AD-A240 760

ENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final:18 July 1991 to June 1993

4. TITLE AND SUBTITLE
TeleSoft, TeleGen2 Ada Host Development System, Version 4.1, for MacII Systems MacIIfx under A/UX 2.0 (Host & Target), 910721I1.11194

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 089

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

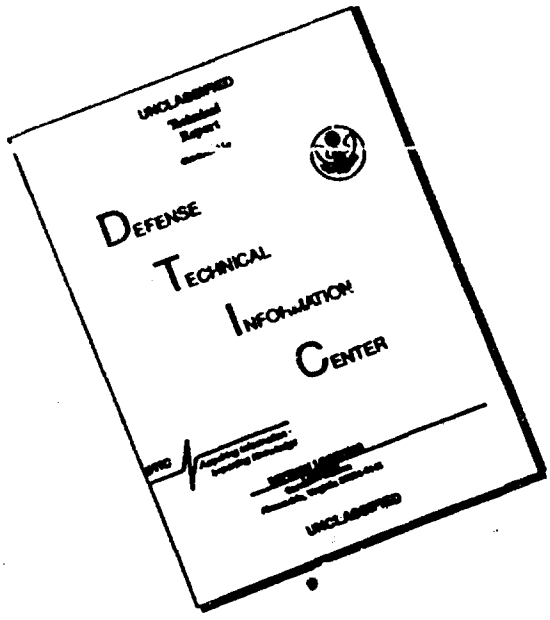
13. ABSTRACT *(Maximum 200 words)*
TeleSoft, TeleGen2 Ada Host Development System, Version 4.1, for MacII Systems, Ottonbrunn, Germany, MacIIfx under A/UX 2.0 (Host & Target), ACVC 1.11.

91-11061

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	

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

AVF Control Number: IABG-VSR 089
18 July, 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910721I1.11194
TeleSoft
TeleGen2™ Ada Host Development System
Version 4.1, for MacII Systems
MacIIfx under A/UX 2.0
Host and Target

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 91-07-21.

Compiler Name and Version: **TeleGen2™ Ada Host Development System, Version 4.1, for MacII Systems.**

Host Computer System: **Apple Macintosh IIfx under A/UX Version 2.0**

Target Computer System: **same as Host**

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

As a result of this validation effort, Validation Certificate #910721I1.11194 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
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 Host Development System, Version 4.1, for MacII Systems
Host Computer System:	Apple Macintosh IIfx under A/UX Version 2.0
Target Computer System:	Same as Host

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.

Beverly A. Knapp Date: 6 August 1991

TELESOFT

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

1.2 REFERENCES

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

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

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

1.3 ACVC TEST CLASSES

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

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

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

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

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

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly 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 91-05-03.

E28005C	B28006C	C34006D	C35508I	C35508J	C35508M
C35508N	C35702A	C35702B	B41308B	C43004A	C45114A
C45346A	C45612A	C45612B	C45612C	C45651A	C46022A
B49008A	B49008B	A74006A	C74308A	B83022B	B83022H
B83025B	B83025D	B83026B	C83026A	C83041A	B85001L
C86001F	C94021A	C97116A	C98003B	BA2011A	CB7001A
CB7001B	CB7004A	CC1223A	BC1226A	CC1226B	BC3009B
BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E	CD2A23E
CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C	BD3006A
BD3008A	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; for this implementation, there is no such type.

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

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

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)

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

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107B..E CE2107G..H CE2107L CD2110B CE2110D
CE2111D CE2111H CE3111B CE3111D..E CE3114B
CE3115A

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.

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 23 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B71001Q	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 technical and sales information about this Ada implementation, contact:

TeleSoft
5959 Cornerstone Court West
San Diego, CA 921219, 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.

