

-f(ormat

The -f(ormat option specifies the format of the output module. The format of the option is

```
-f <key>
```

where <key> is one of the following; <key> is required.

E	-- *Execute Form (the default)*
S	-- *S-records; suitable for use as input to Motorola-compatible*
	-- *simulators and monitors*
U	-- *User-specified format*
I	-- *IEEE-695*

If *-f* is not used, E(xecute_form is produced. Execute Form is the default output format generated by the linker and is suitable for use as input to the downloader/receiver.

-g(enerate_ofm

This option specifies that one output of the linker is to be linked OF. Linked OF is suitable for incomplete modules and can be used subsequently as input to the Ada linker. The linked OF is put into the library as an OFM ("ofm" in listings). The format of the option is

```
-g <lib_comp>
```

where <lib_comp> is the name of a library component.

If an OFM library component with the specified name already exists in the current working sublibrary, that component is deleted and replaced by the new output.

You may use *-g* in conjunction with the -a(dditional_options option. However, any format or name present in the option file is superceded by a format and name specified on the command line. You may request an OFM (via *-g*) *instead of* the default Execute Form or *in addition to* a load module format. To obtain both an OFM and a load module, use both the *-g* and *-o* options.

-l(ibfile

The -l(ibfile option is one of the two library-search options; the other is -t(emplib. Both of these options allow you to specify the name of a library file other than the default, *liblst.alb*. The two options are mutually exclusive.

The format of the -l(ibfile option is

```
-l <file>
```

where <file> is the name of a library file, which contains a list of sublibraries and optional comments. The file must have the extension ".alb". The first sublibrary is always the working sublibrary; the last sublibrary is generally the basic run-time sublibrary (rtl.sub). Note that comments may be included in a library file and that each sublibrary listed must have the extension ".sub".

-M(ap

This option is used to request and control a link map listing. The link map listing is sent to

```
<unit>.map
```

where <unit> is the name of the main program unit (if present), the name specified as the command line parameter, or the name specified as the first INPUT option, modified as necessary to form a valid UNIX file specification. The format of the option is

```
-M [<key>]
```

where <key> is one or more of the following:

i	(image) Generates a memory image listing in addition to the map listing. The linker writes the image listing to the same file as the link map listing. This is the only optional section of the listing.
e	(excluded) Inserts a list of excluded subprograms into the link map listing.
l	(locals) Includes local symbols in the link map symbol listing.

If more than one of the above suboptions is used, they must appear together, with no spaces. For example:

```
-M iel
```

A -M(ap option specified on the command line supercedes a MAP command in an option file.

-o(utput_load

The -o(utput_load option is used primarily to specify the file name for the load module output created by the linker. The format is

```
-o <file>
```

where <file> is the specification for the output file. If <file> does not include an extension, the linker will append an extension appropriate to the load module format chosen via the -f(ormat option.

```
.ef       – Execute Form (the default)
.sr       – S-records
.ieee     – IEEE-695 format
.<other>  – User-specified format
```

You may use *-o* with -g(enerate_ofm, which specifies that linked OF is to be produced. In this case, both linked OF and a load module in the format specified with the *-f* option will be produced. If *-f* is not used, an EF module is produced by default.

You may use the *-o* option in conjunction with the -a(dditional_options option, which directs the linker to use the options in the option file specified. However, any output file specification present in the option file is superceded by a command-line specification.

-S("asm_listing"

The *-S* option is used to output an assembly listing from the elaboration process. The output is put in a file, <unit>__M.s (two underscores), where <unit> is the name of the main unit being linked. The format of the option is

```
-S <key>
```

where <key> is either "e" or "a".

e [extended] Generate a paginated, extended assembly listing that includes code offsets and object code.

a [assembler] Generate a listing that can later be used as input to an assembler.

The argument of the *-S* option, <key>, is mandatory, so either "e" or "a" must be used with *-S*.

-T(raceback

The -t(raceback option allows you to specify the callback level for tracing a run-time exception that is not handled by an exception handler. The format of the option is

```
-T <n>
```

where <n> is the number of levels in the traceback call chain. The default is 15.

When an exception occurs, the run-time support system stores the history in a preallocated block of memory. Since the size of this block is determined by the *-T* option, setting this value to a large number can introduce objectionable overhead in deeply nested, time-critical code. You may wish to make this value smaller for well-tested programs.

-t(emplib

The -t(emplib option is one of the two library-search options; the other is -l(ibfile. Both of these options allow you to select a set of sublibraries for use during the time in which the command is being executed. The two options are mutually exclusive.

The format of the -t(emplib option is

```
-t <sublib>[,<sublib>] ...
```

where <sublib> is the name of a sublibrary. The name must include the ".sub" extension; it must also be prefaced by a path name if the sublibrary is in a directory other than the current directory. The first sublibrary listed is the working sublibrary by definition. If more than one sublibrary is listed, the names must be separated by a comma. Single or double quotes may be used as delimiters.

The argument string of the -t(emplib option is logically equivalent to the names of the sublibraries listed in a library file. So instead of using

```
-l worklib.alb
```

you could use -t(emplib and specify the names of the sublibraries listed in *worklib.alb* as the argument string.

-W(idth

This option specifies the number of characters per line in the link map. The format of the option is

```
-W <value>
```

where <value> is either 80 or 132. The default is 132.

-w("timeslice"

The *-w* option allows you to specify the slice of time, in milliseconds, in which a task is allowed to execute before the run time switches control to another ready task of equal priority. This timeslicing activity allows for periodic round-robin scheduling among equal-priority tasks. The format of the option is

```
-w <value>
```

where <value> is the timeslice quantum in milliseconds. The default is 0 (i.e., timeslicing is disabled). Please note that no run-time overhead is incurred when timeslicing is disabled.

