REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	31 Jan 90 to 31 Jan 91	Final

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: TeleSoft and MODCOMP, MODCOMP TeleGen2 Ada System, MODCOMP 9730 (Host to Target), 900131I1.10269

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
AVF-IABG-068

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
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 and MODCOMP, MODCOMP TeleGen2 Ada System, Ottobrunn West Germany, MODCOMP Model 9730 under MODCOMP REAL/IX A.1 (Host & Target), ACVC 1.10.

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

15. NUMBER OF PAGES

16. PRICE CODE

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

AVF Control Number: AVF-IABG-063

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #900131I1.10269
Telesoft and MODCOMP
MODCOMP TeleGen2 Ada System
MODCOMP 9730 host and target

Completion of On-Site Testing:
31 January 1990

Prepared By:
IABG mbH, Abt. SZT
Einsteinstrasse 20
D-8012 Ottobrunn
West Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: MODCOMP TeleGen2 Ada Host Compilation System
Version 1.4A

Certificate Number: #900131I1.10269

Host: MODCOMP Model 9730
under MODCOMP REAL/IX A.1

Target: same as host

Testing Completed 31 January 1990 Using ACVC 1.10

This report has been reviewed and is approved.

IABG mbH, Abt. SZT
Dr. S. Heilbrunner
Einsteinstr. 20
D-8012 Ottobrunn
West 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

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO).

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO 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 compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

IABG mbH, Abt. SZT
Einsteinstr. 20
D-8012 Ottobrunn
West Germany

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

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

1.3 REFERENCES

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

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer which executes the code generated by the compiler.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada 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, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

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 every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It 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 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. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and tests. However, some tests contain values that require the test to be

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: MODCOMP TeleGen2 Ada Host Compilation System
Version 1.4A

ACVC Version: 1.10

Certificate Number: #900131I1.10269

Host Computer:

Machine:	MODCOMP Model 9730
Operating System:	MODCOMP REAL/IX A.1
Memory Size:	12 MegaBytes

Target Computer: same as host

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

1) This implementation supports the additional predefined types LONG_INTEGER and LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

1) Some of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

3) This implementation uses no extra bits for extra precision and uses no extra bits for extra range. (See test C35903A.)

4) NUMERIC_ERROR is raised for largest integer comparison and membership tests and no exception is raised for pre-defined

integer comparison and membership tests when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

1) The method used for rounding to integer is round to even. (See tests C46012A..Z (26 tests).)

2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z (26 tests).)

3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR for a two dimensional array subtype where the large dimension is the second one. (See test C36003A)

2) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

3) NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises no exception. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

6) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

8) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

f. Discriminated types.

1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

1) In the evaluation of a multi-dimensional aggregate, the test results indicate that index subtype checks are made as choices are evaluated. (See tests C43207A and C43207B.)

2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

1) The pragma INLINE is supported for procedures and for non-library functions. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

This implementation creates a dependence between a generic body and those units which instatiate it. As allowed by AI-408/11, if the body is compiled after a unit that instantiates it, then that unit becomes obsolete.

1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

1) The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

2) The package DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

9) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

10) Temporary sequential files are given names and not deleted when closed. (See test CE2108A.)

11) Temporary direct files are given names and not deleted when closed. (See test CE2108C.)

12) Temporary text files are given names and not deleted when closed. (See test CE3112A.)

13) More than one internal file can be associated with each external file for sequential files when reading only. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

14) More than one internal file can be associated with each external file for direct files when reading only. (See tests CE2107F..H (3 tests), CE2110D and CE2111H.)

15) More than one internal file can be associated with each external file for text files when reading only (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 314 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 15 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	127	1129	2018	17	23	45	3359
Inapplicable	2	9	297	0	5	1	314
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	TEST CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	573	544	245	172	99	160	332	132	36	250	340	278	3359
N/A	14	76	136	3	0	0	6	0	5	0	2	29	43	314
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84N	CD2A84M	CD50110	CD2B15C	CD7205C	CD2D11B
CD5007B	ED7004B	ED7005C	ED7005D	ED7006C	ED7006D
CD7105A	CD7203B	CD7204B	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 314 tests were inapplicable for the reason indicated.

a. The following 201 tests are not applicable because they 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)

b. C35508I, C35508J, C35508M, and C35508N are not applicable because they include enumeration representation clauses for BOOLEAN types in which the representation values are other than (FALSE => 0, TRUE => 1). Under the terms of AI-00325, this implementation is not required to support such representation clauses.

c. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

d. The following 16 tests are not applicable because this implementation does not support a predefined type SHORT_INTEGER:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	B86001V
CD7101E				

e. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because they acquire a value of SYSTEM.MAX_MANTISSA greater than 32.

f. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

g. B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

h. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

i. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

j. CA2009C, CA2009F, BC3204C and BC3205D are not applicable because this implementation creates a dependence between a generic body and those units which instantiate it (See Section 2.2.i and Appendix F of the Ada Standard).

k. LA3004B, EA3004D, and CA3004F are not applicable because this implementation does not support pragma INLINE for library functions.

l. CD1009C, CD2A41A..B (2 tests), CD2A41E and CD2A42A..J (10 tests) are not applicable because of restrictions on 'SIZE length clauses for floating point types.

m. CD2A61I..J (2 tests) are not applicable because of restrictions on 'SIZE length clauses for array types.

n. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because of restrictions on 'SIZE length clauses for access types.

o. CD4041A is not applicable because of restrictions on record representation clauses with 32 bit alignment.

p. AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

q. AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

r. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

s. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

t. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

u. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

v. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

w. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

x. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

y. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

z. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

aa. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

ab. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

ac. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

ad. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

ae. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

af. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

ag. CE2107B..E (4 tests), CE2107L, CE2110B, and CE2111D are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

ah. CE2107G..H (2 tests), CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

ai. CE3102E is inapplicable because text file CREATE with IN_FILE mode is supported by this implementation.

aj. CE3102F is inapplicable because text file RESET is supported by this implementation.

ak. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.

al. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.

am. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

an. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is supported by this implementation.

ao. CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access

is attempted.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 15 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B71001E	B71001K	B71001Q	B71001W	BA3006A	BA3006B
BA3007B	BA3008A	BA3008B	BA3013A (6 and 7M)		

Tests C34005G, C34005J and C34006D returned the result FAILED because of false assumptions that an element in an array or a record type may not be represented more compactly than a single object of that type. The AVO has ruled these tests PASSED if the only message of failure occurs from the requirements of T'SIZE due to the above assumptions (T is the array type).

In tests CD2C11A and CD2C11B the size specification in the representation clause for the task storage size for task type TTYPE was increased from 1024 to 2048 because 1024 bytes were insufficient for this compiler.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the TeleGen2 Ada Development System for a computing system based on the same instruction set architecture was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the TeleGen2 System successfully passed all applicable tests, and it exhibited the expected behavior on all inapplicable tests. The applicant certified that testing results for the computing system of this validation would be identical to the ones submitted for review prior to validation.

3.7.2 Test Method

Testing of the TeleGen2 Ada Development System using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host: MODCOMP Model 9730
under MODCOMP REAL/IX A.1

Target: same as host

A cartridge containing the customized test suite was loaded onto the host computer. Results were collected on the host computer and transferred via Ethernet to another computer for evaluation and archiving.

The compiler was tested using command scripts provided by TeleSoft and reviewed by the validation team. The tests were compiled using the command

```
ada -O D <filename>
```

and linked with the command

```
ald -v <main unit>
```

The -L qualifier was added to the compiler call for class B, expanded and modified tests. See Appendix E for explanation of compiler and linker switches. The <options file> contained a specification of memory addresses for the target computer.

Tests were compiled, linked, and executed (as appropriate) using one computer. Test output, compilation listings, and job logs were captured on cartridge and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at TeleSoft, San Diego, USA, and was completed on 31 January 1990.

APPENDIX A

DECLARATION OF CONFORMANCE

MODCOMP and TeleSoft submitted the following Declaration of Conformance concerning the TeleGen2 Ada System.

DECLARATION OF CONFORMANCE

Compiler Implementor: TELESOFT
Ada Validation Facility: IABG, Dept. SZT, D-8012 Ottobrunn
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name: MODCOMP TeleGen2 Ada Host Compilation System 1.4A

Version: 1.4A

Host Computer System: MODCOMP Model 9730 with MODCOMP REAL/IX A.1

Target Computer System: Same as Host

Implementor's Declaration

I, the undersigned, representing TELESOFT, declare that TELESOFT has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration. I declare that MODCOMP is TELESOFT's licensee of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the licensee's corporate name.

Bonnie Wergaard Date: 2-9-90
for TELESOFT
Raymond A. Parra, Vice President and General Counsel

Licensee's Declaration

I, the undersigned, representing MODCOMP, declare that MODCOMP has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made in MODCOMP's corporate name.

