ION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 25 Mar 1991 to 01 Jun 1993

1

4. TITLE AND SUBTITLE
TeleSoft, TeleGen2 Ada Cross Development System, Version 3.1, for VAX to 80386, VAX Cluster (2 VAX to 80386 (Host) to Intel SBC 386-120 (80386)(Target), 910325I1.11139

5. FUNDING NUMBERS

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

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

8. PERFORMING ORGANIZATION REPORT NUMBER
IABG-VSR 092

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

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

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

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
TeleSoft, TeleGen2 Ada Cross Development System, Version 3.1, for VAX to 80386, Ottobrunn Germany, VAX Cluster (2 VAX to 80386 (Host) to Intel SBC 386-120 (80386)(Target),ACVC 1.11

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

15. NUMBER OF PAGES

16. PRICE CODE

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

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-03-25.

Compiler Name and Version: **TeleGen2™ Ada Cross Development System, Version 3.1, for VAX to 80386**

Host Computer System: **VAX Cluster with 2 VAX 6210 under VMS 5.3**

Target Computer System: **Intel SBC 386-120 (Intel 80386) with TeleAda-EXEC runtime, Version 1.0**

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

As a result of this validation effort, Validation Certificate #910325I1.11139 is awarded to TeleSoft. This certificate expires on 01 March 1993.

This report has been reviewed and is approved.

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

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

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

AVF Control Number: IABG-VSR 092
25 March 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910325I1.11139
TeleSoft
TeleGen2™ Ada Cross Development System
Version 3.1, for VAX to 80386
VAX Cluster (2 VAX 6210) =>
Intel SBC 386-120 (80386)

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

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

Declaration of Conformance

Customer: Telesoft
Ada Validation Facility: IABG
ACVC Version: 1.11

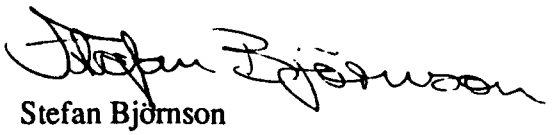
Ada Implementation

Ada Compiler Name: TeleGen2™ Ada Cross Development System,
Version 3.1 for VAX/VMS to 386
Host Computer System: VAX 6210 under VMS 5.3
Target Computer System: Intel SBC 386-120 (80386/387) board
with TeleAda-EXEC runtime

Customer'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, ISO 8652-1987, in the implementation listed in this declaration.

Date: March 22, 1991



Stefan Björnson

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.

CHAPTER 1

INTRODUCTION

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

1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

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

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

1.2 REFERENCES

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

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

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

1.3 ACVC TEST CLASSES

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

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

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

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

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

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

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

1.4 DEFINITION OF TERMS

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

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 91-03-14.

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

2.2 INAPPLICABLE TESTS

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

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

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

The following 21 tests check for the predefined type SHORT_INTEGER:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

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

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

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

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

C46013B, C46031B, C46033B, and C46034B contain 'SMALL representation clauses which are not powers of an integer.

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

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

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

LA3004B, EA3004D, and CA3004F check for pragma INLINE for functions.

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

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

AE2101C and EE2201D..E (2 tests) use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

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.

The following 264 tests check for sequential, text, and direct access files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CF3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 12 tests.

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

B71001Q	BA3006A	BA3006B	BA3007B	BA3008A
BA3008B	BA3013A			

CA2009C, CA2009F, BC3204C, and BC3205D were graded inapplicable by Evaluation Modification as directed by the AVO. Because the implementation

makes the units with instantiations obsolete (see section 2.2), the Class C tests were rejected at link time and the Class B tests were compiled without error.

CE3901A was graded passed by Test Modification as directed by the AVO. This test expects that the checks at line 60 & 70 cannot be meaningfully made for implementations that do not support external files, since these checks depend on the creation of an external file. This implementation does support external access to the console, and so the Report.Legal_Filename call within the call to create an external file at line 52 was replaced by the string "CONSOLE:".