-V(space_size

The -V(space_size option allows you to specify the size of the working space for TeleGen2 components that operate on library contents. The format of the option is

```
-V <value>
```

where the option parameter is specified in 1-Kbyte units; it must be an integer value. The default value is 4000. The upper limit is 2,097,152. Larger values generally improve performance but increase physical memory requirements.

-v(erbose

The -v(erbose option is used to display messages that inform you of the progress of the command's execution. Such messages are prefaced by a banner that identifies the component being executed. If *-v* is not used, no progress messages are output.

-X(ception_show

By default, unhandled exceptions that occur in tasks are not reported; instead, the task terminates silently. The *-X* option allows you to specify that such exceptions are to be reported. The output is similar to that displayed when an unhandled exception occurs in a main program.

-x(ecution_profile

The -x(ecution_profile option is used to obtain a profile of how a program executes. The option is available with *ada*, *ald*, and *aopt*.

Using *-x* with *ada* or *aopt* causes the code generator to insert special run-time code into the generated object. Using *-x* with *ald* causes the binder to link in the run-time support routines that will be needed during execution. These run-time support routines record the profiling data in memory during program execution and then write the data to two host files, *profile.out* and *profile.dic*, via the serial download line as part of program termination. The files can then be used to produce a listing that shows how the program executes.

Important. If you have compiled any code in a program with the -x(ecution_profile option, you must also use *-x* when you bind and link the program.

-Y("task_stack"

The -*Y* option is one of the two *ald* options by which you can alter the size of the task stack (the other is -y). In the absence of a representation specification for task storage_size, the run time will allocate 4096 bytes of storage for each executing task. -*Y* specifies the size of the basic task stack. The format of the option is:

```
-Y <value>
```

where <value> is the size of the task stack in 8-bit bytes. The default is 4096. A representation specification for task storage size overrides a value supplied with this option.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

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

SUN-3/E68 PACKAGE STANDARD INFORMATION

For this target system the numeric types and their properties are as follows:

INTEGER:

size = 16
first = -32768
last = +32767

SHORT_INTEGER:

size = 8
first = -128
last = +127

LONG_INTEGER:

size = 32
first = -2147483648
last = +2147483647

FLOAT:

size = 32
digits = 6
'first = -1.70141E+38
'last = +1.70141E+38
machine_radix = 2
machine_mantissa = 24
machine_emin = -125
machine_emax = +128

LONG_FLOAT:

size = 64
digits = 15
'first = -8.98846567431158E+307
'last = +8.98846567431158E+307
machine_radix = 2
machine_mantissa = 53
machine_emin = -1021
machine_emax = +1024

DURATION:

size = 32
delta = 6.10351562500000E-005
first = -86400
last = +86400

TELESOFT

TeleGen2 Ada Development System for Sun-3 to Embedded MC680X0 Targets

LRM Appendix F Information

NOTE-1768N-V1.1(SUN.E68) 15JAN91

Version 4.01

TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-9819
(619) 457-2700
(Contractor)

Table of Contents

LRM Appendix F for TeleGen2

The Ada language definition allows for certain target dependencies. These dependencies must be described in the reference manual for each implementation, in an "Appendix F" that addresses each point listed in LRM Appendix F. Table 1-1 constitutes Appendix F for this implementation. Points that require further clarification are addressed in sections referenced in the table.

Table 1-1. LRM Appendix F for TeleGen2

(1) Implementation-Dependent Pragmas	(a) Implementation-defined pragmas: Comment, Images, Interface_Information, Interrupt, Linkname, No_Suppress, Preserve_Layout, and Suppress_All (refer to Section 1). (b) Predefined pragmas with implementation- dependent characteristics: • Interface (assembly, Pascal, C, and Fortran) • List and Page (in context of source/error compiler listings.) • Pack. *Other supported predefined pragmas:* Controlled Shared Suppress Elaborate Priority Inline *Predefined pragmas partly supported* Memory_Size Storage_Unit System_Name These pragmas are allowed if the argument is the same as the value specified in the System package. *Not supported:* Optimize
(2) Implementation-Dependent Attributes	The predefined attribute 'Address is not supported for packages. 'Extended_Image 'Extended_Fore 'Extended_Value 'Subprogram_Value 'Extended_Width 'Address 'Extended_Aft 'Offset (in MCI) 'Extended_Digits Refer to Section 2 for information on the implementation-defined extended attributes listed above.
(3) Package System	Refer to Section 3.
(4) Restrictions on Representation Clauses	Supported except as indicated in the following (LRM 13.2 - 13.5). Pragma Pack is supported, except for dynamically sized components.

------- *Continued on next page* -------

Table 1-1. LRM Appendix F for TeleGen2 (Contd)

------- *Continued from previous page* -------	
(5) Implementation-Generated Names	None
(6) Address Clause Expression Interpretation	An expression that appears in an object address clause is interpreted as the address of the first storage unit of the object.
(7) Restrictions on Unchecked Conversions	Supported except for the case where the destination type is an unconstrained record or array type.
(8) Implementation-Dependent Characteristics of the I/O Packages.	1. In Text_IO, the type Count is defined as follows: type Count is range 0..(2 ** 31)-2 2. In Text_IO, the type Field is defined as follows subtype Field is integer range 0..1000 3. In Text_IO, the Form parameter of procedures Create and Open is not supported. (If you supply a Form parameter with either procedure, it is ignored.) 4. The standard library contains preinstantiated versions of Text_IO.Integer_IO for types Short_Integer and Integer, and of Text_IO.Float_IO for types Float and Long_Float. We suggest that you use the following to eliminate multiple instantiations of these packages: Integer_Text_IO Short_Integer_Text_IO Float_Text_IO Long_Float_Text_IO

1. Implementation-defined pragmas

There are nine implementation-defined pragmas in TeleGen2: pragmas Comment, Export, Images, Interface_Information, Interrupt, Linkname, No_Suppress, Preserve_Layout, and Suppress_All.