David L. Klein Date: 2/7/90
MODCOMP
Name and Title: Vice President, Engineering

APPENDIX B

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 the TeleGen2 Ada Development System, as described in this Appendix, are provided by TeleSoft. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

     type INTEGER is range -32768 .. 32767;
     type LONG_INTEGER is range -2147483648 .. 2147483647;

     type FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;
     type LONG_FLOAT is digits 15
        range -8.98846567431158E+307 .. 8.98846567431158E+307;

     type DURATION is delta 2#1.0#E-14 range -86400.0 .. 86400.0;

   ...

end STANDARD;
```

CHAPTER 3: LRM ANNOTATIONS

CHAPTER CONTENTS

CHAPTER 3: LRM ANNOTATIONS

CHAPTER CONTENTS

3. LRM ANNOTATIONS

TeleGen2 compiles the full ANSI Ada language as defined by the *Reference Manual for the Ada Programming Language* (LRM) (ANSI/MIL-STD-1815A). This chapter describes the portions of the language that are designated by the LRM as implementation dependent for the compiler and run-time environment.

The information is presented in the order in which it appears in the LRM. In general, however, only those language features that are not fully implemented by the current release of TeleGen2 or that require clarification are included. The features that are optional or that are implementation dependent, on the other hand, are described in detail. Particularly relevant are the sections annotating LRM Chapter 13 (Representation Clauses and Implementation-Dependent Features) and Appendix F (Implementation-Dependent Characteristics).

3.1. LRM Chapter 2 - Lexical Elements

[LRM 2.1] Character Set. The host and target character set is the ASCII character set.

[LRM 2.2] Lexical Elements,Separators, and Delimiters. The maximum number of characters on an Ada source line is 200.

[LRM 2.8] Pragmas. TeleGen2 implements all language-defined pragmas *except* pragma Optimize. If pragma Optimize is included in Ada source, the pragma will have no effect. Optimization is implemented by using pragma Inline and the optimizer. Pragma Inline is not supported for library-level subprograms. Pragma Priority is not supported for main programs.

Limited support is available for pragmas Memory_Size, Storage_Unit, and System_Name; that is, these pragmas are allowed if the argument is the same as the value specified in the System package.

Pragmas Page and List are supported in the context of source/error listings; refer to the Compiler/Linker chapter of the TeleGen2 *User Guide* for more information.

3.2. LRM Chapter 3 - Declarations and Types

[LRM 3.2.1] Object Declarations. TeleGen2 does not produce warning messages about the use of uninitialized variables. The compiler will not reject a program merely because it contains such variables.

[LRM 3.5.1] Enumeration Types. The maximum number of elements in an enumeration type is 32767. This maximum can be realized only if generation of the image table for the type has been deferred, and there are no references in the program that would cause the image table to be generated. Deferral of image table generation for an enumeration type, P, is requested by the statement:

```
pragma Images (P, Deferred);
```

Refer to "Implementation-Defined Pragmas," in Section 3.8.1, for more information on pragma Images.

[LRM 3.5.4] **Integer Types.** There are two predefined integer types: Integer and Long_Integer. The attributes of these types are shown in Table 3-1. Note that using explicit integer type definitions instead of predefined integer types should result in more portable code.

Table 3-1. Attributes of Predefined Types Integer and Long_Integer

Attribute	Type	
	Integer	Long_Integer
'First	-32768	-2147483648
'Last	32767	2147483647
'Size	16	32
'Width	6	11

[LRM 3.5.8] **Operations of Floating Point Types.** There are two predefined floating point types: Float and Long_Float. The attributes of types Float and Long_Float are shown in Table 3-2. This floating point facility is based on the IEEE standard for 32-bit and 64-bit numbers. Note that using explicit real type definitions should lead to more portable code.

The type Short_Float is not implemented.

Table 3-2. Attributes of Predefined Types Float and Long_Float

Attribute	Type	
	Float	Long_Float
'Machine_Overflows	TRUE	TRUE
'Machine_Rounds	TRUE	TRUE
'Machine_Radix	2	2
'Machine_Mantissa	24	53
'Machine_Emax	127	1023
'Machine_Emin	−125	−1021
'Mantissa	21	51
'Digits	6	15
'Size	32	64
'Emax	84	204
'Safe_Emax	125	1021
'Epsilon	9.53674E−07	8.88178E−16
'Safe_Large	4.25253E+37	2.24711641857789E+307
'Safe_Small	1.17549E−38	2.22507385850721E−308
'Large	1.93428E+25	2.57110087081438E+61
'Small	2.58494E−26	1.99469227433161E−62

3.1. LRM Chapter 4 - Names and Expressions

[LRM 4.10] Universal Expressions. There is no limit on the accuracy of real literal expressions. Real literal expressions are computed using an arbitrary-precision arithmetic package.

3.2. LRM Chapter 9 - Tasks

[LRM 9.6] Delay Statements, Duration, and Time. This implementation uses 32-bit fixed point numbers to represent the type Duration. The attributes of the type Duration are shown in Table 3-3.

Table 3-3. Attributes of Type Duration

Attribute	Value
'Delta	0.000061035156250
'First	-86400.0
'Last	86400.0

[LRM 9.8] Priorities. Sixty-four levels of priority are available to associate with tasks through pragma Priority. The predefined subtype Priority is specified in the package System as

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

Currently the priority assigned to tasks without a pragma Priority specification is 31; that is:

```
(System.Priority'First + System.Priority'Last) / 2
```

[LRM 9.11] Shared Variables. The restrictions on shared variables are only those specified in the LRM.

3.3. LRM Chapter 10 - Program Structure and Compilation Issues

[LRM 10.1] Compilation Units - Library Units. All main programs are assumed to be parameterless procedures or functions that return an integer result type.

3.4. LRM Chapter 11 - Exceptions

[LRM 11.1] Exception Declarations. Numeric_Error is raised for integer or floating point overflow and for divide-by-zero situations. Floating point underflow yields a result of zero without raising an exception.

Program_Error and Storage_Error are raised by those situations specified in LRM Section 11.1. Exception handling is also discussed in the Programming Guide chapter.

3.7. LRM Chapter 13 - Implementation-Dependent Features

As shown in Table 3-4, the current release of TeleGen2 supports most LRM Chapter 13 facilities. The sections below the table document those LRM Chapter 13 facilities that are either not implemented or that require explanation. Facilities implemented exactly as described in the LRM are not mentioned.

Table 3-4. Summary of LRM Chapter 13 Features for TeleGen2

13.1 Representation Clauses	Supported, except as indicated below (LRM 13.2 – 13.5). Pragma Pack is supported, *except for* dynamically sized components. For details on the TeleGen2 implementation of pragma Pack, see Section 3.7.1.
13.2 Length Clauses	Supported: 'Size 'Storage_Size for collections 'Storage_Size for task activation 'Small for fixed-point types See Section 3.7.2 for more information.
13.3 Enumeration Rep. Clauses	Supported, *except for* type Boolean or types derived from Boolean. (Note: users can easily define a non-Boolean enumeration type and assign a representation clause to it.)
13.4 Record Rep. Clauses	Supported *except for* records with dynamically sized components. See Section 3.7.4 for a full discussion of the TeleGen2 implementation.
13.5 Address Clauses	*Supported for:* objects (including task objects). *Not supported for:* packages, subprograms, or task units. See Section 3.7.5 for more information.
13.5.1 Interrupts	For interrupt entries, the address of a TeleGen2-defined *interrupt descriptor* can be given. See "Interrupt Handling" in the Programming Guide chapter for more information.
13.6 Change of Representation	Supported, *except for* types with record representation clauses.

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

Table 3-4. Summary of LRM Chapter 13 Features for TeleGen2 (Contd)

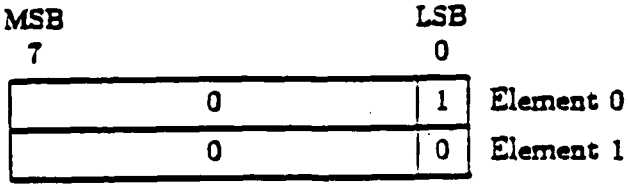
		----- *Continued from the previous page* ----
13.7	Package System	Conforms closely to LRM model. Refer to Section 3.7.7 for details on the TeleGen2 implementation.
13.7.1	System-Dependent Named Numbers	Refer to the specification of package System (Section 3.7.7).
13.7.2	Representation Attributes	Implemented as described in LRM *except that*: 'Address for packages is unsupported. 'Address of a constant yields a null address.
13.7.3	Representation Attributes of Real Types	See Table 3-2.
13.8	Machine Code Insertions	Fully supported. The TeleGen2 implementation defines an attribute, 'Offset, that, along with the language-defined attribute 'Offset, allows addresses of objects and offsets of data items to be specified in stack frames. Refer to "Using Machine Code Insertions" in the Programming Guide chapter for a full description on the implementation and use of machine code insertions.
13.9	Interface to Other Languages	Pragma Interface is supported for Assembly, C, UNIX, and Fortran. Refer to "Interfacing to Other Languages" in the Programming Guide chapter for a description of the implementation and use of pragma Interface.
13.10	Unchecked Programming	Supported except as noted below (LRM 13.10.2).
13.10.1	Unchecked Storage Deallocation	Supported.
13.10.2	Unchecked Type Conversions	Supported *except for* unconstrained record or array types.

3.7.1. Pragma Pack. This section discusses how pragma Pack is used in the TeleGen2 implementation.

a. With Boolean Arrays. You may pack Boolean arrays by the use of pragma Pack. The compiler allocates 16 bits for a single Boolean, 8 bits for a component of an unpacked Boolean array, and 1 bit for a component of a packed Boolean array. The first figure illustrates the layout of an unpacked Boolean array; the one below that illustrates a packed Boolean array:

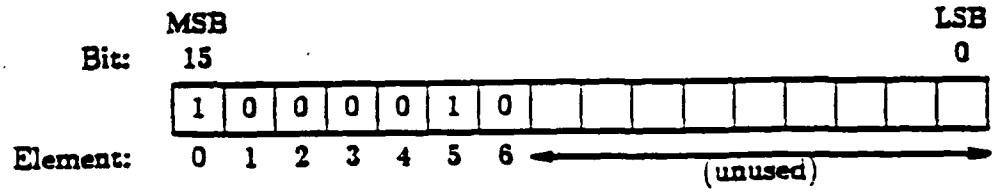
——— Unpacked Boolean array: ———

