AD-A242 474

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

Publi ... r per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
need ... urden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
Head ... erson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of
Mana ...

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 30 Jan 1991 to 01 Jun 1993

4. TITLE AND SUBTITLE
Encore Computer Corporation, Parallel Ada Development System, Revision 1.0, Encore 91 Series (Model No. 91-0340) under UMAX 3.0 (Host & Target), 910130W1.11114

5. FUNDING NUMBERS

DTIC ELECTE NOV 5 1991 S D C

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-446-0991

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
No software available for distribution per Michele Kee
ADA 11/4/91 mgm telecon

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

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Encore Computer Corporation, Parallel Ada Development System, Revision 1.0, Encore 91 Series (Model No. 91-0340) under UMAX 3.0 (Host & Target), ACVC 1.11.

91-15055

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	

NSN 7540-01-280-550

91 1104 126

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

AVF Control Number: AVF-VSR-446-0991
5-September-1991
90-09-18-ECC

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910130W1.11114
Encore Computer Corporation
Parallel Ada Development System, Revision 1.0
Encore 91 Series (Model No. 91-0340) under UMAX 3.0 =>
Encore 91 Series (Model No. 91-0340) under UMAX 3.0

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 30 January 1991.

Compiler Name and Version: Parallel Ada Development System, Revision 1.0

Host Computer System: Encore 91 Series (Model No. 91-0340) under UMAX 3.0

Target Computer System: Encore 91 Series (Model No. 91-0340) under UMAX 3.0

Customer Agreement Number: 90-09-18-ECC

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

As a result of this validation effort, Validation Certificate 910130W1.11114 is awarded to Encore Computer Corporation. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Steven P. Wilson

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

Chris Demmy

for 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: Encore Computer Corporation

Certificate Awardee: Encore Computer Corporation

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

ACVC Version: 1.11

Ada Implementation:

Ada Compiler Name and Version: Parallel Ada Development System, Revision 1.0

Host Computer System: Encore 91 Series (Model No. 91-0340) under UMAX 3.0

Target Computer System: Encore 91 Series (Model No. 91-0340) under UMAX 3.0

Declaration:

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

Customer Signature

Date

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

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

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

Ada Compiler Validation Capability User's Guide [UG89], 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 21 November 1990.

E28005C	B28006C	C34006D	C35702A	B41308B	C43004A
C45114A	C45346A	C45612B	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	C83026A	C83041A	B85001L	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	BD4008A	CD4022A	CD4022D	CD4024B
CD4024C	CD4024D	CD4031A	CD4051D	CD5111A	CD7004C
ED7005D	CD7005E	AD7006A	CD7006E	AD7201A	AD7201E
CD7204B	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 LONG_INTEGER:

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

A35801E checks that FLOAT'FIRST..FLOAT'LAST may be used as a range constant in a floating-point type declaration; for this implementation that range exceeds the safe numbers and must be rejected. (See 2.3)

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 and C45624B 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.

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

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

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

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

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

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

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

2.3 Test Modifications

Modifications (see section 1.3) were required for 23 tests.

IMPLEMENTATION DEPENDENCIES

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	B91001H	BC1303F	BC3005B	BD2B03A
BD2D03A	BD4003A				

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO; the compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. ARM 3.5.7(12)).

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

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

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

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

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

Gary Beerman
6901 W. Sunrise Blvd.
Ft. Lauderdale FL 33340-9148

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

Gary Beerman
6901 W. Sunrise Blvd.
Ft. Lauderdale FL 33340-9148

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.

PROCESSING INFORMATION

a) Total Number of Applicable Tests	3814
b) Total Number of Withdrawn Tests	83
c) Processed Inapplicable Tests	72
d) Non-Processed I/O Tests	0
e) Non-Processed Floating-Point Precision Tests	201
f) Total Number of Inapplicable Tests	273
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

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.

The tests were compiled, linked, and run on the computer system, as appropriate. The results were captured on the computer system.