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

a) Total Number of Applicable Tests	3802	
b) Total Number of Withdrawn Tests	94	
c) Processed Inapplicable Tests	73	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	274	(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

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

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

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

Compiler Option Information

B TESTS:

ada -O D -L <test_name>

option	description
ada	invoke Ada compiler
-O D	perform optimizations
-L	generate interspersed source-error listing
<test_name>	name of Ada source file to be compiled

Non-B Non-Family TESTS:

ada -m <main_unit> -O D <test_name>

option	description
ada	invoke Ada compiler
-m	produce executable code for <main_unit>
<main_unit>	name of main Ada compilation unit
-O D	perform optimizations
<test_name>	name of Ada source file to be compiled

Non-B Family TESTS:

ada -O D <test_name>
ald <main_unit>

option	description
ada	invoke Ada compiler
-O D	perform optimizations
<test_name>	name of Ada source file to be compiled
ald	invoke linker
<main_unit>	name of main Ada compilation unit

LINK:

ald <main_unit>

option	description
ald	invoke Linker
<main_unit>	name of main Ada 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	'"' & (1..V-2 => 'A') & '"'

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

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_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+37
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	0.0
$HIGH_PRICRITY	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         2147483648

$MIN_INT                -2147483648

$NAME                   NO_SUCH_TYPE_AVAILABLE

$NAME_LIST              TELEGEN2

$NAME_SPECIFICATION1    /tmp/X2120A

$NAME_SPECIFICATION2    /tmp/X2120B

$NAME_SPECIFICATION3    /tmp/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	0.02
$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.

2
Command Summary

This chapter presents the commands available with TeleGen2. They appear in alphabetical order.

Chapter 2 Contents

2.1. ada (Ada Compiler)

The *ada* command invokes the TeleGen2 Ada Compiler. 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 make sure you have access to TeleGen2 and have a working sublibrary and library file available. This was explained in the "Getting started" section of the Overview. We suggest you review that section, and then compile, link, and execute the sample program as indicated before you attempt to compile other programs.

The syntax of the *ada* command is shown below.

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

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

Library search	-l(ibfile, -t(emplib
Execution/output	Associate object: -A(ssociate Enable debugging: -d(ebug Abort after errors: -E(rror_abort Run front end only: -e(rrors_only Suppress checks: -i(nhibit Keep source: -K(eep_source Keep intermediates: -k(eep_intermediates Compile, then link: -m(ain Optimize code: -O(ptimize, -G(raph, -I(nline Update library for multiple files: -u(pdate_invoke Include execution profile: -x(ecution_profile
Listing	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> 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
```

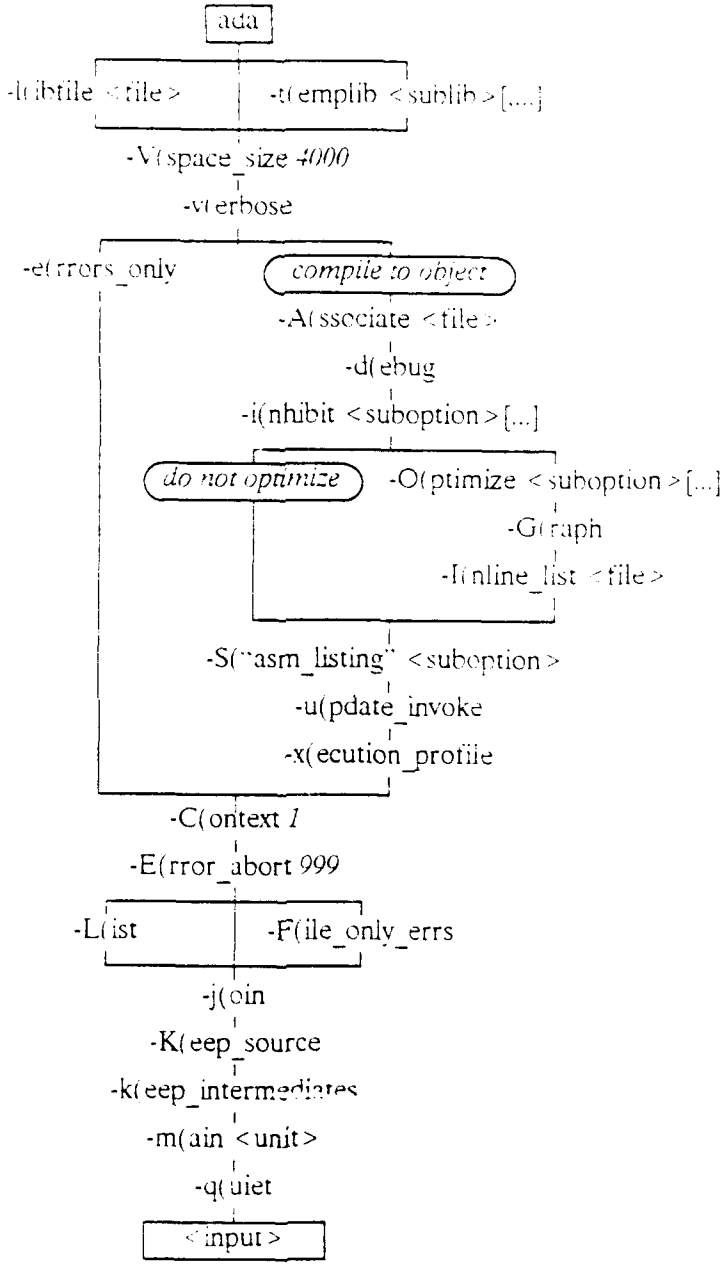
  If more than one file is specified, the names must be separated by a space.

- A file containing names of files to be compiled. Such a file must have the extension ".ilf"; each name in the file must be on a separate line. You can find details for using input-list files in the *User Guide* portion of your TeleGen2 documentation set.

- 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>" 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
-A(ssociate <file>
-d(ebug
-i(nhibit <suboption>[...]
do not optimize
-O(ptimize <suboption>[...]
-G(raph
-I(nline_list <file>
-S("asm_listing" <suboption>
-u(pdate_invoke
-x(ecution_profile
-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>

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* appear below in alphabetical order.

-A(ssociate

The -A(ssociate option is used to associate a foreign object with an Ada compilation unit. The format of the option is

