REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

AD-A233 775

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED
Final: 4 Feb 1991 to 01 Mar 1993

4. TITLE AND SUBTITLE
Ada Compiler Validation Summary Report: CONVEX Computer Corporation
CONVEX Ada Version 2.0, CONVEX C220 (Host & Target), 900910W1.11027

5 FUNDING NUMBERS

6. AUTHOR(S)
Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB
Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-387.0291

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)*
CONVEX Computer Corporation, CONVEX Ada Version 2.0, Wright-Patterson AFB, OH, CONVEX C220 ConvexOS 8.1 (Host & 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

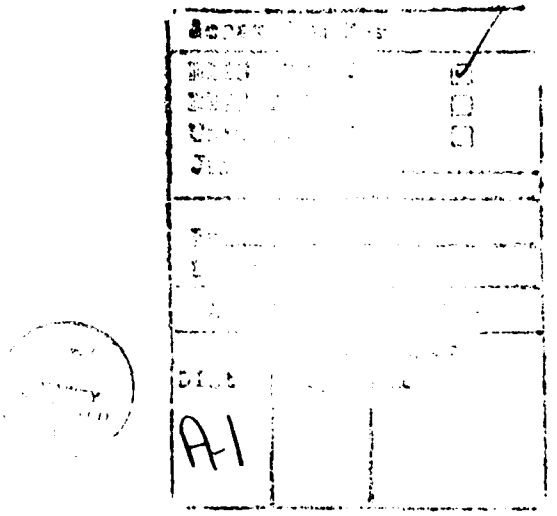
16. PRICE CODE

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

AVF Control Number: AVF-VSR-387.0291
4 February 1991
90-04-20-CVX

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 900910W1.11027
CONVEX Computer Corporation
CONVEX Ada Version 2.0
CONVEX C220 => CONVEX C220

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503



Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 10 September 1990.

Compiler Name and Version: CONVEX Ada Version 2.0

Host Computer System: CONVEX C220 ConvexOS 8.1

Target Computer System: CONVEX C220 ConvexOS 8.1

Customer Agreement Number: 90-04-20-CVX

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

As a result of this validation effort, Validation Certificate 900910W1.11027 is awarded to CONVEX Computer Corporation. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

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

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

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: CONVEX Computer Corporation

Certificate Awardee: CONVEX Computer Corporation

Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503

ACVC Version: 1.11

Ada Compiler Name and Version: CONVEX Ada Version 2.0

Host Computer System: Convex C220 ConvexOS 8.1

Target Computer System: CONVEX C220 ConvexOS 8.1

Declaration:

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

Frank J. Maulucci
Customer Signature

9/10/90
Date

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC) 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 2 September 1990.

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

IMPLEMENTATION DEPENDENCIES

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

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

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35702B, C35713C, B86001U, and C86006G check for the predefined type LONG_FLOAT.

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

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

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

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

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

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

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

CE2102H checks for CREATE with modes IN_FILE and OUT_FILE for direct access file.

CE2102R..W (6 tests) check for CREATE with modes OUT_FILE and INOUT_FILE for direct access files.

CE2102S..V (4 tests) check for CREATE with mode INOUT_FILE for direct access files.

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

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

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

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

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

CE3413A checks for OPEN with mode IN_FILE for text files.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 13 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:

B24009A	B33301B	B38003A	B38003B	B38009A	B38009B
B85008G	B85008H	BC1303F	BC3005B	BC2B03A	BD2D03A
BD4003A					

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 technical information about this Ada implementation system, see:

Larry Grossman
3000 Waterview Parkway
P.O. Box 833851
Richardson TX 75083-3851

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

Larry Grossman
3000 Waterview Parkway
P.O. Box 833851
Richardson TX 75083-3851

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].

PROCESSING INFORMATION

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	3817
b) Total Number of Withdrawn Tests	74
c) Processed Inapplicable Tests	78
d) Non-Processed I/O Tests	0
e) Non-Processed Floating-Point Precision Tests	201
f) Total Number of Inapplicable Tests	279
g) Total Number of Tests for ACVC 1.11	4170

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

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 279 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

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

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

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

OPTION	EFFECT
-nw	Suppress warning messages.
-M	Upon successful compilation, perform link phase and build an executable.