1.1. Pragma Comment

Pragma Comment is used for embedding a comment into the object code. Its syntax is

```
pragma Comment ( <string_literal> );
```

where "<string_literal>" represents the characters to be embedded in the object code. Pragma Comment is allowed only within a declarative part or immediately within a package specification. Any number of comments may be entered into the object code by use of pragma Comment.

1.2. Pragma Export

Pragma Export allows you to call an Ada subprogram or reference an Ada object within an Ada program from program that is written in non-Ada code. The syntax of this pragma is

```
pragma Export ( [ Name => ] <subprogram> | <object_simple_name>
                [, [ Link_Name => ] <string_literal> ]
                [, [ Language => ] <identifier> ]
                [, [ Form => ] <string_literal> ] );
```

where:

Name	The simple name of an Ada subprogram or object, within the restrictions of this pragma.
Link_Name	An optional string literal that defines the link name that external languages will use to access the named subprogram or object.
Language	An optional identifier that defines the name of the foreign language that will access the subprogram or object. Supported languages include Assembly, C and FORTRAN.
Form	An optional string literal used to encode information about how to pass parameters or about the kind of storage to use for objects. This target dependent argument is ignored in the MC680X0 implementation of TeleGen2.

The pragma Export can be given for a library subprogram or a nonderived subprogram. It is also allowed to be given for an object that is declared immediately within a package specification or body, and is declared within another subprogram, task, or generic unit

The pragma Export may be used only once for a given object or subprogram name. It must be given immediately within the same specification or generic part that contains the declaration of the object or subprogram. In the case of a library subprogram, the pragma must immediately following the subprogram declaration. If the name corresponds to more than one subprogram declared earlier within the same package specification or declarative part, the compiler issues a warning and the pragma is ignored.

The arguments for pragma Export are defined here.

Name — The name argument should be the simple name of a subprogram or object, and must not be a name declared by an object renaming declaration. The name also must not refer to an object that is not statically sized. If your object requires dynamic storage allocation, you can declare another object with the type System.Address that could initialized to the address of the dynamic object. You may then export the address object.

The name is allowed to be a name given by a subprogram renaming only if the renaming declaration occurs immediately within the same package specification or declarative part as the subprogram that is renamed, and an Export pragma does not otherwise apply to the subprogram. By applying pragma Export to a renamed subprogram, it is possible to export one or more subprograms from among a set of overloaded declarations.

The name must not denote a subprogram for which you have specified pragma Interface. Similarly, pragma Interface must not be used with a subprogram for which you have specified pragma Export.

Link_Name — The link name is used by external languages to access the named subprogram or object. If you do not explicitly state a link name, the simple name of the subprogram or object is used by default.

Language — This optional argument defines the name of the foreign language that will access the subprogram or object. If you do not specify a value for this argument for a subprogram, the TeleGen2 implementation uses the default calling convention for external calls to the subprogram.

This argument is not normally specified when using pragma Export for objects. This useage depends on the special allocations and access needs specific to a particular language

Form This argument is currently ignored in the MC680X0 implementation of TeleGen2.

1.3. Pragma Images

Pragma Images controls the creation and allocation of the image and index tables for a specified enumeration type. The image table is a literal string consisting of enumeration literals catenated together. The index table is an array of integers specifying the location of each literal within the image table. The length of the index table is therefore the sum of the lengths of the literals of the enumeration type; the length of the index table is one greater than the number of literals.

The syntax of this pragma is

```
pragma Images(<enumeration_type>, Deferred);
        -- or --
pragma Images(<enumeration_type>, Immediate);
```

The "deferred" option saves space in the literal pool by not creating image and index tables for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If one of these attributes is used, the tables are generated in the literal pool of the compilation unit in which the attribute appears. If the attributes are used in more than one compilation unit, more than one set of tables is generated, eliminating the benefits of deferring the table. In this case, the "immediate" option saves space by causing a single image table to be generated in the literal pool of the unit declaring the enumeration type. For the MC680X0, "immediate" is the default option.

For a very large enumeration type, the length of the image table will exceed Integer'Last (the maximum length of a string). In this case, using either

```
pragma Images(<enumeration_type>, Immediate);
```

or the 'Image, 'Value, or 'Width attribute for the type will result in an error message from the compiler. Therefore, use the "deferred" option, and avoid using 'Image, 'Value, or 'Width in this case.

1.4. Pragma Interface_Information

The existing Ada interface pragma only allows specification of a language name. In some cases, the optimizing code generator will need more information than can be derived from the language name. Therefore there is a need for an implementation-specific pragma, Interface_Information.

There is an extended usage of this pragma for machine code insertion (MCI) procedures which does not use a preceding pragma Interface. Other than that case, a pragma Interface_Information is always associated with a Pragma

Interface. The syntax is

```
pragma Interface_Information (Name,
                              Link_Name,
                              Mechanism,
                              Parameters,
                              Clobbered_Regs);
```

where

```
name                :== ada_subprogram_identifier, required
link_name           :== string, default = ""
mechanism           :== string, default = ""
parameters          :== string, default = ""
clobbered_regs      :== string, default = ""
```

Scope

Pragma Interface_Information is allowed wherever the standard pragma Interface is allowed, and must be immediately preceded by a pragma Interface referring to the same Ada subprogram, in the same declarative part or package specification; no intervening declaration is allowed between the Interface and Interface_Information pragmas. Unlike pragma Interface, this pragma is not allowed for overloaded subprograms (it specifies information that pertain to one specific body of non-Ada code). If the user wishes to use overloaded Ada names, the Interface_Information pragma may be applied to unique renaming declarations.

The pragma is also allowed for a library unit; in that case, the pragma must occur immediately after the corresponding Interface pragma, and before any subsequent compilation unit.