```
-A <file>
```

where <file> is the name of the foreign object file. The object is assumed to be in the working sublibrary. Using the -A(ssociate option is meaningful only when the object is referenced by a pragma Interface within the file being compiled. For example, if you use

```
ada -A new.o calc.ada
```

the foreign object *new.o* is associated with the Ada unit in *calc.ada* (let's say it's unit Calc). Whenever Calc is bound, the foreign object *new.o* will also be bound.

The option is particularly useful for associating foreign objects with a main program. For example, instead of having to explicitly name the foreign object during linking, like this:

```
ald -v -p 'get_arg.o' show_argument
```

you can associate the object during compilation, like this:

```
ada -v -A get_arg.o show_arg.ada
```

If more than one object needs to be associated with a given file, put the objects in a UNIX archive (<file>.a) or do a partial link (ld -r) of the objects.

-C(ontext

When an error message is sent to *stderr*, it is helpful to see 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 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 <ctrl>-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).

-F(ile_only_errs

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. The structure and interpretation of call graphs is addressed in the Global Optimizer chapter of the TeleGen2 *User Guide*.

-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 file must contain subprogram names in a specific form as noted 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. Such optimizations are explained in detail in the Global Optimizer chapter of the TeleGen2 *User Guide*.

-i(nhibit

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

The format of the option is

```
-i <suboption>[...]
```

where <suboption> 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 writes errors, warning messages, and information messages that are generated during compilation back into the source file. Such errors and messages appear in the file as Ada comments. The comments thus generated 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.

Some of the information within the intermediate forms may be required later, which is the reason -k(eep_intermediates is available with *ada*. For example, High Form is required if the unit is to be referenced by the Ada cross-referencer (*axr*). In addition, both the debugger and optimizer require information that is saved within intermediate forms.

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. If the unit has been compiled with *-k*, the listing will show the attributes high_form and low_form for the unit.

-L(ist

The -L(ist 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 (rt.sub). Note that comments may be included in a library file and that each sublibrary listed must have the extension ".sub". For example, an alternate library file, *worklib.alb*, might contain the following lines.