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

APPENDIX A

MACRO PARAMETERS

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

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

MACRO PARAMETERS

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

Macro Parameter	Macro Value
$MAX_IN_LEN	499
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	16777216
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	CONVEX_UNIX
$DELTA_DOC	0.000000000465661287307739257812 5
$ENTRY_ADDRESS	16#1E#
$ENTRY_ADDRESS1	16#1F#
$ENTRY_ADDRESS2	16#1D#
$FIELD_LAST	2147483647
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	:CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100000.0
$GREATER_THAN_DURATION_BASE_LAST	10000000.0
$GREATER_THAN_FLOAT_BASE_LAST	1.0E+308
$GREATER_THAN_FLOAT_SAFE_LARGE	8.98846567431157E+307

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        1.70141E+38

$HIGH_PRIORITY          99

$ILLEGAL_EXTERNAL_FILE_NAME1
              /no/such/directory/ILLEGAL_EXTERNAL_FILE_NAME1

$ILLEGAL_EXTERNAL_FILE_NAME2
(1..41=>'THIS-FILE-NAME-IS-TOO-LONG-FOR-MY-SYSTEM-', 42..256=>'d')

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006E1.TST")

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     FORTRAN

$LESS_THAN_DURATION     -75_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -10_000_073.0

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        code_2'(and_op, s0, s0);