This pragma may be applied to any interfaced subprogram, regardless of the language or system named in the interface pragma. The code generator is responsible for rejecting or ignoring illegal or redundant interface information. The optimizing code generator will process and check the legality of such interfaced subprograms at the time of the spec compilation, instead of waiting for an actual use of the interfaced subprogram. This will save the user from extensive recompilation of the offensive specification and all its dependents should an illegal pragma have been used.

This pragma is also used for MCI procedures. In that case, the "mechanism" should be set to "mci." This allows the user to specify detailed parameter characteristics for the call and inlined call to the MCI procedure. When used in conjunction with pragma Inline, this allows the user to directly insert a minimal set of instructions into the call location.

Parameters

Name: Ada subprogram identifier. The rule detailed in LRM 13.9 for a subprogram named in a pragma Interface apply here as well. As explained above, the subprogram must have been named in an immediately preceding Interface pragma.

This is the only required parameter. Since the other parameters are optional, positional association may only be used if all parameters are specified, or only the rightmost ones are defaulted.

Link Name: string literal. When specified, this parameter indicates the name the code generator must use to reference the named subprogram. This string name may contain any characters allowed in an Ada string and must be passed unchanged (in particular, not case-mapped) to the code generator. The code generator will reject names that are illegal in the particular language or system being targeted.

If this parameter is not specified, it defaults to a null string. The code generator will interpret a default link_name differently, depending on the target language/system (the default is generally the Ada name, or is derived from it, for example, "_Ada_name" for 'C' calls).

Mechanism: string literal. The only mechanism currently implemented is the "mci" mechanism used strictly in conjunction with MCI procedures.

Parameters: string literal. This string, when present, tells the code generator where to pass each parameter. This string is interpreted as a positional aggregate where each position refers to a parameter of the interfaced subprogram. Each position may be one of the following: null, the name of a register, or the word "stack." Null arguments imply standard conventions. Thus the string "r3, stack, r5" specifies that the first parameter is to be passed in register r3, the second parameter is to be put on the stack in the parameter block (in the proper position of the second parameter), and the third parameter is to be passed in register r5.

Clobbered Regs. These are the registers (comma-separated list) that are destroyed by this operation. The code generator will save anything valuable in these registers at the point of the call.

A simple example of the use of pragma Interface_Information is

```
procedure Do_Something (Addr: System.Address; Len: Integer);
pragma Interface (Assembly, Do_Something);
pragma Interface_Information ( Name       => Do_Something,
                               Link_Name  => "DOIT",
                               Parameters => "R3,R5");
```

1.5. Pragma Interrupt

The Ada LRM provides for interrupt handlers written in Ada. The approach is to associate a task entry with an interrupt source by means of an address clause. Such an entry is referred to as an "interrupt entry" (LRM 13.5.1). A task containing an interrupt entry is referred to in this section as an "interrupt task." When an interrupt occurs, it is handled as if an entry call had been made by the hardware to the entry associated with that interrupt. For example (according to the LRM)

```
task Interrupt_Handler is
  entry Done;
  for Done use at 16#40#; -- Assume that System.Address is
                          -- an integer type
end Interrupt_Handler;
```

In this example, the interrupt entry Done is associated with the interrupt vector at hexadecimal address 40. When a physical device causes an interrupt through that vector, an entry call is made to Done, which can *handle* the interrupt in an accept statement.

The AEE provides the facilities required by the LRM and goes substantially beyond those requirements to meet the needs of realistic systems. This section describes the interrupt-related facilities of the AEE and contrasts them with the minimal mechanism defined by the LRM.

In the TeleGen2 approach, the address clause designating an interrupt entry refers to the address of an interrupt descriptor, rather than to the address of the physical interrupt source. The Interrupt package provides a private descriptor type for this purpose.

```
type Descriptor is private;
```

The descriptor type Descriptor is used to associate an Ada task entry with an interrupt source.

If a suitable descriptor object of type Descriptor is declared, the LRM example then appears as follows:

```
Device : Interrupt.Descriptor,
```

```
task Interrupt_Handler is
   entry Done;
   for Done use at Device'Address;
end Interrupt_Handler;
```

Optimized interrupt entries

The facilities described so far are sufficient to implement interrupt handlers in Ada. However, the process of handling an interrupt in this fashion is potentially complicated and time-consuming. Ada does not restrict the language features that can be used inside the body of an accept statement. Therefore, an interrupt handler could contain entry calls to other tasks or even delay statements. Furthermore, in the general case, a full Ada context switch must be made to the interrupt handler task and then a full context switch back to the interrupted task (or potentially some other ready task) when the rendezvous is completed.

In some cases, the properties of fully general Ada interrupt handlers may suit the intended application. In other cases, however, it may be necessary to trade a reduction in generality for an increase in performance in order to meet application requirements. The AEE addresses these needs by allowing programmers to select one of two optimized constructs by which task entries can handle interrupts:

- **Synchronization Optimizations**
  The interrupt serves only to cause the handler task to become *ready to execute* without requiring an actual context switch as part of servicing the interrupt.

- **Function-Mapped Optimizations**
  All processing associated with handling the interrupt occurs during the rendezvous (in the body of the accept statement) and no interactions with other tasks occur during the rendezvous.

Synchronization optimizations

A synchronization optimization corresponds to having an empty accept body that simply puts the interrupt handler on a ready queue. This optimization is always applied when appropriate, without explicit programmer request. For example

```
task body Actuator_Driver is
begin
  accept Device_Ready;
  -- Actions responding to the device-ready signal
end Actuator_Driver;
```

Occurrence of the sighup signal causes Actuator_Driver to be placed in the ready queue. It is activated subsequently when its priority relative to other competing tasks so indicates.