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/Switch	Effect
-v	Verbose

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
$MAX_IN_LEN	499
$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:"

MACRO PARAMETERS

$MAX_STRING_LITERAL '"' & (1..V-2 => 'A') & '"'

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

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	16777216
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	Umaxv_88k
$DELTA_DOC	0.0000000004656612873077392578125
$ENTRY_ADDRESS	SYSTEM."+"(16#40#)
$ENTRY_ADDRESS1	SYSTEM."+"(16#80#)
$ENTRY_ADDRESS2	SYSTEM."+"(16#100#)
$FIELD_LAST	2147483647
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_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
$GREATER_THAN_FLOAT_BASE_LAST	1.8E+308
$GREATER_THAN_FLOAT_SAFE_LARGE	5.0E307

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	9.0E37
$HIGH_PRIORITY	99
$ILLEGAL_EXTERNAL_FILE_NAME1	"/illegal/file_name/2}]%2102c.dat"
$ILLEGAL_EXTERNAL_FILE_NAME2	"/illegal/file_name/CE2102C*.dat"
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006D1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	C
$LESS_THAN_DURATION	-100000.0
$LESS_THAN_DURATION_BASE_FIRST	-10000000.0
$LINE_TERMINATOR	ASCII.LF
$LOW_PRIORITY	0
$MACHINE_CODE_STATEMENT	CODE_0' (OP => NOP);
$MACHINE_CODE_TYPE	CODE_0
$MANTISSA_DOC	31
$MAX_DIGITS	15
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648
$MIN_INT	-2147483648

MACRO PARAMETERS

$NAME	TINY_INTEGER
$NAME_LIST	Umaxv_88k
$NAME_SPECIFICATION1	"/u5/acvcl.11/work/ce2" & "X21202A"
$NAME_SPECIFICATION2	"/u5/acvcl.11/work/ce2" & "X21202B"
$NAME_SPECIFICATION3	"/u5/acvcl.11/work/ce3" & "X3119A"
$NEG_BASED_INT	16#F000000E#
$NEW_MEM_SIZE	65535
$NEW_STOR_UNIT	16
$NEW_SYS_NAME	Umaxv_88k
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	RECORD SUBP: OPERAND; END RECORD;
$RECORD_NAME	CODE_0
$TASK_SIZE	32
$TASK_STORAGE_SIZE	1024
$TICK	0.01
$VARIABLE_ADDRESS	VAR_1'ADDRESS
$VARIABLE_ADDRESS1	VAR_2'ADDRESS
$VARIABLE_ADDRESS2	VAR_3'ADDRESS
$YOUR_PRAGMA	PRAGMA PASSIVE

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 – invoke the Ada compiler

SYNOPSIS

ada [*options*] [*source_file*.**a**]... [*linker_options*] [*object_file*.**o**]...

DESCRIPTION

ada executes the Ada compiler and compiles *source_file*. *source_file* must end with the **.a** suffix and must reside in a directory that has been initialized as an Ada library. The **ada.lib** file in this directory is modified after each Ada unit is compiled.

You can specify non-Ada object files (.o files produced by compilers for other languages) to be linked with the specified Ada object files.

By default, **ada** produces only object and nets files. If you specify the **–M** option, the compiler automatically invokes **a.ld** and builds a complete program, with the specified library unit as the main program.

The order of compilation and the order of the files to be passed to the linker can be significant. You can, however, specify command line options in any order.

Specify no more than one of the following options: **–E**, **–e**, **–El**, **–el**, **–ev**.

The options are:

–# *identifier type value* — (define) Define an *identifier* of the specified *type* and *value*. (For further information, see "Ada Preprocessor Reference.")

–a *file_name* — (archive) Treat *file_name* as an object archive file created by **ar**. This option distinguishes archive files, some of which end with **.a**, from Ada source files, all of which end with **.a**.

–d — (dependencies) Analyze for dependencies only, performing neither semantic analysis nor code generation. Update the library, marking any dependent units as uncompiled. The **a.make** utility uses this information to establish dependencies among new files.