$MACHINE_CODE_TYPE      opcode

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2_147_483_648

$MIN_INT                -2147483648
```

MACRO PARAMETERS

$NAME	TINY_INTEGER
$NAME_LIST	CONVEX_UNIX
$NAME_SPECIFICATION1	/texec/acvc1.11/ctests/ce/X2120A
$NAME_SPECIFICATION2	/texec/acvc1.11/ctests/ce/X2120B
$NAME_SPECIFICATION3	/texec/acvc1.11/ctests/ce/C3119A
$NEG_BASED_INT	16#FFFFFFFDF#
$NEW_MEM_SIZE	16777216
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	CONVEX_UNIX
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	RECORD NULL; END RECORD;
$RECORD_NAME	CODE_0
$TASK_SIZE	32
$TASK_STORAGE_SIZE	10000
$TICK	0.0001
$VARIABLE_ADDRESS	16#FFFF0000#
$VARIABLE_ADDRESS1	16#FFFF0004#
$VARIABLE_ADDRESS2	16#FFFF0008#

$YOUR_PRAGMA	dynamic_select
	external_name
	force_parallel
	force_vector
	global_opt_unit
	implicit_code
	inline_only
	interface_name
	link_with
	local_opt_unit
	max_trips
	no_image
	no_parallel
	no_recurrence
	no_side_effects
	no_vector
	parallel_unit
	passive
	prefer_parallel
	prefer_vector
	pstrip
	scalar
	scalar_unit
	share_body
	share_code
	spread_task
	synch_parallel
	unroll
	vector_unit
	volatile
	vstrip

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler 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.

NAME

ada - Ada compiler

SYNTAX

ada [options] [ada_source.a]... [linker_options]

[object_file.o]...

DESCRIPTION

The command ada executes the Ada compiler and compiles the named Ada source (.a suffix) files. The files must reside in an Ada program library. The ada.lib file in this library is modified after each Ada unit is compiled.

The object for each compiled Ada unit is placed in a file with the same name as that of the source file with 01, 02, etc. substitute for .a. The -o option can be used to produce an executable with a name other than a.out, the default.

By default, ada produces only object and net files. If the -M option is used, the compiler automatically invokes a.ld and builds a complete program with the named library unit as the main program.

Object files (.o files produced by a previous compilation) may be given as arguments to ada. There files are passed on to the linker and linked with Ada object files produced by the current compilation.

Command line options may be specified in any order, but the order of compilation and the order of the files to be passed to the linker can be significant.

OPTIONS

-a file name
: Treat file name as an ar file. Since archive files end with .a, -a is used to distinguish archive files from Ada source files.

-d
: Analyze for dependencies only. Do not perform semantic analysis or code generation. Update the library, marking any defined units as uncompiled.

The -d option is used by a.make to establish dependencies among new files.

-ds Enable dynamic_selection of loops.

-e Process compilation error messages using a.error and direct it to stdout. Only the source lines containing errors are listed. The options -e and -E are mutually exclusive.

-E

-E file

-E directory
Without a file or directory argument, process error messages using a.error and direct the output to stdout; the raw error messages are left in ada source.err. If a file pathname is given, the raw error messages are placed in that file. If a directory argument is supplied, the raw error output is placed in dir/source.err. The options -e and -E are mutually exclusive.

-el Intersperse error messages among source lines and direct to stdout.

-El

-El file

-El directory
Same as the -E option except that a source listing with errors is produced.

-ep number of processors
Specify the number of expected processors the program will have available during execution. The default is the number of CPU's installed on the system where the compile is performed.

-ev Process syntax error messages using a.error, embed them in the source file, and invoke the environment editor ERROR_EDITOR. If ERROR_EDITOR is defined, the environment variable ERROR_PATTERN is an editor search command that locates the first occurrence of ### in the error file. If no editor is specified, invoked vi.

-l file abbreviation
Link this library file. (No space between the -l and the file abbreviation.) See also operating system documentation. ld(1)

-M unit name
Produce an executable program using the named unit as the main program. The unit must be either a parameterless procedure or a parameterless function returning an integer. The executable program will be written to the file a.out unless overridden with the -o option.

-M ada source.a
The same as -M unit name, except that the unit name is assumed to be the root name of the .a file (for foo.a the unit is foo). Only one .a file may be preceded by -M.

-no Select no optimization.

-nw Suppress warning diagnostics.

-o executable file
This option is to be used in conjunction with the -M option. executable file is to be the name of the executable rather than the default a.out.

-or Optimization report option
The -or option accepts one of none, all, loop, array, or nest. The parameter indicates the level of optimization summary desired. The default is loop.

-O[0-3] Select default program optimization: -O0 means basic block optimization on ly, -O1 means global optimization, -O2 means vectorization, -O3 means parallelization.