```
Unpacked_Bool_Arr_Type is array (Natural range 0..1) of Boolean
U_B_Arr: Unpacked_Bool_Arr_Type := (True,False);
```



——— Packed Boolean array: ———

```
Packed_Bool_Arr_Type is array (Natural range 0..6) of Boolean;
    pragma Pack (Packed_Bool_Arr_Type);
    P_B_Arr: Packed_Bool_Arr_Type := (P_B_Arr(0) => True,
      P_B_Arr(5) => True, others => False);
```



b. With Records. You may pack records by use of pragma Pack. Packed records follow these conventions:

1. The total size of the record is a multiple of 8 bits.
2. Packed records may cross word boundaries.
3. Records are packed to the bit level if the elements are themselves packed.

Below is an example of packing in a procedure, Rep_Proc, that defines three records of different lengths. Objects of these three packed record types are components of the packed record Rec. The storage allocated for Rec is 16 bits; that is, it is maximally packed.

```
procedure Rep_Proc is

  type A1 is array (Natural range 0 .. 8) of Boolean;
  pragma Pack (A1);

  type A2 is array (Natural range 0 .. 3) of Boolean;
  pragma Pack (A2);

  type A3 is array (Natural range 0 .. 2) of Boolean;
  pragma Pack (A3);

  type A_Rec is
    record
      One   : A1;
      Two   : A2;
      Three : A3;
    end record;
  pragma Pack (A_Rec);

  Rec   : A_Rec;

begin
  Rec.One       := ( 0 => True,   1 => False,  2 => False,
                     3 => False,  4 => True,   5 => False,
                     6 => False,  7 => False,  8 => True );
  Rec.Two (3)   := True;
  Rec.Three (1) := True;
end Rep_Proc;
```

3.7.2. [LRM 13.2] Length Clauses. A length clause specifies an amount of storage associated with a type. The sections below describe how length clauses are supported in this implementation of TeleGen2 and how to use length clauses effectively within the context of TeleGen2.

3.7.2.1. (a) Specifying Size: T'Size. The prefix T denotes an object. The size specification must allow for enough storage space to accommodate every allowable value of these objects. The constraints on the object and on its subcomponents (if any) must be static. For an unconstrained array type, the index subtypes must also be static.

For this implementation, Min_Size is the smallest number of bits logically required to hold any value in the range: no sign bit is allocated for non-negative ranges. Biased representations are not supported; e.g., a range of 100 .. 101 requires 7 bits, not 1. Warning: in the current release, using a size clause for a discrete type may cause inefficient code to be generated. For example, given...

```
type Nibble is range 0 .. 15;
for Nibble'Size use 4;
```

...each object of type Nibble will occupy only 4 bits, and relatively expensive bit-field instructions will be used for operations on Nibbles. (A single declared object of type Nibble will be aligned on a storage-unit boundary, however.)

For floating-point and access types, a size clause has no effect on the representation. (Task types are implemented as access types).

For composite (array or record) types, a size clause acts like an implicit pragma Pack, followed by a check that the resulting size is no greater than the requested size. Note that the composite type will be packed whether or not it is necessary to meet the requested size. The size clause for a record must be a multiple of storage units.

3.7.2.2. (b) Specifying Collection Size: T'Storage_Size. A collection is the entire set of objects created by evaluation of allocators for an access type.

The prefix T denotes an access type. Given an access type Acc_Type, a length clause for a collection allocated using Acc_Type objects might look like this:

```
for Acc_Type'Storage_Size use 64;
```

In TeleGen2, the above length clause allocates from the heap 64 bytes of contiguous memory for objects created by Acc_Type allocators. Every time a new object is created, it is put into the remaining free part of the memory allocated for the collection, provided there is adequate space remaining in the collection. Otherwise, a storage error is raised.

Keeping the objects in a contiguous span of memory allows system storage reclamation routines to deallocate and manage the space when it is no longer needed. Pragma Controlled can prevent the deallocation of a specified collection of objects. Objects can be explicitly deallocated by calling the Unchecked_Deallocation procedure instantiated for the object and access types.

Header Record

In this configuration of TeleGen2, information needed to manage storage blocks in a collection is stored in a collection header that requires 20 bytes of memory, adjacent to the collection, in addition to the value specified in the length clause.

Minimum Size

When an object is deallocated from a collection, a record containing link and size information for the space is put in the deallocated space as a placeholder. This enables the space to be located and reallocated. The space allocated for an object must therefore have the minimum size needed for the placeholder record. For this TeleGen2 configuration, this minimum size is the sum of the sizes of an access type and a integer type, or 6 bytes.

Dynamically Sized Objects

When a dynamically-sized object is allocated, a record requiring 2 bytes accompanies it to keep track of the size of the object for when it is put on the free list. The record is used to set the size field in the placeholder record since compaction may modify the value.

Size Expressions

Instead of specifying an integer in the length clause, you can use an expression to specify storage for a given number of objects. For example, suppose an access type Dict_Ref references a record Symbol_Rec containing five fields:

```
type Tag is String(1..8);

type Symbol_Rec;
type Dict_Ref is access Symbol_Rec;

type Symbol_Rec is
  record
    Left   : Dict_Ref;
    Right  : Dict_Ref;
    Parent : Dict_Ref;
    Value  : Integer;
    Key    : Tag;
   end record;
```

To allocate 10 Symbol_Rec objects, you could use an expression such as:

```
for Dict_Ref'Storage_Size use ((Symbol_Rec'Size * 10)+20);
```

where 20 is the extra space needed for the header record. (Symbol_Rec is obviously larger than the minimum size required, which is equivalent to one access type and one integer.)

In another implementation, Symbol_Rec might be a variant record that uses a variable length for the string Key:

```
type Symbol_Rec(Last : Natural :=0) is
  record
    Left   : Dict_Ref;
    Right  : Dict_Ref;
    Parent : Dict_Ref;
    Value  : Integer;
    Key    : String(1..Last);
   end record;
```

In this case, Symbol_Rec objects would be dynamically sized depending on the length of the string for Key. Using a length clause for Dict_Ref as above would then be illegal since Symbol_Rec'Size cannot be consistently determined. A length clause for Symbol_Rec objects, as described in (a) above, would be illegal since not all components of Symbol_Rec are static. As defined, a Symbol_Rec object could conceivably have a Key string with Integer'Last number of characters.

3.7.2.3. (c) Specifying Storage for Task Activation: T'Storage_Size. The prefix T denotes a task type. A length clause for a task type specifies the number of storage units to be reserved for an activation of a task of the type. The TeleGen2 default stack size is 8192 bytes.

3.7.2.4. (d) Specifying 'Small for Fixed Point Types: T'Small. Small is the absolute precision (a positive real number) while the prefix T denotes the first named subtype of a fixed point type. Elaboration of a real type defines a set of model numbers. T'Small is generally a power of 2, and model numbers are generally multiples of this number so that they can be represented exactly on a binary machine. All other real values are defined in terms of model numbers having explicit error bounds.

Example:

```
type Fixed is delta 0.25 range -10.0 .. 10.0;
```

Here...

Fixed'Small = 0.25 — *A power of 2*

3.0 = 12 * 0.25 — *A model number but not a power of 2*

The value of the expression of the length clause must not be greater than the delta of the first named subtype. The effect of the length clause is to use this value of 'Small for the representation of values of the fixed point base type. The length clause thereby also affects the amount of storage for objects that have this type.

If a length clause is not used, for model numbers defined by a fixed point constraint, the value of Small is defined as the largest power of two that is not greater than the delta of the fixed accuracy definition.

If a length clause is used, the model numbers are multiples of the specified value for Small. For this configuration of TeleGen2, the specified value must be (mathematically) equal to either an exact integer or the reciprocal of an exact integer.

Examples:

```
1.0, 2.0, 3.0, 4.0, . . . are legal
0.5, 1.0/3.0, 0.25, 1.0/3600.0 are legal
2.5, 2.0/3.0, 0.3 are illegal
```

3.7.3. [LRM 13.3] Enumeration Representation Clauses. Enumeration representation clauses are supported, except for Boolean types.

Performance note: Be aware that use of such clauses will introduce considerable overhead into many operations that involve the associated type. Such operations include indexing an array by an element of the type, or computing the 'Pos, 'Pred, or 'Succ attributes for values of the type.

3.7.4. [LRM 13.4] Record Representation Clauses. Since record components are subject to rearrangement by the compiler, you must use representation clauses to guarantee a particular layout. Such clauses are subject to the following constraints:

- Each component of the record must be specified with a component clause.
- The alignment of the record is restricted to mods 1 and 2, byte and word aligned.
- Bits are ordered right to left within a byte.
- Components may cross word boundaries.

Here is a simple example showing how the layout of a record can be specified by using representation clauses:

```
package Repspec_Example is
  Bits : constant := 1;
  Word : constant := 4;

  type Five is range 0 .. 16#1F#;
  type Seventeen is range 0 .. 16#1FFFF#;
  type Nine is range 0 .. 511;

  type Record_Layout_Type is record
      Element1 : Seventeen;
```

```
    Element2 : Five;
    Element3 : Boolean;
    Element4 : Nine;
  end record;

for Record_Layout_Type use record at mod 2;
    Element1 at 0*Word range 0 .. 16;
    Element2 at 0*Word range 17 .. 21;
    Element3 at 0*Word range 22 .. 22;
    Element4 at 0*Word range 23 .. 31;
  end record;

Record_Layout : Record_Layout_Type;
end Repspec_Example;
```

3.7.5. [LRM 13.5] Address Clauses. The Ada compiler supports address clauses for objects, subprograms, and entries. Address clauses for packages and task units are not supported.

Address clauses for objects may be used to access hardware memory registers or other known memory locations. The use of address clauses is affected by the fact that the System.Address type is private. For the MC680x0 target, literal addresses are represented as integers, so an unchecked conversion must be applied to these literals before they can be passed as parameters of type System.Address. For example, in the examples in this document the following declaration is often assumed:

```
function Addr is new Unchecked_Conversion (Long_Integer,System.Address);
```

This function is invoked when an address literal needs to be converted to an Address type. Naturally, user programs may implement a different convention. Below is a sample program that uses address clauses and this convention. Package System must be explicitly *with*ed when using address clauses.

```
with System;
with Unchecked_Conversion;
procedure Hardware_Access is
  function Addr is new Unchecked_Conversion (Long_Integer, System.Address);
  Hardware_Register : integer;
  for Hardware_Register use at Addr (16#FF0000#);
begin
  ...
end Hardware_Access;
```

When using an address clause for an object with an initial value, the address clause should immediately follow the object declaration:

```
Obj: Some_Type := <init_expr>;
for Obj use at <addr_expr>;
```

This sequence allows the compiler to perform an optimization wherein it generates code to evaluate the <addr_expr> as part of the elaboration of the declaration of the object. The expression <init_expr> will then be evaluated and assigned directly to the object, which is stored at <addr_expr>. If another declaration had intervened between the object declaration and the address clause, the compiler would have had to create a temporary object to hold the initialization value before copying it into the object when the address clause is elaborated. If the

object were a large composite type, the need to use a temporary could result in considerable overhead in both time and space. To optimize your applications, therefore, you are encouraged to place address clauses immediately after the relevant object declaration.

As mentioned above, arrays containing components that can be allocated in a signed or unsigned byte (8 bits) are packed, one component per byte. Furthermore, such components are referenced in generated code by MC680x0 byte instructions. The following example indicates how these facts allow access to hardware byte registers:

```
with System;
with Unchecked_Conversion;
procedure Main is
  function Addr is new Unchecked_Conversion (Long_Integer, System.Address);
  type Byte is range -128..127;
  HW_Regs : array (0..1) of Byte;
  for HW_Regs use at Addr (16#FFF310#);

  Status_Byte : constant integer := 0;
  Next_Block_Request: constant integer := 1;
  Request_Byte : Byte := 119;
  Status : Byte;

begin
  Status := HW_Regs(Status_Byte);
  HW_Regs(Next_Block_Request) := Request_Byte;
end Main;
```

Two byte hardware registers are referenced in the example above. The status byte is at location 16#FFF310# and the next block request byte is at location 16#FFF311#.

Function Addr takes a long integer as its argument. Long_Integer'Last is 16#7FFFFFFF#, but there are certainly addresses greater than Long_Integer'Last. Those addresses with the high bit set, such as FFFA0000, cannot be represented as a positive long integer. Thus, for addresses with the high bit set, the address should be computed as the negation of the 2's complement of the desired address. According to this method, the correct representation of the sample address above would be Addr(-16#00060000#).

3.7.6. [LRM 13.6] Change of Representation. TeleGen2 supports changes of representation, except for types with record representation clauses.

3.7.7. [LRM 13.7] The Package System. The specification of TeleGen2's implementation of package System is presented in the LRM Appendix F section at the end of this chapter.

3.7.8. [LRM 13.7.2] Representation Attributes. The compiler does not support 'Address for packages.

3.7.9. [LRM 13.7.3] Representation Attributes of Real Types. The representation attributes for the predefined floating point types were presented in Table 3-2.

3.7.10. [LRM 13.8] Machine Code Insertions. Machine code insertions, an optional feature of the Ada language, are fully supported in TeleGen2. Refer to the "Using Machine Code Insertions" section in the Programming Guide chapter for information regarding their

implementation and for examples on their use.

3.7.11. [LRM 13.9] Interface to Other Languages. In pragma Interface is supported for Assembly, C, UNIX, and Fortran. Refer to "Interfacing to Other Languages" in the Programming Guide chapter for for information on the use of pragma Interface. TeleGen2 does not currently allow pragma Interface for library units.

3.7.12. [LRM 13.10] Unchecked Programming. Restrictions on unchecked programming as it applies to TeleGen2 are listed in the following paragraphs.

[LRM 13.10.2] **Unchecked Type Conversions.** Unchecked conversions are allowed between types (or subtypes) T1 and T2 as long as they are not unconstrained record or array types.

3.8. LRM Appendix F for TeleGen2

The Ada language definition allows for certain target dependencies. These dependencies must be described in the reference manual for each implementation, in an "Appendix F" that addresses each point listed in LRM Appendix F. Table 3-5 constitutes Appendix F for this implementation. Points that require further clarification are addressed in sections refererenced in the table.

Table 3-5. LRM Appendix F for TeleGen2

(1) Implementation-Dependent Pragmas	(a) Implementation-defined pragmas: Comment, Linkname, Images, and No_Suppress (Section 3.8.1). (b) Predefined pragmas with implementation-dependent characteristics: • Interface (assembly, UNIX, C, and Fortran—see "Interfacing to Other Languages." Not supported for library units. • List and Page (in context of source/error compiler listings.) (See the *User Guide*.) • Pack. See Section 3.7.1. • Inline. Not supported for library-level subprograms. • Priority. Not supported for main programs. *Other supported predefined pragmas:* Controlled Shared Suppress Elaborate *Predefined pragmas partly supported* (see Section 3.1): Memory_Size Storage_Unit System_Name *Not supported:* Optimize
(2) Implementation-Dependent Attributes	'Offset. Used for machine code insertions. The predefined attribute 'Address is not supported for packages. See "Using Machine Code Insertions" earlier in this chapter for information on 'Offset and 'Address. 'Extended_Image 'Extended_Value 'Extended_Width 'Extended_Aft 'Extended_Digits Refer to Section 3.8.2 for information on the implementation-defined extended attributes listed above.
(3) Package System	See Section 3.7.7.
(4) Restrictions on Representation Clauses	Summarized in Table 3-4.

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

Table 3-5. LRM Appendix F for TeleGen2 (Contd)

------ *Continued from the previous page* ------	
(5) Implementation-Generated Names	None
(6) Address Clause Expression Interpretation	An expression that appears in an object address clause is interpreted as the address of the first storage unit of the object.
(7) Restrictions on Unchecked Conversions	Summarized in Table 3-4.
(8) Implementation-Dependent Characteristics of the I/O Packages.	1. In Text_IO, the type Count is defined as follows: type Count is range 0..System.Max_Text_IO_Count; -- or 0..Max_Int-1 OR 0..2_147_483_646 2. In Text_IO, the type Field is defined as follows: subtype Field is integer range System.Max_Text_IO_Field; 3. In Text_IO, the Form parameter of procedures Create and Open is not supported. (If you supply a Form parameter with either procedure, it is ignored.) 4. Sequential_IO and Direct_IO cannot be instantiated for unconstrained array types or discriminated types without defaults. 5. The standard library contains preinstantiated versions of Text_IO.Integer_IO for types Integer and Long_Integer and of Text_IO.Float_IO for types Float and Long_Float. We suggest that you use the following to eliminate multiple instantiations of these packages: Integer_Text_IO Long_Integer_Text_IO Float_Text_IO Long_Float_Text_IO

3.8.1. Implementation-Defined Pragmas. There are four implementation-defined pragmas in TeleGen2: pragmas Comment, Linkname, Images, and No_Suppress.

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

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

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

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

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

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

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

A simple example of the use of pragma Linkname is:

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

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

The syntax of this pragma is:

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

The default, Deferred, saves space in the literal pool by not creating image and index tables for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If one of these attributes is used, the tables are generated in the literal pool of the compilation unit in which the attribute appears. If the attributes are used in more than one compilation unit. more than one set of tables is generated, eliminating the benefits of deferring the table. In this case, using

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

will cause a single image table to be generated in the literal pool of the unit declaring the enumeration type.

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

pragma Images(<enumeration_type>, Immediate);

or the 'Image, 'Value, or 'Width attribute for the type will result in an error message from the compiler.

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

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

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

where <identifier> is the type of check you want to suppress (e.g., access_check; refer to LRM 11.7)

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

If neither Suppress nor No_Suppress are present in a program, no checks will be suppressed. You may override this default at the command level, by compiling the file with the –i(nhibit option and specifying with that option the type of checks you want to suppress. For more information on –i(nhibit, refer to your TeleGen2 *Overview and Command Summary* document.

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

3.8.2. Implementation-Dependent Attributes.

3.8.2.1. 'Address and 'Offset. These were discussed within the context of using machine code insertions, in the Programming Guide chapter.

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

1. Extended attributes take more parameters and allow control of the format of the output string.
2. Extended attributes are defined for all scalar types, including fixed and floating point types.

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

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

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

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

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


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

3.8.2.2.1. Integer Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Width,Base,Based,Space_IF_Positive)

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

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

Parameter Descriptions:

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

Examples:

Suppose the following subtype were declared:

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

Then the following would be true:

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

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

'Extended_Value

Usage:

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

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

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

Parameter Description:

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

Examples:

Suppose the following subtype were declared:

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

Then the following would be true:

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

'Extended_Width

Usage:

```
X'Extended_Width(Base,Based_Space_If_Positive)
```

Returns the width for subtype of X.

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

Parameter Descriptions:

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

Examples:

Suppose the following subtype were declared:

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

Then the following would be true:

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

3.8.2.2.2. Enumeration Type Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Width,Uppercase)

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

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

Parameter Descriptions:

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

Examples:

Suppose the following types were declared:

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

Then the following would be true:

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

'Extended_Value

Usage:

X'Extended_Value(Item)

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

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

Parameter Descriptions:

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

Examples:

Suppose the following type were declared:

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

Then the following would be true:

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

'Extended_Width

Usage:

X'Extended_Width

Returns the width for subtype of X.

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

Parameter Descriptions:

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

Examples:

Suppose the following types were declared:

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

Then the following would be true:

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

3.8.2.2.3. Floating Point Attributes

'Extended_Image

Usage:

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

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

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

Parameter Descriptions:

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

Examples:

Suppose the following type were declared:

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

Then the following would be true:

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

'Extended_Value

Usage:

X'Extended_Value(Item)

Returns the value associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it skips any leading zeros, then reads a plus or minus sign if present then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.2.8:9, 14.3.8:10)

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

Parameter Descriptions:

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

Examples:

Suppose the following type were declared:

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

Then the following would be true:

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

'Extended_Digits

Usage:

X'Extended_Digits(Base)

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

Parameter Descriptions:

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

Examples:

Suppose the following type were declared:

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

Then the following would be true:

```
X'Extended_Digits  = 5
```

3.8.2.2.4. Fixed Point Attributes

'Extended_Image

Usage:

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

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

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

Parameter Descriptions:

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

Examples:

Suppose the following type were declared:

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

Then the following would be true:

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

'Extended_Value

Usage:

X'Extended_Value(Image)

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

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

Parameter Descriptions:

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

Examples:

Suppose the following type were declared:

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

Then the following would be true:

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

'Extended_Fore

Usage:

X'Extended_Fore(Base,Based)

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

Parameter Descriptions:

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

Examples:

Suppose the following type were declared:

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

Then the following would be true:

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

'Extended_Aft

Usage:

X'Extended_Aft(Base,Based)

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

Parameter Descriptions:

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

Examples:

Suppose the following type were declared:

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

Then the following would be true:

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

3.8.3. Package System. The current specification of package System is provided below.

```
package System is

   type Address is access integer;

   type Subprogram_Value is private;

   type Name     is (TeleGen2);

   System_Name   : constant name := TeleGen2;

   Storage_Unit : constant := 8;
   Memory_Size  : constant := (2 ** 31)  -1;

   -- System-Dependent Named Numbers:

   -- See Table 3-2 for the values for attributes of
   -- types Float and Long_Float

   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);
   Tick          : constant := 10.0E-3;

   -- Other System-Dependent Declarations

   subtype Priority is integer range 0 .. 63;

   Max_Object_Size    : constant := Max_Int;
   Max_Record_Count   : constant := Max_Int;
   Max_Text_IO_Count  : constant := Max_Int -1;
   Max_Text_IO_Field  : constant := 1000;

private
   -- ...
end System;
```

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	199 * 'A' & '1'
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	199 * 'A' & '2'
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except	100 * 'A' & '3' & 99 * 'A'

Name and Meaning	Value
for a character near the middle.	
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	100 * 'A' & '4' & 99 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	197 * '0' & "298"
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	195 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 100 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 99 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	180 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_646
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2147483647
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	TELEGEN2
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	1000
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	131_073.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	63
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BADCHAR*^/%
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/NONAME/DIRECTORY

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32768
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-131_073.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	200
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:" & 195 * '0' & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 193 * '0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 198 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE_AVAILABLE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	TELEGEN2
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

Name and Meaning	Value
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2147483647
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	TELEGEN2
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C This test expects that the string "-- TOP OF PAGE. -- 63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

b. A39005G This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E This test contains an unitended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING-_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests] These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests] These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

i. CD2B15C & CD7205C These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C & D, ED7006C & D [5 tests] These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B, & CD7204B These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is

expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

q. CE3111C This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INFUT (lines 103, 107, 118, 132, & 136).

s. CE3411B This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

References and page numbers in this appendix are consistent with compiler documentation and not with this report.

2. COMPILATION TOOLS

This chapter discusses the commands to invoke the TeleGen2 components that are associated with the process of compilation. The components are the compiler (invoked by the *ada* command; see Section 2.1) and the linker (invoked by the *ald* command; see Section 2.2).

Optimization is part of the compilation process as well. In the TeleGen2 documentation set, however, optimization is discussed separately from compilation. In this volume, the commands associated with optimization (*ada* *-O*; *aopt*) are discussed in the "Other Tools" chapter. (One exception is the Option Summary table below, where *aopt* options are included for comparison.)

Table 2-1 summarizes the options that are used by the compilation tools. Note that several options are common to the commands shown.

Table 2-1. Compilation Tools Option Summary

Option	Command		
	ada	*ald*	*aopt*
-l(ibfile	x	x	x
-t(emplib	x	x	x
-V(space_size	x	x	x
-v(erbose	x	x	x
-b(ind_only		x	
-C(ontext	x		
-D(elay_NonPreempt		x	
-d(ebug	x		
-E(rror_abort	x		
-e(rrors_only	x		
-F(ile_only_errs	x		
-G(raph	x		x
-I(nline	x		x
-i(nhibit	x		
-k(eep	x		x
-L(ist	x		
-m(ain	x		
-N(ame			x
-O(ptimize	x		x
-o(utput		x	
-S[a]	x	x	x
-s(oftware_float	x	x	x
-T(raceback		x	
-u(pdate_lib	x		
-w("timeslice"		x	
-X(ception_show		x	
-x(ecution_profile	x	x	x
-Y and -y		x	

Note

a: The functionality of the -S option of *ada* and the -S option of *ald* is somewhat different. Refer to the text.

2.1. The Ada Compiler ("ada")

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

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

This section focuses specifically on command-level information relating to compilation, that is, on invoking the compiler and using the various options to control the compilation process. Details on the TeleGen2 compilation process and guidelines for using the compiler most effectively are in the Compiler chapter of the *User Guide* volume. (You might want to look at Figure 3-1 in that volume right now, to give you insight into the TeleGen2 compilation process and to see how the options mentioned in this Command Summary volume relate to the actual compilation process.)

The syntax of the command to invoke the Ada compiler is:

```
ada {<"common_option">} {<option>} <input_spec>
```

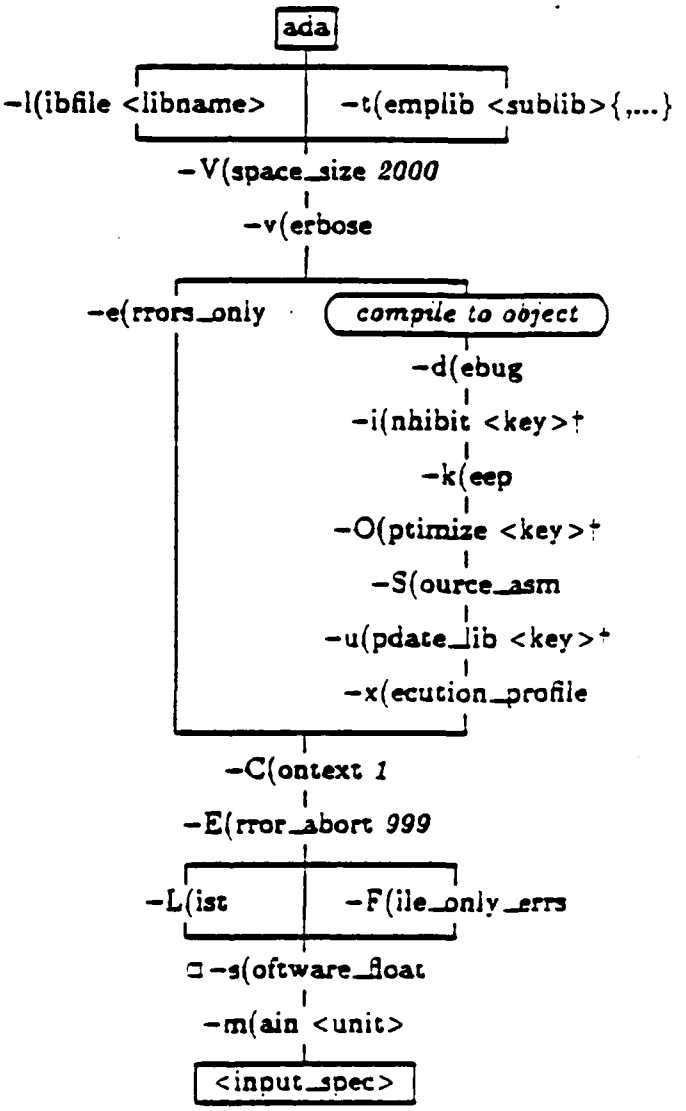
where:

<"common_option">	None or more of the following set of options that are common to many TeleGen2 commands: -l(ibfile or -t(emplib -V(space_size -v(erbose These options were discussed in Chapter 1.
<option>	None or more of the compiler-specific options discussed below.
<input_spec>	The Ada source file(s) to be compiled. It may be: • One or more Ada source files, for example: /user/john/example Prog_A.text ciosrc/calc_mem.ada calcio.ada myprog.ada *.ada • A file containing names of files to be compiled. Such a file must have the extension ".ilf". You can find details for using input-list files in the *User Guide* portion of your TeleGen2 documentation set. • A combination of the above.

Please note that the compiler defaults are set for your convenience. In most cases you will not need to use additional options: a simple "ada <input_spec>" is sufficient. However, options

are included to provide added flexibility. You can, for example, have the compiler quickly check the source for syntax and semantic errors but not produce object code (−e(rrors_only) or you can compile, bind, and link an 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. Remember that each of the options listed is identified by a dash followed by a single letter (e.g., "−e"). The parenthesis and the characters following the option are for descriptive purposes only; they are not part of the option.



† (1) <key> for −O: refer to *asopt*. (2) <key> for −u: i or s: s is the default. (3) <key> for −x: a or certain combinations of inc.

The options available with the *ada* command are summarized in Table 2-2. The default situation (that is, what happens if the option is not used) is explained in the middle column. Each option is described in the paragraphs that follow the table.

Table 2-2. Summary of Compiler Options

Option	Default	Discussed in Section
Common options:		
-l(ibfile <libname>	Use liblst.alb as the library file.	1.4.1
-t(emplib <sublib...>	None	1.4.1
-V(space_size <value>	Set size to 2000 Kbytes.	1.4.2
-v(erbose	Do not output progress messages.	1.4.3
-d(ebug	Do not include debug information in object code. (*-d* sets -k(eep.)	2.1.1
-E(rror_abort <value>	Abort compilation after 999 errors.	2.1.2
-e(rrors_only	Run middle pass and code generator, not just front end.	2.1.3
-i(nhibit <key>†	Do not suppress run-time checks, source line references, or subprogram name information in object.	2.1.4
-k(eep	Discard intermediate representations of secondary units.	2.1.5
-m(ain <unit>	Do not produce executable code (binder/linker not executed).	2.1.6
-O(ptimize <key>†	Do not optimize code.	2.1.7
-s(oftware_float	Use hardware floating-point support.	2.1.8
-u(pdate_lib <key>†	Do not update library when errors are found (muiti-unit compilations).	2.1.9
-x(ecution_profile	Do not generate execution-profile code.	2.1.10
Listing options:		
-C(ontext <value>	Include 1 line of context with error message.	2.1.11.1
-L(ist	Do not generate a source-error listing.	2.1.11.2
-F(ile_only_errs	Do not generate an errors-only listing. only.	2.1.11.3
-S(ource_asm	Do not generate assembly listing.	2.1.11.4

2.1.1. -d(ebug – Generate Debugger Information. The code generator must generate special information for any unit that is to be used with the TeleGen2 symbolic debugger. The generation of this information is enabled by use of the *-d* option. The use of *-d* automatically

† (1) <key> for *-O*: refer to *-opt*. (2) <key> for *-u*: i or s; s is the default. (3) <key> for *-i*: a or certain combinations of lmc.

sets the -k(eep option. This to make sure that the High Form, Low Form, and debugger information for secondary units are not deleted.

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. The default situation is that no debugger information is produced.

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.

2.1.2. -E(rror_abort - Set an Error Count for Aborting Compilation. The compiler maintains separate counts of all syntactic errors, semantic errors, and warning messages detected by the front end during a compilation.

A large number of errors generally indicates that errors in statements appearing earlier in the unit have "cascaded" through the rest of the code. A classic example is an error occurring in a statement that declares a type. This causes subsequent declarations that use the type to be in error, which further causes all statements using the declared objects to be in error. In such a situation, only the first error message is useful. Aborting the compilation at an early stage is therefore often to your advantage; the *-E* option allows you to do it.

The format of the option is:

-E <value>

where <value> is the number of errors or warnings allowed. The default value is 999. The minimum value is 1. **Caution:** If you do not use the *-E* option, it is possible to have 999 warning messages *plus* 999 syntax errors *plus* 999 semantic errors without aborting compilation, since each type of error is counted separately.

2.1.3. -e(rrors_only - Check Source But Don't Generate Code. This 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, of course, that only front end errors are detected and that only the High Form intermediates are generated.) This option is typically used during early code development where execution is not required and speed of compilation is important.

Note: Although High Form intermediates are generated with the *-e* option, these intermediates are deleted at the end of compilation. This means that the library is not updated.

The *-e* option cannot be used with -S(ource_asm, since the latter requires the generation of object code. If *-e* is not used (the default situation), the source is compiled to object code (if no errors are found). The *-e* option is also incompatible with -k(eep, -d(ebug, -O(ptimize, and other options that require processing beyond the front end phase of compilation.

2.1.4. -i(nhibit - Suppress Checks and Source Information. 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 Ada language requires a wide variety of *run-time checks* to ensure the validity of operations. For example, arithmetic overflow checks are required on all numeric operations, and range checks are required on all assignment statements that could result in an illegal value being assigned to a variable. While these 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.

Although the Ada language provides pragma Suppress to selectively suppress classes of checks, using the pragma requires you to modify the Ada source. The –i(nhibit option provides an alternative mechanism.

The compiler by default stores *source line and subprogram name information* in the object code. This information is used to display a source level traceback when an unhandled exception propagates to the outer level of a program: it is particularly valuable during development, since it provides a direct indication of the source line at which the exception occurs and the subprogram calling chain that led to the line generating the exception.

The inclusion of source line information in the object code, however, 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. 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; but it can be suppressed with the –i(nhibit option. When source line information is suppressed, the traceback indicates the offset of the object code at which the exception occurs instead of the source line number. When subprogram name information is suppressed, the traceback indicates the offsets of the subprogram calls in the calling chain instead of the subprogram names. (For more information on the traceback function, refer to the Programming Guide chapter in your *Reference Information* volume.)

The format of the –i(nhibit option is:

–i <suboption>{<suboption>}

where <suboption> is one or more of the single-letter suboptions listed below. Combinations of suboptions are possible. 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.
n[ame_info]	Suppress subprogram name information in object code.
c[hecks]	Suppress run-time checks – elaboration, overflow, storage access, discriminant, division, index, length, and range checks.
a[ll]	Suppress source line information, subprogram name information, and run-time checks. In other words, a (=inhibit all) is equivalent to lnc.

As an example of use, the command...

ada –v –i lc my_file.ada

...inhibits the generation of source line information and run-time checks in the object code of the units *my_file.ada*.

2.1.5. -k(eep - Retain Intermediate Forms. As a default, the compiler deletes the High Form and Low Form intermediate representations of all compiled secondary units from the working sublibrary. Deletion of these intermediate forms can significantly decrease the size of sublibraries — typically 50% to 80% for multi-unit programs. On the other hand, some of the information within the intermediate forms may be required later. For example, High Form is required if the unit is to be referenced by the Ada Cross-Referencer (*axr*). In addition, information required by the debugger and the Global Optimizer must be saved if these utilities are used. For these reasons, the *-k* option is provided with the *ada* command. The *-k* option:

- Must be used if the compiled unit is to be optimized later with *aopt*; otherwise, *aopt* issues an error message and the optimizer aborts.
- Should be used if the unit is to be cross-referenced later; otherwise, an error message is issued when the Ada Cross-Referencer attempts to cross-reference that unit.
- Need not be used with -d(ebug, since *-k* is set automatically whenever *-d* is used.

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

2.1.6. -m(ain - Compile a Main Program. 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 by calling *ald* 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, it does not have to be in the input file. However, the body of the main unit, if previously compiled, must be present in the current working sublibrary.

Note: Options specific to the linker (invoked via *ald*) may be specified on the *ada* command line when the *-m* option is used. With *-m*, the compiler will call *ald* when compilation is complete, passing to it *ald*-specific options specified with the *ada* command. For example...

ada -m welcome -T 2 -o new sample.ada

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

2.1.7. -O(ptimize - Optimize Object Code. This option causes the compiler to invoke the global optimizer to optimize 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. The format of this option is:

-O <key>

where <key> is at least one of the optimizer suboption keys discussed in the Global Optimizer chapter. Please refer to that chapter for all information regarding the use of the optimizer. The chapter discusses using the optimizer as a standalone tool for collections of compiled but unoptimized units and using the -O(ptimize option with the *ada* command. The latter topic includes a definition of the -O(ptimize suboption key values plus a presentation of two other *ada* options (-G(raph and -I(nline_list, not shown on the *ada* chart) that may be used in conjunction with the -O(ptimize option. *Note:* We strongly recommend that you do not attempt to use the optimizer until the code being compiled has been fully debugged and tested.

2.1.8. -s(oftware_float – Use Software Floating-Point Support. *This option may not be available with your TeleGen2 system; please consult the Overview portion to see if it is provided.* The Ada linker selects hardware floating-point support by default. If you do not have hardware floating point support or if you wish to generate code compatible with such machines, use the *-s* option. In addition: If you use the *-s* option, the library file you use for compilation must contain the the name of the *software* floating point run-time sublibrary, *s_rt.sub*. Refer to the Library Manager chapter in your *User Guide* volume for more information on the run-time sublibrary.

2.1.9. -u(pdate_lib – Update the Working Sublibrary. The -u(pdate_lib option tells the compiler when to update the library. It is most useful for compiling multiple source files. The format of the option is:

-u <key>

where <key> is either "s" (source) or "i" (invocation).

i "i" tells the compiler to update the working sublibrary after all files submitted in that invocation of *ada* have compiled successfully. If an error is encountered, the library is not updated, even for source files that compile successfully. In addition, all remaining source files will be compiled for syntactic and semantic errors only. *Implications:* (1) If an error exists in any source file you submit, the library will not be updated, even if all other files are error free. (2) Compilation is faster, since the library is updated only once, at the end of compilation.

s (This is the default; it is equivalent to not using the -u(pdate_lib option at all.) "s" tells the compiler to update the library after all units within a single source file compile successfully. If the compiler encounters an error in any unit within a source file, all changes to the working sublibrary for the erroneous unit and for all other units in the file are discarded. However, library updates for units in previous or remaining source files are unaffected. *Implications:* (1) You can submit files containing possible errors and still have units in other files compile successfully into the library. (2) Compilation is slightly slower, since the library is updated once for each file.

Therefore:

Use "u s" (or no -u(pdate option) when:

You're not sure all units will compile successfully.
Compilation speed is not especially important.

Use "u i" when:

You are reasonably certain your files will compile successfully.
Fast compilation is important.

2.1.10. -x(ecution_profile - Generate Profile Information. The -x(ecution_profile option uses the code generation phase of compilation to place special information in the generated code that can be used to obtain a "profile" of a program's execution. This information is generated by a facility known as "the profiler." Refer to your *User Guide* volume for information on how to use the profiler to obtain execution timing and subprogram call information for a program.

Important: If any code in a program has been compiled with the -x(ecution_profile option, that option must also be used with *ald* when the program is bound and linked. Otherwise, linking aborts with an error such as "Undefined RECORDSSCURRENT".

2.1.11. Listing Options. The listing options specify the content and format of listings generated by the compiler. Assembly code listings of the generated code can also be generated.

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

The format of the option is:

-C <value>

where <value> is the number of source context lines output for each error. The default for <value> 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. Let's say that *trialprog.ada*, which consists of the following text...