```
Name:  mywork.sub
-- For the Remco Database project
Name:  calcproj/calclib.sub
Name:  $TELEGEN2/lib/rt.sub
```

Then to use *worklib.alb* instead of the default, *liblst.alb*, you would use:

```
-l worklib.alb
```

-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 prelinker (binder) and the native linker to bind and link the program with its extended family. An executable file named <unit> is left in the current directory. The linker may also be invoked directly by the user with the *ald* command.

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 binder/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*, *-X*, or any of the other *ald* options. For example, the command

```
ada -m welcome -o new sample.ada
```

instructs the compiler to compile the Ada source file *sample.ada*, which contains the main program unit Welcome. After compilation, the compiler calls the linker, passing to it the *-o* option with its arguments. The linker produces an executable version of the unit, placing it in file *new* as requested by the *-o* option.

-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 during compilation; this 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. Please refer to the TeleGen2 *User Guide* for information on optimizing strategies.

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

By default, information messages are output even if the -v(erbose option is not used. The -q(uiet option allows you to suppress such messages. Using -v(erbose alone gives error messages, banners, and information messages. Using -v(erbose with -q(uiet gives error messages and banners, but suppresses information messages. The option is particularly useful during optimization, when large numbers of information messages are likely to be output.

-S("asm_listing"

The *-S* option instructs the compiler to generate an assembly listing. The 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 <suboption>
```

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

- **e** [extended] Generate a paginated, extended assembly listing that includes code offsets and object code. The assembly file is named <unit>.e if it is a body or <unit>_.e if it is a specification.

- **a** [assembly] Generate a listing that can later be used as input to an assembler. The assembly file is named <unit>.s if it is a body or

<unit>_.s if it is a specification.

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>[,...]
```

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* (separated by commas) as the argument string.

-u(pdate_invoke

The -u(pdate_invoke (short for "-u(pdate_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 blocks; 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. Please read the section on adjusting the size of the virtual space in the Programming Guide chapter of the TeleGen2 *Programmer's Reference Manual* for more information.

-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, the banner and progress messages are not output. However, information messages such as those output by the optimizer may still be output whether -v(erbose is used or not.

-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.

Also make sure that the PROFILING environment variable is set before you attempt to execute a program. If the variable is not set in one of your log-in scripts, type

```
setenv PROFILING on
```

before you execute a profiled program. Refer to the Profiler chapter of the TeleGen2 *User Guide* for more information on profiling.

2.2. ald (Ada Linker)

The *ald* command invokes the TeleGen2 Ada Linker. The linker takes the object (of a main program) that is produced by the compiler and produces a UNIX executable module. To produce executable code, the linker (1) generates elaboration code and a link script (this is called "binding" or "prelinking"), then (2) calls the UNIX link editor (*ld*) to complete the linking process. ("Linker" refers to the TeleGen2 Ada Linker; "link editor" refers to the UNIX link editor.)

The linker is invoked by the *ald* command; it can also be invoked with the -main 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
```

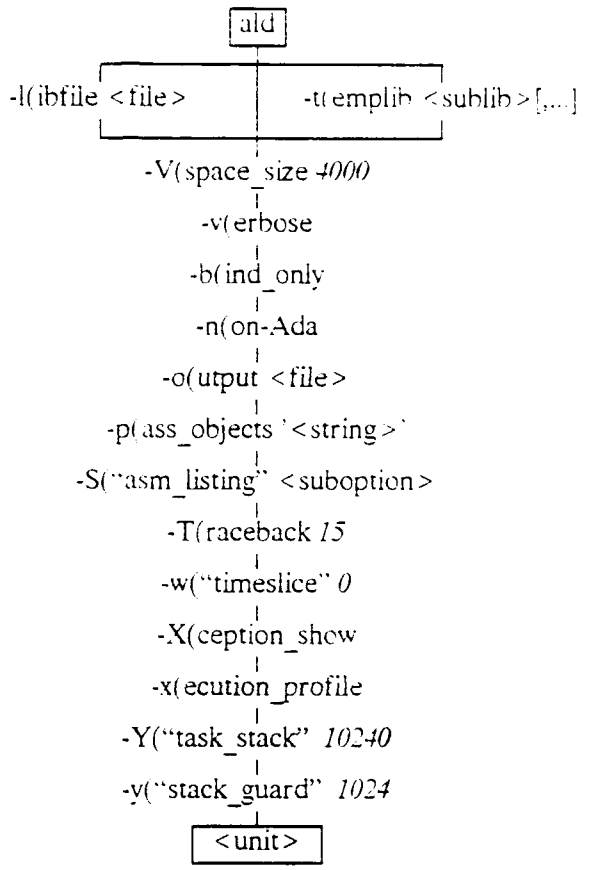
`<option>` One of the options available with the command.