-pa Enable generation of instrumentation points for the Convex Performance Analyzer CXpa. Routine entry/exits and loops can be analyzed. This also causes a.ld to link in the monitor routines which are needed by CXpa if the -M option is used.

-pab Enable generation of instrumentation points for the Convex Performance Analyzer CXpa. Basic blocks can be analyzed. This also causes a.ld to link in the monitor routines which are needed by CXpa if the -M option is used.

-par Enable generation of instrumentation points for the Convex Performance Analyzer CXpa. Routine entry/exits can be analyzed. This also causes a.ld to link in the monitor routines which are needed by CXpa.

-re Parallel reentrant compilation flag. Useful when making calls from within a parallelized loop to a routine which has also been parallelized.

-rl Enable automatic loop unrolling and dynamic_selection of loops.

-R ada program library
Recompile all generic instantiations that are out-of-date in the specified Ada program library. The default is the current library if the ada program library is omitted

-S Apply pragma SUPPRESS to the entire compilation for all suppressible checks.

-tm hardware type
Specify the target machine instruction set. The parameter hardware type can be either C1, c1, C2, or c2. C1, c1 selects the Convex C100 series instruction set as the target machine. C2, c2 selects the Convex C200 series instruction set as the target machine. The default is the host machine type.

LINKER OPTIONS

The 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 linker documentation and not to this report.

-uo Enable unsafe optimizations.

-ur Enable automatic loop unrolling.

-v Print compiler version number, date and time of compilation, name of file compiled, command input line, total compilation time, and error summary line.

-V Same as -v.

SEE ALSO

CONVEX Ada User's Guide and operating system documentation, ld(1).

DIAGNOSTICS

The diagnostics produced by the CONVEX Ada compiler are intended to be self-explanatory. Most refer to the reference manual (RM) for the Ada programming language. Each RM reference includes a section number and, optionally, a paragraph number enclosed in parentheses.

NAME

a.ld - prelinker

SYNTAX

a.ld [options] unit_name [ld_options]

DESCRIPTION

a.ld collects the object files needed to make unit name a main program and calls the UNIX linker ld(1) to link the necessary Ada and other objects required to produce an executable module in a.out. unit name is the main program and must be a non-generic subprogram. If unit name is a function, it must return a value of the type STANDARD.INTEGER. This integer result is passed back to the UNIX shell as the status code of the execution. The utility uses the net files produced by the Ada compiler to check dependency information. a.ld produces an exception mapping table and a unit elaboration table and passes this information to the linker. a.ld processes directives

for generating executables from the ada.lib file in the Ada program libraries on the search list. Besides information generated by the compiler, these directives also include WITHn directives that allow the automatic linking of object modules compiled from other languages or Ada object modules not named in context clauses in the Ada source. Any number of WITH directives may be placed into a library, but they may be numbered contiguously beginning at WITH1. The directives are recorded in the library ada.lib file and have the following form:

WITH1:LINK:object file:

WITH:LINK:archive file:

WITH directives may be placed in the local Ada libraries or in any library on the search list. A WITH directive in the current Ada program library or earlier on the library search list will hide the same numbered WITH directive in a library later in the library search list.

Use the utility a.info to change or report library directives in the current library. All arguments after unit name are passed on to the linker. These may be linker options, archive libraries, library abbreviations, or object files.

OPTIONS

-E unit_name
Elaborate unit name as early in the elaboration order as possible.

-F print a list of dependent files in order and suppress linking.

-o executable file
Use the specified file name as the name of the output rather than the default, a.out.

-pa, -par, or -pab
Specifies that support object code for use in the Convex Performance Analyzer CXpa is to be included in the load. Programs which are to be analyzed using CXpa must be compiled with either the -pa or -pab option.

-tm hardware type
Specify the target machine instruction set. The parameter hardware type can be either C1, c1, C2, or c2. C1, c1 selects the Convex C100 series instruction set as the target machine. C2, c2 selects the Convex C200 series instruction set as the target machine. The default is the host machine type.

-U Print a list of dependent units in order and suppress linking.

-v Print the linker command before executing it.

-V Print the linker command, but suppress execution.

FILES

ada location/standard/ (Startup and standard library routines) objects/ (Ada object files) a.out (default output file)

SEE ALSO

CONVEX Ada User's Guide, UNIX ld(1)

DIAGNOSTICS

Self-explanatory diagnostics are produced for missing files, etc. Occasional additional messages are produced by the linker.

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:

```
package STANDARD is

    ...

    type INTEGER is range -2147483648 .. 2147483647;

    type FLOAT is digits 15 range
        -8.98846567431157E+307 .. 8.98846567431157E+307;

    type DURATION is delta 1.0E-4 range
        -1.31072E+5 .. 1.31071999938965E+5;

    type SHORT_INTEGER is range -32768 .. 32787;

    type SHORT_FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;

    type TINY_INTEGER is range -128 .. 127;

    ...

end STANDARD;
```

F

Implementation-Dependent Characteristics

This appendix describes the implementation-dependent characteristics of CONVEX Ada as outlined in Appendix F of the *American National Standard Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A-1983). Supplementary information is included as appropriate.

This appendix is organized as follows:

- Implementation-dependent pragmas
- Implementation of predefined pragmas
- Implementation-dependent attributes
- Specification of the package SYSTEM
- Restrictions on representation clauses
- Conventions for implementation-generated names
- Interpretation of expressions in address clauses
- Restrictions on unchecked conversions
- Restrictions on unchecked deallocations
- Implementation characteristics of I/O packages

F.1 Implementation-Dependent Pragmas

CONVEX Ada supports the following implementation-dependent pragmas:

DYNAMIC_SELECT(*param1*,*param2*,*param3*)

This pragma causes the compiler to generate multiple versions of the loop that immediately follows based on trip counts supplied by the user. Up to four versions of the loop can be generated: scalar, vector, parallel, and parallel-outer vector-inner. The compiler also generates code to allow runtime selection of which version to execute.

The DYNAMIC_SELECT pragma accepts three parameters, which specify the trip (iteration) count at which the compiler is to select vector, parallel, or parallel-vector execution. Each parameter may be an integer, the keyword DEFAULT, or the keyword NONE. The compiler selects a version of the loop to execute based on the following rules:

- If the actual trip count is less than the minimum trip count specified in the pragma, the loop runs scalar.
- If the actual trip count is greater than the maximum trip count specified in the pragma, the loop runs in the mode corresponding to the maximum specified trip count.
- In all other cases, the loop runs in the mode corresponding to the largest trip count specified that the actual trip count exceeds.

If you omit one or more of the trip counts by entering DEFAULT instead of an integer, the compiler selects a default trip value for the test. If you use the keyword NONE instead of an integer, the compiler does not generate code for the corresponding mode.

The DYNAMIC_SELECT pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

EXTERNAL_NAME(*name*)

This pragma takes the name of a subprogram or variable defined in Ada and allows the user to specify a different external name that may be used to reference the entity from other languages. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification.

FORCE_PARALLEL

This pragma tells the compiler that the iterations of the following loop are independent and that the loop should be parallelized. If both this pragma and FORCE_VECTOR precede a loop, the loop is vectorized, and the strip-mine loop is parallelized.

The FORCE_PARALLEL pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

FORCE_VECTOR

This pragma tells the compiler that the iterations of the following loop are independent but that the loop should be vectorized rather than parallelized. If both this pragma and FORCE_PARALLEL precede a loop, the loop is vectorized, and the strip-mine loop is parallelized.

The FORCE_VECTOR pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

GLOBAL_OPT_UNIT

This pragma tells the compiler to perform only machine-dependent and scalar optimizations and is equivalent to the *-O1* optimization level.

Only one SCALAR_UNIT, GLOBAL_OPT_UNIT, LOCAL_OPT_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragma may appear in a library unit. Any subsequent SCALAR_UNIT, GLOBAL_OPT_UNIT, LOCAL_OPT_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragmas are flagged as errors and ignored.

IMPLICIT_CODE(ON|OFF)

This pragma takes one of the identifiers ON or OFF as the single argument. This pragma is only allowed within a machine-code procedure. It specifies whether implicit code generated by the compiler should be allowed or disallowed. A warning is issued if OFF is used and any implicit code needs to be generated. The default value is ON.

INLINE_ONLY(*subprogram_name*)

This pragma tells the compiler to inline the subprogram. This pragma also suppresses the generation of a callable version of the routine which saves code space. If a user erroneously makes an INLINE_ONLY subprogram recursive, a warning is issued and a PROGRAM_ERROR is raised at run time.

INTERFACE_NAME(*link_argument*)

This pragma takes the name of a variable defined in another language and allows it to be referenced directly in Ada. The pragma replaces all occurrences of the variable name with an external reference to the second (*link_argument*).

The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar, array, record, access type, procedure, or function. The object may not be any of the following:

- A loop variable
- A constant
- An initialized variable

LINK_WITH(*filename*)

This pragma is used to pass arguments to the target linker. It may appear in any declarative part and accepts one argument, a constant string expression. This argument is passed to the target linker when the unit containing the pragma is include in a link.

For example, the following package will put the *-lm* option on the command line for the linker when MY_PACKAGE is included in the linked program.