Function-mapped optimizations

In a function-mapped optimization, all the interrupt handling work is done inside the accept body during the rendezvous. When this optimization is invoked, the compiler maps the accept body into a function that can be directly called from the signal handler. This kind of optimization is restricted to accept statements that do not interact with other tasks during the rendezvous. Consider, for example, the following fragment

```
task body Actuator_Driver is
begin
  pragma Interrupt (Function_Mapping);
  accept Device_Ready do
    -- Actions responding to the Device_Ready interrupt.
  end Device_Ready;
end Actuator_Driver;
```

The pragma Interrupt applies to the statement immediately following it, which must be one of the following three constructs:

1. A simple accept statement, as described in the preceding.

2. A while loop directly enclosing only a single accept statement, discussed in the following.

3. A select statement that includes an interrupt accept alternative.

For reasons related to the loop optimization discussed in the following, the server task with a function-mapped accept cannot have a user-specified priority.

The body of the accept statement handling the interrupt is executed in the environment of the interrupted current task. Note that the function-mapped body acts much like a classic interrupt procedure and requires no context switch even though it acts in the proper lexical environment.

The interrupt server often executes a small or null amount of non-handler code between accepting interrupt entry calls. The interrupt support is designed to take advantage of this occurrence to minimize latency in the driver and execute another handler with the minimum number of task switches. The best special case for this is an accept statement directly embedded inside a loop. For instance, the actuator driver is presented with a buffer of actuator commands. The driver contains a loop that waits on successive occurrences of the Device_Ready interrupt and issues commands out of the buffer. The function-mapping optimization caters to

this possibility as well. Consider the following fragment

```
task body Actuator_Driver is
begin
  -- ...
  pragma Interrupt (Function_mapping);
  while More_Commands loop
    accept Device_Ready do
      -- Issue the next command
    end Device_Ready;
  end loop;
  -- ...
end Actuator_Driver;
```

This example shows the second class of constructs to which the function-mapping optimization can be applied—a while loop that immediately contains (and *only* contains) an accept statement for an interrupt entry. The accept statement must meet the constraint described earlier (i.e., contain no interactions with other tasks).

1.6. Pragma Linkname

Pragma Linkname is used to provide interface to any routine whose name cannot be specified by an Ada string literal. This allows access to routines whose identifiers do not conform to Ada identifier rules.

Pragma Linkname takes two arguments. The first is a subprogram name that has been previously specified in a pragma Interface statement. The second is a string literal specifying the exact link name to be employed by the code generator in emitting calls to the associated subprogram. The syntax is

```
pragma Interface ( assembly, <subprogram_name> );
pragma Linkname ( <subprogram_name>, <string_literal> );
```

If pragma Linkname does not immediately follow the pragma Interface for the associated program, a warning will be issued saying that the pragma has no effect

A simple example of the use of pragma Linkname is

```
procedure Dummy_Access( Dummy_Arg : System.Address );
pragma Interface (assembly, Dummy_Access );
pragma Linkname (Dummy_Access, "_access");
```

Note It is preferable that the user use pragma Interface_Information for this functionality

1.7. Pragma No_Suppress

No_Suppress is a TeleGen2-defined pragma that prevents the suppression of checks within a particular scope. It can be used to override pragma Suppress in an enclosing scope. No_Suppress is particularly useful when you have a section of code that relies upon predefined checks to execute correctly, but you need to suppress checks in the rest of the compilation unit for performance reasons.

Pragma No_Suppress has the same syntax as pragma Suppress and may occur in the same places in the source. The syntax is

```
pragma No_Suppress (<identifier> [, [ON =>] <name>]);
```

where:

<identifier> The type of check you want to suppress. Checks that may be suppressed are Access_Check, Discriminant_Check, Index_Check, Length_Check, Range_Check, Division_Check, Overflow_Check, Elaboration_Check, and Storage_Check (refer to LRM 11.7).

<name> The name of the object, type/subtype, task unit, generic unit, or subprogram within which the check is to be suppressed; <name> is optional.

If neither Suppress nor No_Suppress is present in a program, checks will not be suppressed. You may override this default at the command level, by compiling the file with the -i(nhibit option and specifying with that option the type of checks you want to suppress. For more information on -i(nhibit, refer to other TeleGen2 documentation.

If either Suppress or No_Suppress are present, the compiler uses the pragma that applies to the specific check in order to determine whether that check is to be made. If both Suppress and No_Suppress are present in the same scope, the pragma declared last takes precedence. The presence of pragma Suppress or No_Suppress in the source takes precedence over an -i(nhibit option provided during compilation.

1.8. Pragma Preserve_Layout

The TeleGen2 compiler reorders record components to minimize gaps within records. Pragma Preserve_Layout forces the compiler to maintain the Ada source order of components of a given record type, thereby preventing the compiler from performing this record layout optimization.

The syntax of this pragma is

```
Pragma Preserve_Layout ( ON => Record_Type_Name )
```

Preserve_Layout must appear before any forcing occurrences of the record type and must be in the same declarative part, package specification, or task specification. This pragma can be applied to a record type that has been packed. If Preserve_Layout is applied to a record type that has a record representation clause, the pragma only applies to the components that do not have component clauses. These components will appear in Ada source order after the components with component clauses.

1.9. Pragma Suppress_All

Suppress_All is a TeleGen2-defined pragma that will suppress all checks in a given scope. Pragma Suppress_All contains no arguments and can be placed in the same scopes as pragma Suppress.

In the absence of pragma Suppress_All or any other suppress pragma, the scope which contains the pragma will have checking turned off. This pragma should be used in a safe piece of time critical code to allow for better performance.

2. Implementation-dependent attributes

2.1. 'Address and 'Offset

For MCI users who need to access Ada objects other than register parameters, two attributes are utilized, 'Address and 'Offset. These attributes allow you to access compiler information on the location of variables. 'Address is a *language-defined* attribute that has implementation-specific characteristics; 'Offset is an *implementation-defined* attribute.