–E [*file*] [*directory*] — (error output) Use **a.error** to process error messages. If neither *file* nor *directory* is specified, **ada** directs a brief listing to standard output, placing the raw error messages in *ada_source*.**err**. If *file* is specified, **ada** places the raw error messages in the file with that name. If *directory* is specified, **ada** places the raw error messages in *directory/source*.**err**. You can use the file of raw error messages as input to **a.error**.

–e — (error) Use **a.error** to process compilation error messages, sending the listing to standard output. Only the source lines containing errors are listed.

–El [*file*] [*directory*] — (error listing) Same as the **–E** option, except that error messages are interspersed among source lines.

–el — (error listing) Same as the **–e** option, except that error messages are interspersed among source lines.

–ev — (error **vi**(1)) Process syntax error messages using **a.error**, embed them in the source file, and call the environment editor ERROR_EDITOR. If no editor is specified, call **vi**(1). (If ERROR_EDITOR is defined, the environment variable ERROR_PATTERN should also be defined. ERROR_PATTERN is an editor search command that locates the first occurrence of the string ### in the error file.)

–K — (keep) Keep the intermediate language (IL) file produced by the compiler front end; name the file *Ada_source*.**i**, and place it in the **.objects** directory.

−L *library_name*
(library) Operate in the Ada library *library_name*. The default is the current working directory.

Note: If two files of the same name from different directories are compiled in the same Ada library using the **−L** option, the second compilation overwrites the first, even if the contents and unit names are different. For example, ada /usr/directory2/foo.a -L /usr/PADS/test

overwrites

```
ada /usr/directory1/foo.a -L /usr/PADS/test
```

in library /usr/PADS/test.

−l*file_abbreviation*
(library search) Direct the linker, **ld**(1), to search the library file specified by *file_abbreviation*.

−M *[unit_name]*
(main) Produce an executable program by linking *unit_name* as the main program. *unit_name* must be either a parameterless procedure or a parameterless function returning an integer. Unless it is being compiled by this invocation of **ada**, *unit_name* must already have been compiled. The executable program is named **a.out** unless you use the **−o** option to specify another name.

−M *source_file*
(main) Produce an executable program by compiling and linking *source_file*. The main unit of the program is assumed to be the root name of the **.a** file (in **foo.a**, for example, the main unit is **foo**). Unless you use the **−o** option to specify another name, the executable program is named **a.out**. Only one **.a** file can be preceded by **−M**.

−o *executable_file*
(output) Name the executable program *executable_file* rather than the default, **a.out**. This option is used in conjunction with the **−M** option.

−O[0-9]
(optimize) Invoke the code optimizer (OPTIM3). The optional digit provides the level of optimization. The default is **−O4**.

This version of the compiler includes a preliminary M88k-specific optimizer. The optimizer schedules load instructions to avoid pipeline conflicts and moves instructions to the delay slots of branches and calls. Since it can be slow for some programs, it is enabled only at optimization levels greater than 4.

−O	full optimization
−O0	no optimization
−O1	no hoisting
−O2	no hoisting but more passes
−O3	no hoisting but even more passes
−O4	hoisting from loops
−O5	hoisting from loops but more passes
−O6	hoisting from loops with maximum passes
−O7	hoisting from loops and branches
−O8	hoisting from loops and branches, more passes

	–O9 hoisting from loops and branches, maximum passes
	Note: Hoisting from branches (and **case** alternatives) can be slow and does not always provide significant performance gains. You might therefore want to suppress it.
–P	(preprocessor) Invoke the Ada preprocessor, **a.app**.
–R *library_name*	(recompile instantiation) Force analysis of all generic instantiations, causing reinstantiation of any that are out of date.
–S	(suppress) Apply **pragma** SUPPRESS to the entire compilation for all suppressible checks.
–sh	(show) Display the name of the executable compiler, but do not execute it. *(Several versions of* PADS *may exist on one system.* The **ada** command in any *PADS_location*/**bin** executes the correct version of the compiler based upon visible library directives.)
–T	(timing) Print timing information for the compilation.
–v	(verbose) Print compiler version number, date and time of compilation, name of file compiled, command input line, total compilation time, and error summary line. Provide information about the object file's use of storage. With OPTIM3 the output format of compression (the size of optimized instructions) is shown as a percentage of input (unoptimized instructions).
–w	(warnings) Suppress warning diagnostics.