```
package MY_PACKAGE is
   pragma LINK_WITH("mylib.a");
end;
```

LOCAL_OPT_UNIT

This pragma tells the compiler to perform only machine-dependent optimizations and is equivalent to the *-O0* optimization level.

Only one SCALAR_UNIT, GLOBAL_OPT_UNIT, LOCAL_OPT_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragma may appear in a library unit. Any subsequent SCALAR_UNIT, GLOBAL_OPT_UNIT, LOCAL_OPT_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragmas are flagged as errors and ignored.

MAX_TRIPS(*n*)

This pragma tells the compiler that the expected number of iterations (trips) for the following loop is approximately *n*. The compiler uses this information for profitability analysis, dynamic selection, and variable strip mining. The pragma accepts a single integer constant argument.

The MAX_TRIPS pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

NO_IMAGE(*enum_type_name*)

This pragma suppresses the generation of the image array used for the IMAGE attribute of enumeration types. This eliminates the overhead required to store the array in the executable image. An attempt to use the IMAGE attribute on a type whose image array has been suppressed will result in a compilation warning and PROGRAM_ERROR raised at run time.

NO_PARALLEL

This pragma prevents parallelization of the following loop but does not prevent vectorization. If both this pragma and the NO_VECTOR pragma precede a loop, the effect is the same as if the SCALAR pragma is used.

The NO_PARALLEL pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

NO_RECURRENCE

This pragma instructs the compiler to vectorize a loop even if the compiler cannot prove that there are no recurrence vector dependencies in the loop. If the loop does, in fact, contain recurrences and the NO_RECURRENCE pragma is specified, incorrect code may be generated.

The NO_RECURRENCE pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

NO_SIDE_EFFECTS

This pragma tells the compiler that the subprogram being compiled does not change the value of any global objects. The optimizer portion of the compiler can then move operations on such variables across calls to the subprogram.

The NO_SIDE_EFFECTS pragma must appear in the declarative part of a function or procedure.

NO_VECTOR

This pragma prevents vectorization of the following loop but does not prevent parallelization. If both this pragma and the NO_PARALLEL pragma precede a loop, the effect is the same as if the SCALAR pragma is used.

The NO_VECTOR pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

PARALLEL_UNIT

This pragma causes the compiler to attempt to generate code that will execute in parallel for the compilation unit in which the pragma occurs. Vectorization may also take place.

Only one SCALAR_UNIT, GLOBAL_OPT_UNIT, LOCAL_OPT_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragma may appear in a library unit. Any subsequent SCALAR_UNIT, GLOBAL_OPT_UNIT, LOCAL_OPT_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragmas are flagged as errors and ignored.

PREFER_PARALLEL

This pragma tells the compiler that, if there is a choice of loops in a nest to parallelize and it is valid to parallelize the loop following the pragma, then it should be chosen for parallelization. All dependencies are honored.

The PREFER_PARALLEL pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

PREFER_VECTOR

This pragma tells the compiler that, if there is a choice of loops in a nest to vectorize and it is valid to vectorize the loop following the pragma, then it should be chosen for vectorization. All dependencies are honored.

The PREFER_VECTOR pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

PSTRIP(*n*)

This pragma tells the compiler that the parallel loop that follows should be strip mined with length *n*. This pragma reduces the overhead required to synchronize CPUs working together on the loop. The pragma increases to *n* the number of iterations each CPU picks up as it gets its next unit of work. The pragma accepts a single integer constant argument.

The PSTRIP pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

SCALAR

This pragma prevents vectorization of the loop that follows. The SCALAR pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

Loops nested within a loop that has the SCALAR pragma applied to it are eligible for vectorization.

SCALAR_UNIT

This pragma may appear any place in the declarative part of a library unit and tells the compiler that no vectorization should be performed on the unit. The SCALAR_UNIT pragma overrides any optimization option specified on the compiler command line.

Only one SCALAR_UNIT, GLOBAL_OPT_UNIT, LOCAL_OPT_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragma may appear in a library unit. Any subsequent SCALAR_UNIT, GLOBAL_OPT_UNIT, LOCAL_OPT_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragmas are flagged as errors and ignored.

SHARE_CODE(*name*,TRUE | FALSE

This pragma takes the name of a generic instantiation or a generic unit as the first argument and one of the identifiers TRUE or FALSE as the second argument. This pragma is only allowed immediately at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

When the first argument is a generic unit, the pragma applies to all instantiations of that generic. When the first argument is the name of a generic instantiation, the pragma applies only to the specified instantiation or overloaded instantiations.