'Address

This attribute is normally used to access some global control variable or composite structure. 'Address is also used in conjunction with local labels. See details in the following sections on usage of 'Address in the actual code statements. Note that no special code is generated automatically; this attribute simply provides the appropriate value for the absolute address.

'Offset

This attribute yields the offset of an Ada object from its parent frame. For a global object, this is the offset from the base of the compilation unit data section (although 'Address is the preferred way to access globals). For objects inside subprograms, 'Offset yields the offset in the local stack frame. This is primarily for usage with parameters that are not passed in registers. A secondary usage is to code an MCI "function" where an Ada function is wrapped around an MCI procedure declaration and then calls the MCI procedure with inlining. provides an efficient way to overcome the language limitation that MCI subprograms can only be procedures.

2.2. Extended attributes for scalar types

The extended attributes extend the concept behind the Text_IO attributes 'Image, 'Value, and 'Width to give the user more power and flexibility when displaying values of scalars. Extended attributes differ in two respects from their predefined counterparts:

1. Extended attributes take more parameters and allow control of the format of the output string.

2. Extended attributes are defined for all scalar types, including fixed and floating point types.

Extended versions of predefined attributes are provided for integer, enumeration, floating point, and fixed point types:

Integer:	'Extended_Image,	'Extended_Value,	'Extended_Width
Enumeration:	'Extended_Image,	'Extended_Value,	'Extended_Width
Floating Point:	'Extended_Image,	'Extended_Value,	'Extended_Digits
Fixed Point:	'Extended_Image, 'Extended_Aft	'Extended_Value,	'Extended_Fore,

The extended attributes can be used without the overhead of including Text_IO in the linked program. The following are examples that illustrates the difference between instantiating Text_IO.Float_IO to convert a float value to a string and using Float'Extended_Image:

```
with Text_IO;
function Convert_To_String ( Fl : Float ) return String is
  Temp_Str : String ( 1 .. 6 + Float'Digits );
package Flt_IO is new Text_IO.Float_IO (Float);
begin
  Flt_IO.Put ( Temp_Str, Fl );
  return Temp_Str;
end Convert_To_String;


function Convert_To_String_No_Text_IO( Fl : Float ) return String is
begin
  return Float'Extended_Image ( Fl );
end Convert_To_String_No_Text_IO;


with Text_IO, Convert_To_String, Convert_To_String_No_Text_IO;
procedure Show_Different_Conversions is
  Value : Float := 10.03376;
begin
  Text_IO.Put_Line ( "Using the Convert_To_String, the value of the variable
  is : " & Convert_To_String ( Value ) );
  Text_IO.Put_Line ( "Using the Convert_To_String_No_Text_IO, the value
  is : " & Convert_To_String_No_Text_IO ( Value ) );
end Show_Different_Conversions;
```

2.2.1. Integer attributes

'Extended_Image

Usage:

```
X'Extended_Image(Item,Width,Base,Based,Space_If_Positive)
```

Returns the image associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that the value of Item is an integer literal with no underlines, no exponent, no leading zeros (but a single zero for the zero value), and a minus sign if negative. If the resulting sequence of characters to be output has fewer than Width characters, leading spaces are first output to make up the difference. (LRM 14.3.7:10,14.3.7:11)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more than one parameter. The parameter Item must be an integer value. The resulting string is without underlines, leading zeros, or trailing spaces.

Parameter descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Width	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. *Optional*
Base	The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in base notation or not. If no preference is specified, the default (false) is assumed. *Optional*
Space_If_Positive	An indication of whether or not a positive integer should be prefixed with a space in the string returned. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following subtype were declared

```
subtype X is Integer Range -10..16;
```

Then the following would be true

```
X'Extended_Image(5)                    = "5"
X'Extended_Image(5,0)                  = "5"
X'Extended_Image(5,2)                  = " 5"
X'Extended_Image(5,0,2)                = "101"
X'Extended_Image(5,4,2)                = " 101"
X'Extended_Image(5,0,2,True)           = "2#101#"
X'Extended_Image(5,0,10,False)         = "5"
X'Extended_Image(5,0,10,False,True)    = " 5"
X'Extended_Image(-1,0,10,False,False)  = "-1"
X'Extended_Image(-1,0,10,False,True)   = "-1"
X'Extended_Image(-1,1,10,False,True)   = "-1"
X'Extended_Image(-1,0,2,True,True)     = "-2#1#"
X'Extended_Image(-1,10,2,True,True)    = "      -2#1#"
```

'Extended_Value

Usage:

```
X'Extended_Value(Item)
```

Returns the value associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that given a string, it reads an integer value from the beginning of the string. The value returned corresponds to the sequence input. (LRM 14.3.7:14)

For a prefix X that is a discrete type or subtype, this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter description:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type X. *Required*

Examples:

Suppose the following subtype were declared

```
Subtype X is Integer Range -10..16;
```

Then the following would be true

```
X'Extended_Value("5")          = 5
X'Extended_Value(" 5")         = 5
X'Extended_Value("2#101#")     = 5
X'Extended_Value("-1")         = -1
X'Extended_Value(" -1")        = -1
```

'Extended_Width

Usage:

```
X'Extended_Width(Base,Based,Space_If_Positive)
```

Returns the width for subtype of X.

For a prefix X that is a discrete subtype, this attribute is a function that may have multiple parameters. This attribute yields the maximum image length over all values of the type or subtype X.