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

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

TeleSoft Europe
Bryggargatan 6
P.O. Box 1001
S-14901 Nynäshamn
Sweden

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

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

a)	Total Number of Applicable Tests	3546	
b)	Total Number of Withdrawn Tests	93	
c)	Processed Inapplicable Tests	66	
d)	Non-Processed I/O Tests	264	
e)	Non-Processed Floating-Point Precision Tests	201	
f)	Total Number of Inapplicable Tests	531	(c+d+e)
g)	Total Number of Tests for ACVC 1.11	4170	(a+b+f)

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

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 531 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 and 264 executable tests that use file operations not supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

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

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

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by a serial interface, and run. The results were captured on the host computer system.

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

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test are given on the next page, which was supplied by the customer.

Compiler Option Information

B-TESTS:

```
TSADA/E386/ADA/MONITOR/LIBFILE=<BIN_NAME>/VIRTUAL_SPACE=3000/OPTIMIZE/LIST
<TEST_NAME>
```

Legend:

TSADA/E386/ADA	invoke Ada compiler
LIST	generate source listing
MONITOR	verbose
OPTIMIZE	optimize with all options
LIBFILE=<BIN_NAME>	each bin gets its own library list allowing multiple compiles in the same directory and setting up a new library for each compilation
<TEST_NAME>	name of Ada source file to be compiled
VIRTUAL_SPACE=3000	set virtual space of the library manager greater than default

EXECUTABLE TESTS:

```
TSADA/E386/ADA/MONITOR/LIBFILE=<BIN_NAME>/VIRTUAL_SPACE=3000/OPTIMIZE
/LIST/BIND=<MAIN_NAME> <TEST_NAME>
```

Legend:

TSADA/E386/ADA	invoke Ada compiler
LIST	generate source listing
MONITOR	verbose
OPTIMIZE	optimize with all options
LIBFILE=<BIN_NAME>	each bin gets its own library list allowing multiple compiles in the same directory and setting up a new library for each compilation
<TEST_NAME>	name of Ada source file to be compiled
VIRTUAL_SPACE=3000	set virtual space of the library manager greater than default
BIND=<MAIN_NAME>	generate a bind with the the main program of <MAIN_NAME>

```
TSADA/E386/LINK/LIBFILE=<BIN_NAME>/OPT=<OPTIONS_FILE>
/LOAD_MODULE <MAIN_NAME>
```

Legend:

TSADA/E386/LINK	invoke Ada linker
LIBFILE=<BIN_NAME>	each bin gets its own library list allowing multiple compiles in the same directory and setting up a new library for each compilation
OPT=<OPTIONS_FILE>	options file appropriate for target configuration
LOAD_MODULE	output executable in Telesoft execute form suitable for downloading to enhance speed

APPENDIX A

MACRO PARAMETERS

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

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

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

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_646
$DEFAULT_MEM_SIZE	2147483647
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	Ada386
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	ENT_ADDRESS
$ENTRY_ADDRESS1	ENT_ADDRESS1
$ENTRY_ADDRESS2	ENT_ADDRESS2
$FIELD_LAST	1000
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	4.25354E+37
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.80141E+38
$HIGH_PRIORITY	63

```
$ILLEGAL_EXTERNAL_FILE_NAME1
                        BADCHAR*^/%

$ILLEGAL_EXTERNAL_FILE_NAME2
                        /NONAME/DIRECTORY

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.ADA")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006E1.ADA")

$INTEGER_FIRST          -32768

$INTEGER_LAST           32767

$INTEGER_LAST_PLUS_1    32768

$INTERFACE_LANGUAGE     ASSEMBLY

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -131_073.0

$LINE_TERMINATOR        ' '

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        I386_CODE'(OP => NOP);

$MACHINE_CODE_TYPE      OPCODES

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2_147_483_648

$MIN_INT                -2147483648

$NAME                   NO_SUCH_TYPE_AVAILABLE

$NAME_LIST              TELEGEN2

$NAME_SPECIFICATION1    ACVC:[TEMP]X2120A.;1

$NAME_SPECIFICATION2    ACVC:[TEMP]X2120B.;1
```