```
package P is
  type T1 is range 1..10;
  type T2 is digits 1;
  type Arr is array (1..2) of integer; type T3 is new Arr;     -- OK.

    package Inner is
      type In1 is new T1;      -- ERROR: T1 DERIVED.
      type In2 is new T2;      -- ERROR: T2 DERIVED.
      type In3 is new T3;      -- ERROR: T3 DERIVED.
      type Inarr is new Arr;   -- OK.

    end Inner;

end P;
```

...were compiled as follows:

```
ada  -e  -C 2  trialprog.ada
```

(The -e option here is used for error checking and -C(ontext is set to 2 to display two lines of source.) The output produced would look like this:

```
      7:     package Inner is
      8:       type In1 is new T1;      -- ERROR: T1 DERIVED.
                                ------
>>>     Illegal parent type for derivation <3.4:15,7.4.1:4>


      8:       type In1 is new T1;      -- ERROR: T1 DERIVED.
      9:       type In2 is new T2;      -- ERROR: T2 DERIVED.
                                ------
>>>     Illegal parent type for derivation <3.4:15,7.4.1:4>


      9:       type In2 is new T2;      -- ERROR: T2 DERIVED.
     10:       type In3 is new T3;      -- ERROR: T3 DERIVED.
                                ------
>>>     Illegal parent type for derivation <3.4:15,7.4.1:4>
```

2.1.11.2. -L(ist - Generate a Source Listing. This 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_spec>.l. where <file_spec> is the name of the source file (minus the extension). If <file_spec>.l already exists. it is overwritten.

If input to the *ada* command is an input-list file (<file_spec>.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.

2.1.11.3. -F(ile_only_errs - Put Only Errors in Listing File. This option is used to produce a listing containing only the errors generated during compilation; source is not included. The output is sent to <file_spec>.l. -F is incompatible with -L.

2.1.11.4. -S(ource_asm - Generate a Source/Assembly Listing. This option instructs the compiler to generate an assembly listing and send it to a file named <unit>.<ext>, where <unit> is the name of the unit in the user-supplied source file and <ext> is the file extension (it may be "s" or something else, depending on your configuration). The listing consists of assembly code intermixed with source code as comments. If input to the *ada* command is an input-list file (<file_spec>.ilf), a separate assembly listing file is generated for each unit contained in each source file listed in the input file. If *-S* is not used (the default situation), an assembly listing is not generated.

2.2. The Ada Linker ("ald")

The TeleGen2 Ada Compiler produces object code from Ada source code. The TeleGen2 Ada Linker takes the object (of a main program) that is produced by the compiler and produces a UNIX executable module. The TeleGen2 Ada Linker will be called "the linker" in the remainder of this manual.

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.

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

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 directory where *ald* was executed, under the name of the main unit used as the argument to *ald*. For example, the command