Parameter descriptions:

Base	The base for which the width will be calculated. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether the subtype is stated in based notation. If no value for based is specified, the default (false) is assumed. *Optional*
Space_If_Positive	An indication of whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following subtype were declared

```
Subtype X is Integer Range -10..16;
```

Then the following would be true

```
X'Extended_Width                    = 3  -- "-10"
X'Extended_Width(10)                = 3  -- "-10"
X'Extended_Width(2)                 = 5  -- "10000"
X'Extended_Width(10,True)           = 7  -- "-10#10#"
X'Extended_Width(2,True)            = 8  -- "2#10000#"
X'Extended_Width(10,False,True)     = 3  -- " 16"
X'Extended_Width(10,True,False)     = 7  -- "-10#10#"
X'Extended_Width(10,True,True)      = 7  -- " 10#16#"
X'Extended_Width(2,True,True)       = 9  -- " 2#10000#"
X'Extended_Width(2,False,True)      = 6  -- " 10000"
```

2.2.2. Enumeration type attributes

'Extended_Image

Usage:

```
X'Extended_Image(Item,Width,Uppercase)
```

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that given an enumeration literal, it will output the value of the enumeration literal (either an identifier or a character literal). The character case parameter is ignored for character literals. (LRM 14.3.9:9)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more that one parameter. The parameter Item must be an enumeration value. The image of an enumeration value is the corresponding identifier, which may have character case and return string width specified.

Parameter descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Width	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. If the Width specified is larger than the image of Item, the return string is padded with trailing spaces. If the Width specified is smaller than the image of Item, the default is assumed and the image of the enumeration value is output completely. *Optional*
Uppercase	An indication of whether the returned string is in uppercase characters. In the case of an enumeration type where the enumeration literals are character literals, Uppercase is ignored and the case specified by the type definition is taken. If no preference is specified, the default (true) is assumed. *Optional*

Examples:

Suppose the following types were declared

```
type X is (red, green, blue, purple);
type Y is ('a', 'B', 'c', 'D');
```

Then the following would be true

```
X'Extended_Image(red)              = "RED"
X'Extended_Image(red, 4)           = "RED "
X'Extended_Image(red,2)            = "RED"
X'Extended_Image(red,0,false)      = "red"
X'Extended_Image(red,10,false)     = "red       "
Y'Extended_Image('a')              = "'a'"
Y'Extended_Image('B')              = "'B'"
Y'Extended_Image('a',6)            = "'a'  "
Y'Extended_Image('a',0,true)       = "'a'"
```

'Extended_Value

Usage:

```
X'Extended_Value(Item)
```

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that it reads an enumeration value from the beginning of the given string and returns the value of the enumeration literal that corresponds to the sequence input. (LRM 14.3.9:11)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter descriptions:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of X. *Required*

Examples:

Suppose the following type were declared

```
type X is (red, green, blue, purple);
```

Then the following would be true

```
X'Extended_Value("red")          = red
X'Extended_Value(" green")       = green
```

```
X'Extended_Value("    Purple")   = purple
X'Extended_Value(" GreEn  ")     = green
```

'Extended_Width

Usage:

```
X'Extended_Width
```

Returns the width for subtype of X.

For a prefix X that is a discrete type or subtype; this attribute is a function. This attribute yields the maximum image length over all values of the enumeration type or subtype X.

Parameter descriptions:

There are no parameters to this function. This function returns the width of the largest (width) enumeration literal in the enumeration type specified by X.

Examples:

Suppose the following types were declared

```
type X is (red, green, blue, purple);
type Z is (X1, X12, X123, X1234);
```

Then the following would be true

```
X'Extended_Width      = 6  -- "purple"
Z'Extended_Width      = 5  -- "X1234"
```

2.2.3. Floating point attributes

'Extended_Image

Usage:

```
X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)
```

Returns the image associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameter descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Fore	The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. *Optional*
Aft	The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. *Optional*
Exp	The minimum number of digits in the exponent. The exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. *Optional*
Base	The base that the image is to be displayed in. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true

```
X'Extended_Image(5.0)                    = " 5.0000E+00"
X'Extended_Image(5.0,1)                  = "5.0000E+00"
X'Extended_Image(-5.0,1)                 = "-5.0000E+00"
X'Extended_Image(5.0,2,0)                = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)              = " 5.0"
X'Extended_Image(5.0,2,0,0,2)            = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)       = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)       = "2#1.1#E+02"
```

'Extended_Value

Usage:

```
X'Extended_Value(Item)
```

Returns the value associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it skips any leading zeros, then reads a plus or minus sign if present then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter descriptions:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of the input string. *Required*

Examples:

Suppose the following type were declared

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true

```
X'Extended_Value("5.0")          = 5.0
```

```
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Digits

Usage:

```
X'Extended_Digits(Base)
```

Returns the number of digits using base in the mantissa of model numbers of the subtype X.

Parameter descriptions:

Base	The base that the subtype is defined in. If no base is specified, the default (10) is assumed. *Optional*

Examples:

Suppose the following type were declared

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true

```
X'Extended_Digits    = 5
```

2.2.4. Fixed-point attributes

'Extended_Image

Usage:

```
X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)
```

Returns the image associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

For a prefix X that is a discrete type or subtype, this attribute is a function that may have more than one parameter. The parameter Item must be a Real value. The resulting string is without underlines or trailing spaces

Parameter descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Fore	The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. *Optional*
Aft	The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. *Optional*
Exp	The minimum number of digits in the exponent; the exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. *Optional*
Base	The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true

```
X'Extended_Image(5.0)                    = " 5.00E+00"
X'Extended_Image(5.0,1)                  = "5.00E+00"
X'Extended_Image(-5.0,1)                 = "-5.00E+00"
X'Extended_Image(5.0,2,0)                = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)              = " 5.0"
X'Extended_Image(5.0,2,0,0,2)            = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)       = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)       = "2#1.1#E+02"
```

'Extended_Value

Usage:

```
X'Extended_Value(Image)
```

Returns the value associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it skips any leading zeros, reads a plus or minus sign if present, then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter descriptions:

Image	Parameter of the predefined type string. The type of the returned value is the base type of the input string. *Required*

Examples:

Suppose the following type were declared

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
```

```
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Fore

Usage:

```
X'Extended_Fore(Base,Based)
```

Returns the minimum number of characters required for the integer part of the based representation of X.

Parameter descriptions:

Base	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true

```
X'Extended_Fore                    = 3   -- "-10"
X'Extended_Fore(2)                 = 6   -- " 10001"
```

'Extended_Aft

Usage:

```
X'Extended_Aft(Base,Based)
```

Returns the minimum number of characters required for the fractional part of the based representation of X.

Parameter descriptions:

Base	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true

```
X'Extended_Aft      = 1  -- "1" from 0.1
X'Extended_Aft(2)   = 4  -- "0001" from 2#0.0001#
```

3. Package System

The current specification of package System is provided by the following.