`<unit>` The name of the main unit of the Ada program to be linked.

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, the *ald* command takes one argument — the name of the main unit of the Ada program structure that is to be linked — and produces one output file — the executable file produced by the linking process. The executable file is placed in the current working directory, under the name of the main unit used as the argument to *ald*. For System V versions of UNIX, if the name is longer than 14 characters, it is truncated.

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



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

1. (No options) The following command links the object modules of all the units in the extended family of the main unit Welcome, producing an executable file, *welcome*, in the working directory.

   ```
   ald welcome
   ```

2. The following command links the main unit Welcome, producing an executable file, *new*, in the working directory.

   ```
   ald -S a -v -o new welcome
   ```

   An assembly listing file, *new__M.s*, is produced as well. Progress messages are output as the command executes.

The options available with *ald* appear below in alphabetical order.

-b(ind_only

The -b(ind_only option causes the linker to quit after it has created the elaboration code and the linking order, but before it invokes the UNIX link editor. Using this option allows you to edit the linking order for special applications and then invoke the link editor directly.

The linking order is contained in a link script, which is an executable script that invokes the link editor with the appropriate options and arguments. The name of the script produced is <unit>.lnk, which is placed in the working directory. To complete the link process, enter "<unit>.lnk". The name of the file containing the elaboration code is <unit>__M.o, which is placed in the object directory of the working sublibrary.

For System V versions of UNIX, the file names generated as a result of linking are created by appending the 3-letter extension to the unit name and truncating the result to 14 characters.

-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 (rt.sub). Note that comments may be included in a library file and that each sublibrary listed must have the extension ".sub". For example, an alternate library file, *worklib.alb*, might contain the following lines.

```
Name:  mywork.sub
-- For the Remco Database project
Name:  calcproj/calclib.sub
Name:  $TELEGEN2/lib/rt.sub
```

Then to use *worklib.alb* instead of the default, *liblst.alb*, you would use:

```
-l worklib.alb
```

-n(on_Ada

The -n(on_Ada option tells the binder to make the elaboration procedure accessible from code written in another language. With *-n*, The linker generates elaboration code and produces a link script, <unit>.lnk, but does not call the link editor. The link script can be edited and submitted to the link editor.

The link script produced with the -n(on-Ada option differs from that produced by the -b(ind_only option in that the former includes the *.env_foreign* module instead of *.env*. Both modules are in $TELEGEN2/lib.

-o(utput

The -o(utput option allows you to specify the name of the output file produced by the linker. The format of the option is

```
-o <file>
```

where <file> is the name of the output file. For example, the command below causes the linker to put the executable module in the file "yorkshire" rather than "main".

```
ald -o yorkshire main
```

-p(ass_objects

The -p(ass_objects option allows you to pass a string of arguments directly to the UNIX link editor. The format of the option is

```
-p '<string>'
```

where <string> is a string of characters that the UNIX link editor, *ld*, will recognize. The string passed to *ld* may be either *objects* (e.g., 'cosine.o') or *options* (e.g., '-l<lib> -r'). The string must be enclosed in single quotes. For example, the command

```
ald -p 'cosine.o /usr/lib/libm.a' main
```

causes the link editor to link the object file *cosine.o* and to search the library */usr/lib/libm.a* for unresolved symbol references.

Remember that the UNIX link editor searches a library exactly once at the point it is encountered in the argument list, so references to routines in libraries must occur before the library is searched. That is, files that include references to library routines must appear before the corresponding libraries in the argument list. Objects and archives added with the *-p* option will appear in the linking order after Ada object modules and run-time support libraries, but before the standard C library (/lib/libc.a). This library is always the last element of the linking order.