$NAME_SPECIFICATION3	ACVC:[TEMP]X3119A.;1
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483647
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	TELEGEN2
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	RECORD OP : OPCODES; END RECORD;
$RECORD_NAME	I386_CODE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	1024
$TICK	0.01
$VARIABLE_ADDRESS	VAR_ADDRESS
$VARIABLE_ADDRESS1	VAR_ADDRESS1
$VARIABLE_ADDRESS2	VAR_ADDRESS2

APPENDIX B

COMPILATION SYSTEM AND LINKER OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

Table 3-1. Compiler Command Qualifiers.

Qualifier Name	Action	Default
/ABORT_COUNT=<value>	Specify maximum errors/warnings.	999
/[NO]BIND [=<main_unit>]	/BIND runs the Binder on unit being compiled or on unit specified.	/NOBIND
/CONTEXT =<value>	Request <value> context lines on each side of an error in error listing.	1
/[NO]DEBUG	/DEBUG enables generation of information for the Source Level Debugger.	/NODEBUG
/INPUT_LIST	Input file contains names of files to be compiled, not Ada source.	File contains Ada source.
/LIBFILE=<file_spec>	Specify name of library file.	LIBLST.ALB
/[NO]LIST [=<file_spec>]	/LIST creates listing file. Default name is <source_file_name>.LIS, or <file_spec>.LIS, if specified.	/NOLIST
/[NO]MACHINE_CODE [=<file_spec>]	/MACHINE_CODE requests an ASM-386 listing, which is sent to <comp_unit>.AS3 or to <file_spec>, if specified.	/NOMACHINE_CODE
/[NO]MONITOR	/MONITOR requests progress messages.	/NOMONITOR
/[NO]OBJECT	/NOOBJECT restricts compilation to syntactic and semantic analysis.	/OBJECT
/[NO]OFFSET	/OFFSET includes hex offsets in an assembly listing.	/NOOFFSET
/[NO]OPTIMIZE [=(<option>{,<option>})] [<qualifier>]	/OPTIMIZE causes Optimizer to be run on unit(s) being compiled.	/NOOPTIMIZE
/[NO]PROFILE	/PROFILE causes execution profile code to be output in generated object code.	/NOPROFILE
/[NO]SQUEEZE	/SQUEEZE deletes unneeded intermediate unit information after compilation.	/SQUEEZE (/NOSQUEEZE if /DEBUG or /NOOBJECT)
/[NO]SUPPRESS [=<option>{,<option>}]	/SUPPRESS suppresses selected run-time checks and/or source line references in generated object code.	/NOSUPPRESS
/TEMPLIB =(<sublib>{,<sublib>})	Specify a temporary library containing listed sublibraries.	None.
/[NO]UPDATE	/UPDATE updates the working sublibrary after each successful compilation.	/UPDATE

3.4.2.2. Using a List File: /INPUT_LIST. The /INPUT_LIST qualifier instructs the compiler that the command line parameter is a list file specification and that the file contains a list of source file specifications for units to be compiled. The list file must contain one source file specification per line. For example, to compile source files CALC_ARITH.ADA and CALC_MEM.ADA in the current default directory and file CALC_IO.ADA in directory [CIOSRC], first prepare a file CALC_COMPILE.LIS containing the following text:

APPENDIX C

APPENDIX F OF THE Ada STANDARD

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

Attachment F: Package Standard Information

For the VAX/E386 Ada compiler running on a VAX Server under VMS 5.3 Operating System, the numeric types and their properties are as follows:

Integer types:

INTEGER
size = 16
first = -32768
last = +32767

LONG_INTEGER
size = 32
first = -2147483648
last = +2147483647

Floating-point types:

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

LONG_FLOAT
size = 64
digits = 15
'first = -1.79769E+308
'last = 1.79769E+308
machine_radix = 2
machine_mantissa = 53
machine_emin = -1021
machine_emax = +1024

Fixed-point types:

DURATION
size = 32
delta = 2#1.0#e-14
first = -86400
last = +86400

Table C-3. Attributes of Type Duration.