If the second argument is TRUE, the compiler tries to share code generated for a generic instantiation with code generated for other instantiations of the same generic. When the second argument is FALSE, each instantiation gets a unique copy of the generated code. The extent to which code is shared between instantiations depends on this pragma and the generic formal parameters declared for the generic unit.

The SHARE_CODE pragma may also be referenced as SHARE_BODY.

SPREAD_TASK

The SPREAD_TASK pragma may appear any place in the declarative part of a library unit and tells the compiler to use multiple CPUs (rather than one CPU) for tasking at runtime.

SYNCH_PARALLEL

This pragma tells the compiler, at optimization level *-O3, that the loop that follows should be run in parallel even though it requires synchronization that may result in a significant loss of efficiency.*

The SYNCH_PARALLEL pragma must immediately precede the for or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

UNROLL

This pragma tells the compiler to attempt unrolling on the loop immediately following the pragma. Unrolling is performed only if the iteration count is less than 5.

The UNROLL pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

VECTOR_UNIT

This pragma may appear any place in the declarative part of a library unit and tells the compiler that library unit is a candidate for vectorization. Loops within the library unit are analyzed and those that have no recurrence vector dependencies are vectorized. The compiler can be forced to vectorize a loop, even if recurrence dependencies exist, with the NO_RECURRENCE pragma.

The VECTOR_UNIT pragma overrides any optimization option specified on the compiler command line.

Only one SCALAR_UNIT, GLOBAL_OPT_UNIT, LOCAL_OPT_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragma may appear in a library unit. Any subsequent SCALAR_UNIT, GLOBAL_OPT_UNIT, LOCAL_OPT_UNIT, PARALLEL_UNIT, or VECTOR_UNIT pragmas are flagged as errors and ignored.

VOLATILE(*object*)

This pragma guarantees that loads and stores to the named object will be performed as expected after optimization.

VSTRIP(*n*)

This pragma tells the compiler that the vector loop immediately following the pragma should be strip mined with length *n*. This pragma allows the user to reduce strip-mine length, thus creating more iterations of the strip-mine loop so that it can be effectively parallelized. The pragma accepts a single integer constant argument.

The VSTRIP pragma must immediately precede the *for* or *while* loop to which it applies. If the pragma applies to a hand written loop, the pragma must appear immediately before the labeled statement at the head of the hand written loop.

F.2 Implementation of Predefined Pragmas

CONVEX Ada implements the predefined pragmas as follows.

CONTROLLED

This pragma is recognized by the compiler but has no effect.

ELABORATE

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language*.

INLINE

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language*.

INTERFACE

This pragma supports calls to C, FORTRAN, and assembler routines.

The types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS in SYSTEM. All parameters must have mode IN. Record and array objects can be passed by reference using the ADDRESS attribute.

LIST

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language*.

MEMORY_SIZE

This pragma is recognized by the compiler but has no effect. The implementation does not allow SYSTEM to be modified by pragmas; the SYSTEM package must be recompiled.

OPTIMIZE

This pragma is recognized by the compiler but has no effect.

PACK

This pragma causes the compiler to choose a nonaligned representation for composite types but does not cause objects to be packed at the bit level.

PAGE

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language*.

PRIORITY

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language*.

SHARED

This pragma is recognized by the compiler but has no effect.

STORAGE_UNIT

This pragma is recognized by the compiler but has no effect. The implementation does not allow SYSTEM to be modified by pragmas; the SYSTEM package must be recompiled.

SUPPRESS

This pragma is implemented as described in Appendix B of the *American National Standard Reference Manual for the Ada Programming Language* except that RANGE_CHECK and DIVISION_CHECK cannot be suppressed.

SYSTEM_NAME

This pragma is recognized by the compiler but has no effect. The implementation does not allow SYSTEM to be modified by pragmas; the SYSTEM package must be recompiled.

F.3 Implementation-Dependent Attributes

CONVEX Ada provides the implementation-dependent attribute P'REF, where P can represent an object, a program unit, a label, or an entry.

This attribute denotes the effective address of the first of the storage units allocated to P. For a subprogram, package, task unit, or label, it refers to the address of the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, it refers to the corresponding hardware interrupt.

This attribute is of type OPERAND as defined in the package MACHINE_CODE and is only allowed within a machine-code procedure. This attribute is not supported for a package, task unit, or entry.

F.4 Specification of the Package SYSTEM

The specification of the package SYSTEM follows. This specification is available online in the file *system.a* in the *standard* library.