You can also use -p(ass_objects to specify the link editor's *-l* option, which causes the link editor to search libraries whose names have the form "/lib/lib*name*.a" or "/usr/lib/lib*name*.a". For example, the command below causes the link editor to search the directories */lib* and */usr/lib* (in that order) for file *libxyz.a*.

```
ald -p '-lxyz'
```

("-l" is the *ld* option; "xyz" is the option's argument.)

If you use *-p* but do not invoke the link editor (by using -b(ind_only), the

binding information specified with *-p* is included in the link script.

-S("asm_listing"

The *-S* option is used to output an assembly listing from the elaboration process. The format of the option is

```
-S <suboption>
```

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

e [extended] Generate a paginated, extended assembly listing that includes code offsets and object code. The assembly file is named <unit>__M.e.

a [assembly] Generate a listing that can later be used as input to an assembler. The assembly file is named <unit>__M.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.

-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>[,...]
```

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* (separated by commas) as the argument string.

-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 the first ready task having 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. If the value specified is 15, for example, the run time will check each 15 milliseconds to see if any tasks with a priority equal to that of the executing task are available to execute. If there are, the run time effects a context switch to the first such task.

The UNIX virtual time alarm signal SIGVTALRM is used to implement the *-w* option. The value of *-w* is passed to the UNIX run time, which then sets the UNIX interval timer. Note that while *ald* will accept values beteen 0 and 2**31-1, not all of these values will be meaningful in the UNIX environment. For details on the UNIX timing mechanism, please refer to the appropriate UNIX documentation.

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 blocks; 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. Please read the section on adjusting the size of the virtual space in the Programming Guide chapter of the TeleGen2 *Programmer's Reference Manual* for more information.

-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, the banner and progress messages are not 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*, 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.

Also make sure that the PROFILING environment variable is set before you attempt to execute a program. If the variable is not set in one of your log-in scripts, type

```
setenv PROFILING on
```

before you execute a profiled program. Refer to the Profiler chapter of the TeleGen2 *User Guide* for more information on profiling.

-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 10240 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 10240. A representation specification for task storage size overrides a value supplied with this option.

-y("stack_guard"

The *-y* option is used to specify the size of the stack guard. The stack-guard space is the amount of space allocated per task, from the task stack, to accommodate interrupts and exception-handling operations. The format of

The option is

```
-y <value>
```

where <value> is the size of the stack-guard size in 8-bit bytes. The value given must be less than the task-stack size. The default is 1024 bytes; this is the amount allocated unless otherwise specified.

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.

ATTACHMENT F: 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 = -1.79769E−308
last = −1.79769E−308
machine_radix = 2
machine_mantissa = 53
machine_emin = -1021
machine_emax = −1024

DURATION:

size = 32
delta = 2#1.0#E-14
first = -86400
last = −86400

3.10. LRM Appendix F - Implementation-Dependent Characteristics

The Ada language definition allows for certain target dependencies. These dependencies must be described in the reference manual for each implementation. This section addresses each point listed in LRM Appendix F. Topics that require further clarification are addressed in the sections referenced in the summary.

3.10.1. (1) Implementation-dependent pragmas

TeleGen2 has the following implementation-dependent pragmas:

pragma Comment
pragma Export
pragma Images
pragma Interface_Information
pragma Interrupt
pragma Linkname
pragma No_Suppress
pragma Preserve_Layout
pragma Suppress_All

3.10.1.1. Pragma Comment

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

3.10.1.2. Pragma Export

Pragma Export enables you to export an Ada subprogram or object to either the C language or assembly. The pragma is not supported for Pascal or FORTRAN. The syntax is