Attribute	Value
'Delta	2 ** (-14)
'First	-86400
'Last	86400

[LRM 9.8] Priorities

The compiler provides sixty-four levels of priority to associate with tasks using pragma Priority. Package System specifies the predefined subtype Priority as:

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

The default priority assigned to tasks without a pragma Priority specification is (Priority'First + Priority'Last) / 2 (i.e., 31).

[LRM 9.11] Shared Variables

The restrictions on shared variables are only those specified in the LRM.

C.5. LRM Chapter 10 - Program Structure and Compilation Issues

[LRM 10.1] Compilation Units - Library Units

All main programs are required to be either parameterless procedures or parameterless functions that return an integer result type.

C.6. LRM Chapter 11 - Exceptions

[LRM 11.1] Exception Declarations

The compiler raises Numeric_Error for integer or floating point overflow and for division-by-zero situations. Floating point underflow yields a result of zero without raising an exception.

The compiler raises Program_Error and Storage_Error in those situations that are specified in LRM Section 11.1.

C.7. LRM Chapter 13 - Representation Clauses and Implementation-Dependent Features"

The compiler supports most LRM Chapter 13 facilities. The sections below document those LRM Chapter 13 facilities that are *not* implemented or that have implementation restrictions. Facilities implemented exactly as described in the LRM are not mentioned.

[LRM 13.1] Representation Clauses

Records that are packed using pragma Pack follow these conventions:

1. The allocated size of each component is always a power of 2 (1, 2, 4, ...).
2. Components of records can cross word boundaries.
3. Components that are composite types (arrays and records) are always allocated on a System.Storage_Unit (8-bit or word) boundary.

Refer to Section 6.8 for examples that illustrate these points.

[LRM 13.2] Length Clauses

A length clause specifies an amount of storage associated with a type.

(a) Size specification: 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 discrete and fixed-point types (integer, enumeration (including character, Boolean)), the minimum specifiable size (Min_Size) is somewhat target-dependent. For ADA386, the 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 V3.23.06, using a size clause for a discrete type may cause inefficient code to be generated. For example, consider an integer type Nibble:

```
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 access types, the simple expression in the size clause must be equal to 16. This includes task types, which are implemented as access types.

For floating-point types, the simple expression must be equal to the required hardware length for the type (32 or 48).

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 (8 for s).

(b) Specification of 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 as follows:

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

For ADA386, this length clause allocates from the heap 64 words 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, 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 ADA386, information needed to manage storage blocks in a collection is stored in a collection header that requires **24 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 **8 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 ten Symbol_Rec objects, you could use an expression such as:

```
for Dict_Ref'Storage_Size use ((Symbol_Rec'Size * 10)+24);
```

where 24 bytes 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 an integer.)

In another application, 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.

(c) Specification of storage for a 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. For ADA386, the stack size is specified by the Linker options constant TASK_STACK_SIZE. In the Linker options file provided, the default stack size is 4000 storage units (bytes).

(d) Specification of 'Small for a fixed point type: 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 two, 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 using explicit error bounds. For example, suppose T is a fixed type defined as follows:

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

Fixed'Small = 0.25, a power of two.
3.0 is a model number (3.0 = 12 * 0.25), but it is not a power of two.

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 the ADA386, the specified value must be (mathematically) equal to either an exact integer or the reciprocal of an exact integer. Below are some examples of legal and illegal values for Small:

```
Legal:
1.0, 2.0, 3.0, 4.0, . . .
0.5, 1.0/3.0, 0.25, 1.0/3600.0

Illegal:
2.5, 2.0/3.0, 0.3
```

The Ada language requires the value specified for 'Small to be no larger than the 'Delta of the type (13.2:12).

[LRM 13.3] Enumeration Representation Clauses

The compiler does not support enumeration representation clauses on Boolean types.

[LRM 13.4] Record Representation Clauses

The compiler supports record representation clauses within the following constraints:

1. You must specify each component of the record with a component clause.
2. The alignment of the record is restricted to mod 2, word aligned.
3. The ordering of bits within a byte is right to left, where the high-order bit is at the left.
4. Components can cross word boundaries.
5. Any object of a discrete type of size larger than 8 bits requires a sign bit. In the example below, the type Actually_11_bits appears to be representable in 10 bits:

```
type Actually_11_Bits is new Integer range 0..2**10-1;

type Small_Rec is record
  Isit_10_Bits: Actually_11_Bits;
end record;

for Small_Rec use record at mod 2;
  Isit_10_Bits : at 0 range 0..9;  -- error!  Invalid size.
end record;
```

   Since actually 11 bits are used because of the sign bit, the component clause in the example is illegal.

There are no implementation-dependent names to denote implementation-dependent components.

[LRM 13.5] Address Clauses

The compiler does not support address clauses for packages or tasks.

For address clauses applied to objects, the compiler interprets a simple expression of type Address as a position within the linear address space of the 80386. You must use Unchecked_Conversion to the private type System.Address to specify address constants.

[LRM 13.5.1] Interrupts

For interrupt entries, you can give the address of an *interrupt descriptor*. Refer to Section 6.12.1 for more information.

[LRM 13.6] Change of Representation

The compiler does not support changes of representation for types with record representation clauses.

[LRM 13.7] The Package System

The compiler does not support pragmas System_Name, Storage_Unit, and Memory_Size.

[LRM 13.7.2] Representation Attributes

The compiler does not support 'Address for packages or labels.

[LRM 13.7.3] Representation Attributes of Real Types

The representation attributes for the predefined floating point types Float and Long_Float are shown in Table C-2.

[LRM 13.9] Interfaces to Other Languages

The compiler supports pragma Interface to subprograms in Intel languages ASM386, PL/M, FORTRAN-386, and C-386 if Intel Assemblers and compilers process these routines into Intel OMF-386 format. The Ada-386 Object Module Importer can convert an OMF-386 module into Object Form acceptable to the Ada Linker, and the Linker can link the routine with a compiled Ada-386 program.

[LRM 13.10.2] Unchecked Type Conversions

The compiler allows unchecked conversions between types (or subtypes) T1 and T2 provided that:

1. They have the same static size.
2. The destination is not an unconstrained record or array type.
3. They are not private, unless they are subtypes of or are derived from System.Address.
4. They are not types with discriminants.

Note that the size used in the unchecked conversion is the 'Size of the target, which may not be the same as the static size of the target.

C.8. LRM Appendix A - Predefined Language Attributes

The compiler implements the predefined attribute 'Address as described in Section 6.11. The attribute is not supported for packages or labels.

C.9. LRM Appendix F - Implementation-Dependent Characteristics

The Ada language definition allows for certain target dependencies. These dependencies must be described in the reference manual for each implementation, in an "Appendix F" that addresses each point listed in LRM Appendix F. The sections below constitute Appendix F for this implementation.

C.9.1. Implementation-Defined Pragmas. The compiler has four implementation-defined pragmas: pragma Comment, pragma Linkname, pragma Interrupt, and pragma Images.

Pragma Comment

Use pragma Comment to embed a comment into the object code using the syntax:

pragma Comment(<string_literal>);

where **<string_literal>** represents the characters to be embedded in the object code. You can put pragma Comment at any location within the source code of a compilation unit, except within the generic formal part of a generic unit. You can enter any number of comments into the object code using pragma Comment.

Pragma Linkname

Use pragma Linkname in association with pragma Interface to provide access to a routine whose name can be specified by an Ada string literal. Pragma Linkname takes two arguments, the name of the subprogram specified in a pragma Interface and a string literal specifying the exact link name that the code generator is to use in emitting calls to the associated subprogram. In the example below, pragma Linkname associates the linkname "DM@ACCESS" with the procedure Dummy_Access used in a pragma Interface specification:

```
procedure Dummy_Access(Dummy_Arg: System.Address);
pragma Interface(Intel, Dummy_Access);
pragma Linkname(Dummy_Access, "DM@ACCESS");
```

A pragma Linkname specification must immediately follow the pragma Interface for the associated subprogram or else a warning is issued that the pragma Linkname has no effect.

Pragma Interrupt

Pragma Interrupt is used for function-mapped optimizations of interrupts as described in Section 6.12.1.6.2. The syntax of this pragma is:

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

Pragma Interrupt has the effect that entry calls to the associated entry, on behalf of an interrupt, are made with a reduced call overhead.

Pragma Images

Pragma Images controls the creation and allocation of the image table for a specified enumeration type. The syntax of this pragma is:

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

The default for Ada-386 is Deferred, which saves space in the literal pool by not creating an image table for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If you use one of these attributes, an image table is generated in the literal pool of the compilation unit in which the attribute appears. If you use the attributes in more than one compilation unit, more than one image table is generated, eliminating the benefits of deferring the table. The image table is a string literal whose length is:

(sum of lengths of literals) + (3 * (number of literals + 1))

For a very large enumeration type, the length of the image table might be greater than Integer'Last and too large to fit into a string. For the 80386 target with a 16-bit integer, the upper bound on the length of the image table is 32767. Using the default Deferred value for all enumeration types allows you to declare very large enumeration types. The attempt to use 'Image, 'Value, or 'Width for such a type, however, might cause the compiler to attempt to create an image table that exceeds the upper bound, resulting in an error message.

C.9.2. Implementation-Dependent Attributes. The compiler defines two implementation-dependent attributes, 'Address and 'Offset, which facilitate machine code insertions by allowing you to access Ada objects with little data movement or setup. These attributes and their usage are described in detail in Section 6.11. 'Address is not supported for packages or labels.

C.9.3. Package System. The specification of package System is provided below. Note that the named number Tick is not used by any component of the compiler or run-time system. Similarly, Memory_Size is not used.

```
package System is

   type Address is access integer;

   type Name is (Ada386);

   System_Name  : constant Name := Ada386;

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

       -- System-Dependent Named Numbers:

   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;

       -- Other Declarations:

   type Subprogram_Value is private;

private
   -- ...

end System;
```

C.9.4. Representation Clauses. Restrictions on representation clauses are discussed in the LRM 13.1 paragraph of Section C.7.

C.9.5. Implementation-Generated Names. There are no implementation-generated names to denote implementation-dependent components.

C.9.6. 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.

C.9.7. Unchecked Conversion Restrictions. Restrictions on unchecked conversions are discussed in the LRM 13.10.2 paragraph of Section C.7.

C.9.8. Implementation-Dependent Characteristics of the I/O Packages.

1. Only I/O to a system console is supported in Text_IO.

2. In Text_IO, the type Count is defined as follows:

```
type Count is range 0..2_147_483_646
```

3. In Text_IO, the type Field is defined as follows:

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

4. The standard library contains preinstantiated versions of Text_IO.Integer_IO for type Integer and Long_Integer and of Text_IO.Float_IO for type Float and Long_Float. Use the following to eliminate multiple instantiations of these packages:

   Integer_Text_IO
   Long_Integer_Text_IO
   Float_Text_IO
   Long_Float_Text_IO

5. According to the latest interpretation of the LRM, during a Text_IO.Get_Line call, if the buffer passed in has been filled, the call is completed and any succeeding characters and/or terminators (e.g., line, page, or end-of-file) are not read. The first Get_Line call reads the line up to but not including the end-of-line mark, and the second Get_Line reads and skips the end-of-line mark left by the first read.