FILES

ada.lib	Library reference file
gnrx.lib	Generic instantiation reference file
GVAS.lock, gnrx.lock	Lock the library while reading or writing special library files
GVAS_table	Address assignment file
.imports	Imported Ada units directory
.lines	Line number reference files directory
.nets	DIANA nets files directory
.objects	(global) object files directory

SEE ALSO

a.app(1), a.error(1), a.ld(1), a.make(1)
ld(1), vi(1)

DIAGNOSTICS

The diagnostics produced by the compiler are intended to be self-explanatory. Most refer to the *Ada Language Reference Manual* (Ada RM). Each Ada RM reference includes a section number and, optionally, a paragraph number enclosed in parentheses.

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.

NAME

a.ld – invoke the Ada prelinker

SYNOPSIS

a.ld [*options*] *unit_name* [*ld_options*]

DESCRIPTION

a.ld collects the object files needed to make *unit_name* a main program. **a.ld** then calls the linker ld(1) to link all Ada object files and any non-Ada object files required to produce an executable image in a.out.

unit_name specifies the main program and must be a nongeneric subprogram. If *unit_name* is a function, it must return a value of type STANDARD.INTEGER. This integer result is passed back to the shell as the status code of the execution.

All arguments after *unit_name* are passed to ld(1). These arguments may be ld options, archive libraries, library abbreviations, or object files

The options are:

–DX	(debug) Debug memory overflow. Use this option in cases where linking a large number of units produces the error message "local symbol overflow".
–E *unit_name*	(elaborate) Elaborate *unit_name* as early in the elaboration order as possible.
–F	(files) Display a list of dependent files in order, but suppress linking.
–L *library_name*	(library) Operate in the Ada library *library_name*. The default is the current working directory.
–o *executable_file*	(output) Name the executable file *executable_file* rather than the default, **a.out**.
–r	Retain relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent run of a link editor. The link editor does not complain about unresolved references, and the output file is not executable.
–sh	(show) Display the name of the **a.ld** executable file, but do not execute it. (Several versions of PADS can exist on one system. *PADS_location*/**bin/a.ld** executes the correct version of a.ld based upon directives visible in the **ada.lib** file.)
–T *target*	Use *target* as the target run-time environment.
–U	(units) Print a list of dependent units in order, but suppress linking.
–v	(verbose) Print the linker command before executing it.
–V	(verify) Print the linker command, but suppress execution.

The **a.ld** tool reads the nets files produced by the Ada compiler to determine dependency information. The tool produces an exception mapping table and a unit elaboration table and passes this information to the linker.

a.ld reads instructions for generating executables from the **ada.lib** file in the Ada libraries on the search list. In addition to information generated by the compiler, these instructions include WITH*n* directives, which enable the automatic linking of object modules compiled from other languages or Ada object modules not named in context clauses in the Ada source. The **ada.lib** file can contain any number of WITH*n* directives, but the directives must be numbered consecutively, beginning at WITH1. The directives have the following form: WITH1:LINK:*object_file*: `WITH2:LINK:`*archive_file*: WITH*n* directives can be placed in the local Ada library or in any Ada libraries on the search list. A WITH*n* directive in the local library or earlier on the search list hides any WITH*n* directive with the same number in a library later on the search list.

Use the tool **a.info** to change or display library directives in the current library.

FILES

a.out	Default output file
.nets	DIANA nets files directory
.objects/*	Ada object files
*PADS_location/standard/**	Start-up and standard library routines

SEE ALSO

ada(1), a.info(1)
ld(1)

DIAGNOSTICS

a.ld produces self-explanatory error messages for missing files, etc. Additional messages are produced by the linker, **ld**(1).