```
with Unchecked_Conversion;

package System is

   --=====================
   -- CUSTOMIZABLE VALUES
   --=====================

   type Name     is (TeleGen2);

   System_Name   : constant name := TeleGen2;

   Memory_Size   : constant := (2 ** 31) -1; --Available memory,
                               --in storage units
   Tick          : constant := 1.0 / 100.0;  --Basic clock rate,
                               --in seconds


   --=======================================================
   -- NON-CUSTOMIZABLE, IMPLEMENTATION-DEPENDENT VALUES
   --=======================================================


      Storage_Unit : constant := 8;
      Min_Int      : constant := -(2 ** 31);
      Max_Int      : constant := (2 ** 31) - 1;
      Max_Digits   : constant := 15;
      Max_Mantissa : constant := 31;
      Fine_Delta   : constant := 1.0 / (2 ** Max_Mantissa);


      subtype Priority is Integer Range 0 .. 63;
```

```
--=======================
-- ADDRESS TYPE SUPPORT
--=======================

type Memory is private;
type Address is access Memory;
  --
  -- Ensures compatibility between addresses and access types.
  -- Also provides implicit NULL initial value.


Null_Address: constant Address := null;
  --
  -- Initial value for any Address object


type Address_Value is range -(2**31)..(2**31)-1;
  --
  -- A numeric representation of logical addresses for use in address clauses


function Location is new Unchecked_Conversion (Address_Value, Address);
  --
  -- May be used in address clauses:
  --
  --   Object: Some_Type;
  --   for Object use at Location (16#4000#);


function Label (Name: String) return Address;
pragma Interface (META, Label);
  --
  -- The LABEL meta-function allows a link name to be specified as address
  -- for an imported object in an address clause:
  --
  --  Object: Some_Type;
  --  for Object use at Label("OBJECT$$LINK_NAME");
  --
  -- System.Label returns Null_Address for non-literal parameters.


--
-- Unsigned address comparisons
--
function ">" (Left, Right: Address) return Boolean;
pragma Interface (META, ">");

function "<" (Left, Right: Address) return Boolean;
pragma Interface (META, "<");

function ">=" (Left, Right: Address) return Boolean;
pragma Interface (META, ">=");
```

```
function "<=" (Left, Right: Address) return Boolean;
pragma Interface (META, "<=");


--
-- Jnchecked relative address calculations
--
function "+" (Left: Address;       Right: Address_Value) return Address;
function "+" (Left: Address_Value; Right: Address)       return Address;
pragma Interface (META, "+");

function "-" (Left: Address;       Right: Address_Value) return Address;
function "-" (Left: Address;       Right: Address) return Address_Value;
pragma Interface (META, "-");



--=========================
-- ERROR REPORTING SUPPORT
--=========================

--
-- Report_Error can only be called in an exception handler and provides
--   an exception traceback like tracebacks provided for unhandled
--   exceptions
--
procedure Report_Error;
pragma Interface (Assembly, Report_Error);
pragma Interface_Information (Report_Error, "REPORT_ERROR");



--============
-- CALL SUPPORT
--============

 type Subprogram_Value IS
 record
    Proc_addr    : Address;
    Parent_frame : Address;
 end record;

  --
  -- Value returned by the implementation-defined 'Subprogram_Value
  -- attribute. The attribute is not defined for subprograms with
  -- parameters, or functions.


procedure Call (Subprogram: Subprogram_Value);
procedure Call (Subprogram: Address);

pragma Interface (META, Call);
  --
```

```
      -- The CALL meta-function allows indirect calls to subprograms
      -- given their subprogram value. The result of a call to a nested
      -- procedure whose parent frame does not exist (has been deallocated)
      -- at the time of the call, is undefined.
      --
      -- The second form allows calls to a subprogram given its address,
      -- as returned by the 'Address attribute. The call is undefined if
      -- the subprogram is not a parameterless non-nested procedure.

   Max_Object_Size    : CONSTANT := Max_Int;
   Max_Record_Count   : CONSTANT := Max_Int;
   Max_Text_Io_Count  : CONSTANT := Max_Int-1;
   Max_Text_Io_Field  : CONSTANT := 1000;


private
   type Memory is
   record
     null;
   end record;

end System;
```

3.1. System.Label

The System.Label meta-function is provided to allow users to address objects by a linker-recognized label name. This function takes a single string literal as a parameter and returns a value of System.Address. The function simply returns the run-time address of the appropriate resolved link name, the primary purpose being to address objects created and referenced from other languages.

- When used in an address clause, System.Label indicates that the Ada object or subprogram is to be referenced by a label name. The actual and this capability simply allows the user to import that object and reference it in Ada.

- When used in an expression, System.Label provides the link time address of any name: a name that might be for an object, a subprogram, etc.

3.2. System.Report_Error

Report_Error must only be called from within an exception handler, and must be the first thing done within it. This routine displays the normal exception traceback information to standard output. It is essentially the same traceback that could be obtained if the exception were unhandled and propagated out of the program, but the user may want to handle the exception and still display this information. The user may also want to use this capability in a user handler at the end of a task (since those exceptions will not be

propagated to the main program). Note that users can also get this capability for all tasks by using the -X(ception_show option.