```
-- $CHeader: system.a 0.9 90/03/22 15:57:55 $
-- Copyright 1986 Convex Computer Corp
--
with UNSIGNED_TYPES;
package SYSTEM
is
    pragma SUPPRESS(ALL_CHECKS);
    pragma SUPPRESS(EXCEPTION_TABLES);
    pragma NOT_ELABORATED;

    type NAME is ( convex_unix );
    SYSTEM_NAME     : constant NAME := convex_unix;

    STORAGE_UNIT    : constant := 8;
    MEMORY_SIZE     : constant := 16_777_216;

    -- System-Dependent Named Numbers

    MIN_INT         : constant := -2_147_483_648;
    MAX_INT         : constant := 2_147_483_647;
    MAX_DIGITS      : constant := 15;
    MAX_MANTISSA    : constant := 31;
    FINE_DELTA      : constant := 2.0**(-31);
    TICK            : constant := 0.0001;

    -- Other System-dependent Declarations

    subtype PRIORITY is INTEGER range 0 .. 99;

    MAX_REC_SIZE : integer := 64*1024;

    type ADDRESS is private;

    function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
    function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
    function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
    function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
```

```
        function ADDR_DIFF(A, B: ADDRESS) return INTEGER;
        function INCR_ADDR(A: ADDRESS; INCR: INTEGER) return ADDRESS;
        function DECR_ADDR(A: ADDRESS; DECR: INTEGER) return ADDRESS;
        function MEMORY_ADDRESS(I: INTEGER) return ADDRESS;

        function ">"(A, B: ADDRESS) return BOOLEAN renames ADDR_GT;
        function "<"(A, B: ADDRESS) return BOOLEAN renames ADDR_LT;
        function ">="(A, B: ADDRESS) return BOOLEAN renames ADDR_GE;
        function "<="(A, B: ADDRESS) return BOOLEAN renames ADDR_LE;
        function "-"(A, B: ADDRESS) return INTEGER renames ADDR_DIFF;
        function "+"(A: ADDRESS; INCR: INTEGER) return ADDRESS renames INCR_ADDR;
        function "-"(A: ADDRESS; DECR: INTEGER) return ADDRESS renames DECR_ADDR;
        function "+" (I: INTEGER) return ADDRESS renames MEMORY_ADDRESS;

        pragma inline(ADDR_GT);
        pragma inline(ADDR_LT);
        pragma inline(ADDR_GE);
        pragma inline(ADDR_LE);
        pragma inline(ADDR_DIFF);
        pragma inline(INCR_ADDR);
        pragma inline(DECR_ADDR);
        pragma inline(MEMORY_ADDRESS);

        NO_ADDR : constant ADDRESS;

        type TASK_ID is private;
        NO_TASK_ID : constant TASK_ID;

        type PROGRAM_ID is private;
        NO_PROGRAM_ID : constant PROGRAM_ID;

    private

        type ADDRESS is new INTEGER;

        NO_ADDR : constant ADDRESS := 0;

        type TASK_ID is new INTEGER;
        NO_TASK_ID : constant TASK_ID := 0;

        type PROGRAM_ID is new INTEGER;
        NO_PROGRAM_ID : constant PROGRAM_ID := 0;

    end SYSTEM;
```

F.5 Restrictions on Representation Clauses

This section describes the restrictions on representation clauses in CONVEX Ada.

Size Specification

The size specification T'SMALL is not supported except when the representation specification is the same as the value 'SMALL for the base type.

Record Representation Clauses

Component clauses must be aligned on STORAGE_UNIT boundaries if the component exceeds 4 storage units.

Address Clauses

Address clauses are supported for variables and constants. An object cannot be initialized at the point of declaration if a subsequent address clause is applied to the object.

Interrupts

Interrupt entries are supported for UNIX signals. The Ada *for* clause gives the UNIX signal number.

Representation Attributes

The ADDRESS attribute is not supported for the following entities:

- Packages
- Tasks
- Labels
- Entries

Machine-Code Insertions

Machine-code insertions are supported. The general definition of the package MACHINE_CODE provides an assembly-language interface for the target machine. This package provides the necessary record types needed in the code statement, an enumeration type of all the opcode mnemonics, a set of register definitions, and a set of addressing-mode functions.

The general syntax of a machine-code statement is as follows:

```
CODE_n' (opcode, operand [,operand] ),
```

The parameter *n* indicates the number of operands in the aggregate. A special case arises for a variable number of operands. The operands are listed within a subaggregate in the following format:

```
CODE_n' (opcode, (operand [,operand] ));
```

For those opcodes that require no operands, named notation must be used. The format is as follows:

```
CODE_0' (op => opcode);
```

The *opcode* must be an enumeration literal; it cannot be an object, attribute, or rename. An *operand* can only be an entity defined in the package MACHINE_CODE or with the 'REF attribute.

The arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE. The attribute 'REF may not be used as an argument in any of these functions. Inline expansion of machine-code procedures is supported.

F.6 Conventions for Implementation-Generated Names

There are no implementation-generated names.

F.7 Interpretation of Expressions in Address Clauses

Address clauses are supported for constants and variables. Interrupt entries are specified with the number of the UNIX signal.

F.8 Restrictions on Unchecked Conversions

There are no restrictions on unchecked conversions.

F.9 Restrictions on Unchecked Deallocations

There are no restrictions on unchecked deallocations.

F.10 Implementation Characteristics of I/O Packages

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITS) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as string, where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead.

MAX_REC_SIZE is defined in SYSTEM and can be changed by a program before DIRECT_IO is instantiated to provide an upper limit on the record size. In any case, the maximum size supported is 1024 x 1024 x STORAGE_UNIT bits. DIRECT_IO raises USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITS) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as string, where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM and can be changed by a program before instantiating INTEGER_IO to provide an upper limit on the record size. SEQUENTIAL_IO imposes no limit on MAX_REC_SIZE.