```
ald  main
```

links the object modules of all the units in the extended family of the unit Main. The name of the resulting executable file will simply be "main". Important: When using the *ald* command, the body of the main unit to be prelinked must be in the working sublibrary.

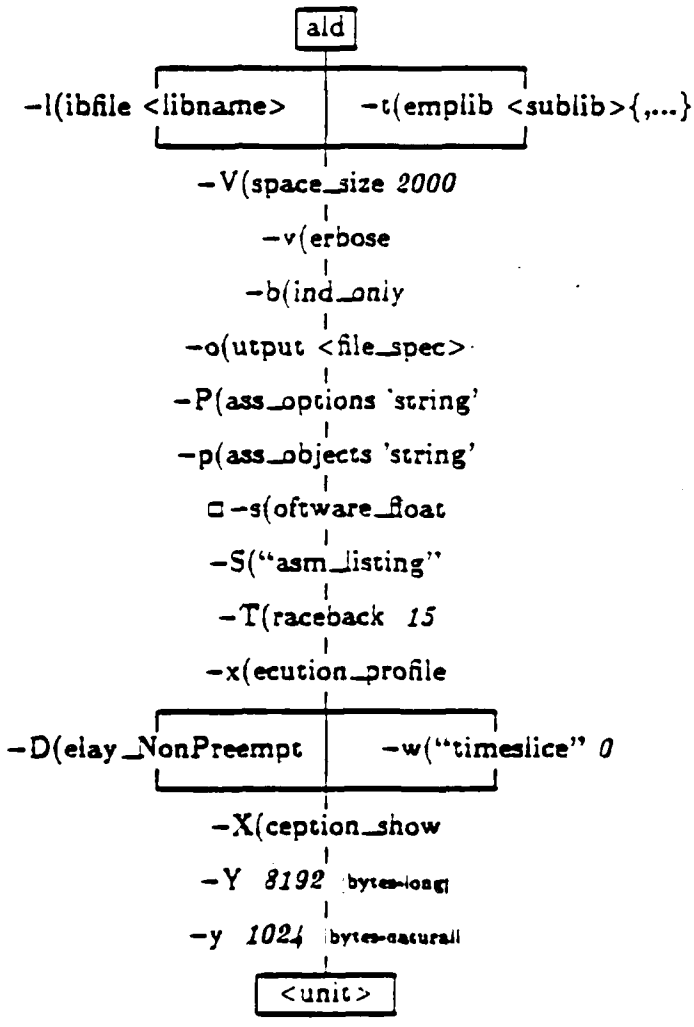
The general syntax of the *ald* command is:

```
ald  {<"common_option">}  {<option>}  <unit>
```

where:

<"common_option">	None or more of the following set of options that are common to many TeleGen2 commands: −l(ibfile or −t(emplib −V(space_size −v(erbose These options were discussed in Chapter 1.
<option>	None or more of the options discussed in the following sections.
<unit>	The name of the main unit of the Ada program to be linked.

The options available with the *ald* command and the relationships among them are illustrated below.



2.2.1. -b(ind_only - Produce Elaboration Code Only. To provide you with more control over the linking process, the *-b* option causes the linker to abort after it has created the elaboration code and the linking order, but before invoking the UNIX link editor. This option allows you to edit the link order for special applications and then invoke the link editor directly. The link order is contained in 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 your working directory. To complete the link process, enter "<unit>.lnk".

The name of the file containing the elaboration code is <unit>.obm, 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.

2.2.2. -o(utput - Name the Output File. This option allows you to specify the name of the output file produced by the linker. For example, the command...

```
ald -o yorkshire main
```

...causes the linker to put the executable module in the file *yorkshire*.

2.2.3. -P(ass_Options – Pass Options to the Linker. This option allows you to pass a string of options directly to the UNIX link editor. For example, the command

```
ald -P '-t -r' main
```

adds the string "-t -r" to the options of the link editor when it is invoked. The options must be quoted (double or single quotes).

2.2.4. -p(ass_objects – Pass Arguments to the Linker. This option allows you to pass a string of arguments directly to the UNIX link editor. For example, the command

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

causes the link editor to link the object file *cosine.o* (which it expects to find in the current working directory), and to search the library */usr/lib/libm.a* for unresolved symbol references. (The location of the *libm.a* library may be different on your system.) Remember that the 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 the *-p* option 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

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

causes the link editor to search the directories */lib* and */usr/lib* (in that order) for file *libxyz.a*.

2.2.5. -S("asm_listing" – Produce an Assembly Listing. The *-S* option is used to output an assembly listing from the elaboration process. The output is put in a file, <file>.obm.s, where <file> is the name of the main unit being linked. (The file extension may be different on your system.)

-s(oftware_float – Use Software Floating-Point Support. *This option may not be available on your TeleGen2 system. Please consult the Overview portion of this volume to see if it is provided.* The Ada linker currently selects hardware floating-point support by default. This default situation is provided for users of systems with an arithmetic coprocessor. If you do not have hardware floating point support or if you wish to generate code compatible with such machines, use the *-s* option. *In addition:* if you use the *-s* option, the library file you use for compiling and linking must contain the name of the *software* floating point run-time sublibrary, *s_rt.sub*. Refer to the Library Manager chapter in your *User Guide* volume for more information on the run-time sublibrary.

2.2.7. -T(raceback – Set Levels for Tracing Exceptions. When a run-time exception occurs (and is not handled by an exception handler), the name and line number of the unit where the exception occurred is displayed along with a recursive history of the units which called that unit. (See the "Exception Handling" section in the Programming Guide chapter of your *Reference Information* volume for a more complete explanation of exception reports.) The *-T* option allows you to set the number of levels in this recursive history. For example, the

command

```
ald -T 3 main
```

specifies that traceback histories will be three levels deep. The default value for this option is 15.

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

2.2.8. -x(ecution_profile - Bind and Link for Profiling. This option is used for units that have been compiled with the *-x* option. Use of *-x* with *ada* causes the code generator to include, in the object, special code that will later be used to provide a profile of the program's execution.

If *-x* is used with *ada*, it must be used with *ald* as well. The *-x* option of *ald* instructs the linker to link in the profiling run-time support routines and generate a subprogram dictionary, *profile.dic*, for the program. The dictionary is a text file containing the names and addresses of all subprograms in the program. The dictionary can be used to produce a listing showing how the program executes.

Refer to the Ada Profiler chapter in your *User Guide* volume for a full discussion of the profiler.

2.2.9. Tasking Options. The following *ald* options are binding options used for task execution. They are therefore useful only for linking programs that contain tasking code.

2.2.9.1. -D(elay_NonPreempt - Specify Non-Preemptive Delay. By default, the TeleGen2 run-time is set for preemptive delay handling. That is, an active task is preempted if another task is waiting that has a priority equal to or greater than that of the active task.

The *-D* option allows you to specify *non*-preemptive delay handling. With non-preemptive delay, a task is scheduled only when a synchronization point is reached. -D(elay_NonPreempt is incompatible with the *-w* option (see below).

2.2.9.2. -w("timeslice" - Limit Task Execution Time. The *-w* option allows you to define the maximum time a task may execute before it is rescheduled. The format of the option is:

```
-w <value>
```

where <value> is the maximum time the task is to execute, in milliseconds, before a task switch occurs between it and a task having the same or higher priority. The default value is 0 (no timeslice). If you choose a value greater than 0, it must be at least as great as the clock interval time.

Since rescheduling of tasks is incompatible with interrupt-scheduling, *-w* is incompatible with -D(elay_NonPreempt (see above).

2.2.9.3. -X(ception_show - Report Unhandled Exceptions. 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.

2.2.9.4. -Y and -y - Alter Stack Size. In the absence of a representation specification for task storage_size, the run time will allocate 8192 bytes of storage for each executing task. You can change the amount of space allocated for tasking by using the *-Y* and *-y* options.

-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 32-bit (long_integer) bytes. The default is 8192.

-y specifies the stack-guard size. The stack-guard space is the amount of additional space allocated per task 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 16-bit (natural) bytes. The value given must be greater than the task-stack size. The default is 1052 bytes; this is the amount allocated unless otherwise specified.

A representation specification for task storage size overrides a value supplied with either option.