```
pragma Export ( [ Name => ] <subprogram_or_object_name>
                [, [ Link_Name => ] <string_literal> ]
                [, [ Language  => ] <identifier> ] );
```

The syntax and use of the pragma is explained in detail in Section 2.8.3.

3.10.1.3. Pragma Images

Pragma Images controls the creation and allocation of the image and index tables for a specified enumeration type. The syntax is

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

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

The syntax and use of the pragma is described in detail in Section 2.7.3.

3.10.1.4. Pragma Interface_Information

Pragma Interface_Information provides information for the optimizing code generator when interfacing non-Ada languages or doing machine code insertions. Pragma Interface_Information is always associated with a pragma Interface except for machine code insertion procedures, which do not use a preceding pragma Interface. The syntax of the pragma is

```
pragma Interface_Information (Name,            -- Ada subprogram, required
                              Link_Name,       -- string, default = ""
                              Mechanism,       -- string, default = ""
                              Parameters,      -- string, default = ""
                              Clobbered_Regs); -- string, default = ""
```

Section 2.3.2.2 explains the syntax and usage of this pragma.

3.10.1.5. Pragma Interrupt

Pragma Interrupt is used for function-mapped optimizations of interrupts. The syntax is

```
pragma Interrupt (Function_Mapping);
```

The pragma has the effect that entry calls to the associated entry, on behalf of an interrupt, are made with a reduced call overhead. This pragma can only appear immediately before a simple accept statement, a while loop directly enclosing only a single accept statement, or a select statement that includes an interrupt accept alternative.

Pragma Interrupt is explained more fully in Sections 2.11, 2.11.1.5, and 2.11.1.7.

3.10.1.6. Pragma Linkname

Pragma Linkname was formerly used to provide interface to any routine whose name cannot be specified by an Ada string literal. Pragma Interface_Information should now be used for this functionality. Pragma Linkname is described here only in support of older code that may still use it.

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 ( <language>, <subprog> );
pragma Linkname ( <subprog>, <string_literal> );
```

If pragma Linkname does not immediately follow the pragma Interface for the associated subprogram, 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");
```

3.10.1.7. Pragma No_Suppress

Pragma 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. The pragma uses the same syntax and can occur in the same places in the source as pragma Suppress. The syntax is

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

<identifier> The type of check you do not want to suppress.

<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.

Section 2.3.2.2 explains the use of this pragma in more detail.

3.10.1.8. 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> )
```

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.

3.10.1.9. Pragma Suppress_All

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

In the presence of pragma Suppress_All or any other Suppress pragma, the scope that 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.

3.10.2. (2) Implementation-dependent attributes

TeleGen2 has the following implementation-dependent attributes:

'Offset (in MCI)
'Subprogram_Value
'Extended_Image
'Extended_Value
'Extended_Width
'Extended_Aft
'Extended_Digits
'Extended_Fore

3.10.2.1. 'Offset

'Offset yields the offset of an Ada object from its parent frame. This attribute supports machine code insertions as described in Section 2.12.2.2.

3.10.2.2. 'Subprogram_Value

This attribute is used by the TeleGen2 implementation to facilitate calls to interrupt support subprograms. The attribute returns the value of the record type Subprogram_Value defined in package System. Refer to Section 2.11.2.1 for more information.

3.10.2.3. Extended attributes for scalar types

The extended attributes extend the concept behind the text 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.

Named parameter associations are not currently supported for the extended attributes.

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

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

For integer and enumeration types, the 'Extended_Value attribute is identical to the 'Value attribute. For enumeration types, the 'Extended_Width attribute is identical to the 'Width attribute.

The extended attributes can be used without the overhead of including Text_IO in the linked program. The following examples illustrate 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;
```

3.10.2.3.1. Integer attributes

'Extended_Image

```
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.

Parameters

`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

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

Values yielded for selected parameters:

```
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

```
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

`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

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

Values yielded for selected parameters:

```
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

```
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.

Parameters

`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

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

Values yielded for selected parameters:

```
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"
```

3.10.2.3.2. Enumeration type attributes

'Extended_Image

```
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.

Parameters

`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 upper case 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

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

Values yielded for selected parameters:

```
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

```
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

`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

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

Values yielded for selected parameters:

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

'Extended_Width

```
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.

Parameters

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

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

Values yielded:

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

3.10.2.3.3. Floating point attributes

'Extended_Image

```
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.

Parameters

`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

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

Values yielded for selected parameters:

```
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

```
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

`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

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

Values yielded for selected parameters:

```
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

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

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

Parameter

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

Examples

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

Values yielded:

```
X'Extended_Digits     = 5
```

3.10.2.3.4. Fixed point attributes

'Extended_Image

```
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.

Parameters

`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

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

Values yielded for selected parameters:

```
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

```
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

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

Examples

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

Values yielded for selected parameters:

```
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

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

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

Parameters

`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

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

Values yielded for selected parameters:

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

'Extended_Aft

`X'Extended_Aft(Base,Based)`

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

Parameters

`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

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

Values yielded for selected parameters:

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

3.10.3. (3) Package System

```
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 := 2.0 / 100.0;   --Basic clock rate, in seconds


   type Task_Data is    --
     record             -- Adaptation-specific customization information
       null;            -- for task objects.
     end record;        --


   --=============================================================
   -- 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


Hex_80000000 : constant Address_Value := - 16#80000000#;
Hex_90000000 : constant Address_Value := - 16#70000000#;
Hex_A0000000 : constant Address_Value := - 16#60000000#;
Hex_B0000000 : constant Address_Value := - 16#50000000#;
Hex_C0000000 : constant Address_Value := - 16#40000000#;
Hex_D0000000 : constant Address_Value := - 16#30000000#;
Hex_E0000000 : constant Address_Value := - 16#20000000#;
Hex_F0000000 : constant Address_Value := - 16#10000000#;
  --
  -- Define numeric offsets to aid in Address calculations
  -- Example:
  --   for Hardware use at Location (Hex_F0000000 + 16#2345678#);


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.




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

procedure Report_Error;
pragma Interface (Assembly, Report_Error);
pragma Interface_Information (Report_Error, "REPORT_ERROR");
```

```
   --
   -- Report_Error can only be called in an exception handler and provides
   --   an exception traceback like tracebacks provided for unhandled
   --   exceptions
   --

   --------------
   -- 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.
     --


private

--

end System;
```

3.10.3.1. System.Label

The System.Label meta-function is provided to allow you 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 object must be created in some other unit (normally by another language), and this capability simply allows you to import that object and reference it in Ada. Any explicit or default initialization will be applied to the object. For example, if the object is declared to be of an access type, it will be initialized to NULL.

- When used in an expression, System.Label provides the link time address of any name, such as a name for an object or a subprogram.

3.10.3.2. System.Report_Error

Report_Error must be called from directly within an exception handler. 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. Report_Error simply allows you to handle the exception and still display this information. You may also want to use this capability in a user handler at the end of a task since exceptions in tasks will not be propagated to the main program. You can also get this capability for all tasks by using the -X binder switch.

For details on the output, refer to Section 2.9, "Exception handling."

3.10.4. (4) Restrictions on representation clauses

Representation clauses are fully supported with the following exceptions:

- Enumeration representation clauses are supported for all enumeration types except Boolean types.
- Record representation clauses are supported except for records with dynamically-sized components.
- Pragma Pack is supported except for dynamically-sized components.

3.10.5. (5) Implementation-generated names

TeleGen2 has no implementation-generated names.

3.10.6. (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.

3.10.7. (7) Restrictions on unchecked conversions

Unchecked programming is supported except for unchecked type conversions where the destination type is an unconstrained record or array type.

3.10.8. (8) Implementation-dependent characteristics of the I/O packages

Text_IO has the following implementation-dependent characteristics:

```
type Count is range 0..(2 ** 31)-2;

subtype Field is integer range 0..1000;
```

In procedures Create and Open, the Form parameter is supported as specified by the POSIX Draft 6, Chapter 8.

The standard run-time sublibrary contains preinstantiated versions of Text_IO.Integer_IO for types Short_Integer, Integer, and Long_Integer, and of Text_IO.Float_IO for types Float and Long_Float. Use the following packages to eliminate multiple instantiations of the Text_IO packages:

Short_Integer_Text_IO
Integer_Text_IO
Long_Integer_Text_IO
Float_Text_IO
Long_Float_Text_IO