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 -1.701411183E+308 .. 1.70141183E+308;

type DURATION is delta 0.001 range -2147483.648 .. 2147483.647;

type SHORT_INTEGER is range -32768 .. 32767;

type SHORT_FLOAT is digits 6 range -3.40282E+38 .. 3.40282E+38;

type TINY_INTEGER is range -128 .. 127;

    ..........

end STANDARD;
```

B

Appendix F of the Ada Language Reference Manual

The Parallel Ada Development System provides the full Ada language as specified in the Ada Language Reference Manual (Ada RM). Within the Ada RM, a number of sections contain the annotation *implementation dependent*, meaning that the interpretation of the section is left to the compiler implementor. This appendix describes the implementation-dependent characteristics of the PADS compiler.

PADS has attempted to provide an essentially unlimited capability to program in Ada. Consequently, applications programmers can usually program in Ada according to the Ada RM and good engineering practices without consideration of any PADS specifics.

PRAGMAS AND THEIR EFFECTS

This section provides a brief description of every pragma supported by PADS. You can find additional information about some of the pragmas under discussions of particular language constructs elsewhere in this manual and in the *Parallel Ada Development System User's Guide*.

pragma CONTROLLED

This pragma is recognized by the implementation but has no effect in the current release.

pragma ELABORATE

This pragma is implemented as described in Appendix B of the Ada RM.

pragma EXTERNAL_NAME

This pragma enables you to specify an external link name for an Ada variable or subprogram so that the object can be referenced 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. Objects must be variables defined in a package specification; subprograms can be either library level or within a package specification. For further information about **pragma** EXTERNAL_NAME, see Chapter 4 of this manual.

pragma IMPLICIT_CODE

Use this pragma with caution. The pragma, used only within the declarative part of a machine code procedure, specifies whether implicit code generated by the compiler is allowed (ON) or disallowed (OFF). Implicit code includes preamble and postamble code (for example, code used to move parameters to and from the stack). A warning is generated if implicit code is required and OFF is specified.

Use of **pragma** IMPLICIT_CODE does not eliminate code generated for run-time checks, nor does it eliminate call/return instructions. (These can be eliminated by **pragma** SUPPRESS and **pragma** INLINE, respectively.)

For further information about **pragma** IMPLICIT_CODE, see Chapter 3 of this manual.

pragma INLINE

This pragma is implemented as described in Appendix B of the Ada RM, with one addition: Recursive calls can be expanded with the pragma up to the maximum depth of 4. Warnings are generated for bodies that are not available for inline expansion. When applied to subprograms that declare tasks, packages, exceptions, types, or nested subprograms, **pragma** INLINE is ignored and causes a warning to be issued.

pragma INLINE_ONLY

When used in the same way as **pragma** INLINE, this pragma indicates to the compiler that the subprogram must *always* be inlined. This is very important for some code procedures. **pragma** INLINE_ONLY also saves code space by suppressing the generation of a callable version of the routine. If you erroneously make an INLINE_ONLY subprogram recursive, a warning is generated and a PROGRAM_ERROR is raised at run time.

pragma INTERFACE

This pragma, with parameters *language* and *subprogram*, supports calls to Ada, C, Pascal, and FORTRAN functions. You can also use **pragma** INTERFACE to call code written in unspecified languages, specifying UNCHECKED as the language name. The Ada specifications can be either functions or procedures.

For Ada, the compiler generates the call as if it were a call to an Ada procedure, but it does not expect a matching procedure body.

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

For Pascal, the types of parameters and the result type for functions must be scalar types, access types, or the predefined **type** ADDRESS in **package** SYSTEM. Record and array objects can be passed by reference using the 'ADDRESS attribute.

For FORTRAN, all parameters are passed by reference The parameter types must have **type** SYSTEM.ADDRESS, and the result type for a function must be a scalar type.

Use UNCHECKED to interface to an unspecified language, such as assembler. The compiler generates the call as if it were a call to an Ada procedure, but it does not expect a matching Ada procedure body.

For related information, see the section entitled "Parameter Passing" later in this appendix.

pragma INTERFACE_NAME

This pragma enables direct reference in Ada to variables or subprograms defined in another language. **pragma** INTERFACE_NAME uses the following format:

pragma INTERFACE_NAME (*Ada_subprogram*, *link_name*);

where *Ada_subprogram* denotes either an object or a subprogram.

The pragma replaces all references to *Ada_subprogram* with an external reference to *link_name* in the object file.

If *Ada_subprogram* denotes an object, 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 or an access type and cannot be any of the following:

- Loop variable
- Constant
- Initialized variable
- Array
- Record

If *Ada_subprogram* denotes a subprogram, a **pragma** INTERFACE must already have been specified for the subprogram.

The *link_name* must be constructed as the linker expects; for example, C variable names must be prefaced with an underscore. The following example makes the C global variable **errno** available within an Ada program:

```
package PACKAGE_NAME is
     ...
     ERRNO:INTEGER;
     pragma INTERFACE_NAME (ERRNO,"_errno");
     ...
end PACKAGE_NAME;
```

For further information about **pragma** INTERFACE_NAME, see Chapter 4 of this manual.

pragma LINK_WITH

Use this pragma to pass arguments to the linker. The pragma can appear in any declarative part and accepts one argument, a constant string expression. This argument is passed to the target linker whenever the unit containing the pragma is included in a link.

For example, the following package puts the **-lm** option on the command line for the linker whenever MATH is included in the linked program:

```
package MATH is
   pragma LINK_WITH( "-lm ");
end;
```

And the following package links with the named object file **sin.o**:

```
package MATH is
-------------------------------------------------------------------
--
--          SIN is a routine written in C or assembly: the object
--          for the routine is in the object file sin.o
-------------------------------------------------------------------
--
    function SIN (X:FLOAT)          return FLOAT;
    pragma interface (C, SIN);
    pragma LINK_WITH("sin.o");
end MATH;
```

If the constant string expression begins with "-", the string is left untouched. If the string begins with neither "-" nor "/", then the string is prefixed with "./".

pragma LIST

This pragma is implemented as described in Appendix B of the Ada RM.

pragma MEMORY_SIZE

This pragma is recognized by the implementation but has no effect in the current release. PADS does not allow modification of **package** SYSTEM by means of pragmas. You can, however, achieve the same effect by copying the file **system.a** in library **standard** to a local Ada library and recompiling it there with new values.

pragma NO_IMAGE

This pragma suppresses the generation of the image array used for the 'IMAGE attribute of enumeration types, eliminating the overhead required to store the array in the executable image. Any attempt to use the 'IMAGE attribute on a type whose image array has been suppressed results in a warning at compile time and causes the **exception** PROGRAM_ERROR to be raised at run time.

pragma NON_REENTRANT

This pragma takes one argument, which can be the name of a library subprogram or a subprogram declared immediately within a library package specification or body. The pragma prevents the subprogram from being called recursively, allowing the compiler to perform specific optimizations. You can apply **pragma** NON_REENTRANT to a subprogram or a set of overloaded subprograms within a package specification or package body.

pragma NOT_ELABORATED

This pragma suppresses the generation of elaboration code, issuing warnings if elaboration code is required. The pragma prevents elaboration of a package that is either part of the run-time system, a configuration package, or an Ada package that is referenced from a language other than Ada. **pragma** NOT_ELABORATED can appear only in a library package specification.

pragma OPTIMIZE

This pragma is recognized by the implementation but has no effect in the current release. For code optimization options, see the **ada – O** entry in Chapter 9 of the *Parallel Ada Development System User's Guide.*

pragma OPTIMIZE_CODE

This pragma specifies whether the compiler optimizes code (ON) or does not optimize code (OFF). When OFF (the default) is specified, the compiler generates the code as specified. You can use the pragma in any subprogram.

You can suppress optimization selectively by using this pragma at the subprogram level. Inline subprograms are optimized even if OPTIMIZE_CODE(OFF) is specified, unless **pragma** OPTIMIZE_CODE(OFF) is also specified for the caller.

pragma PACK

This pragma causes the compiler to minimize gaps between components in the representation of composite types. Objects larger than a single STORAGE_UNIT are packed to the nearest STORAGE_UNIT. Storage optimization generally results in less efficient manipulation of the packed data type.

pragma PAGE

This pragma is implemented as described in Appendix B of the Ada RM. The pragma is also recognized by the source code formatting tool, **a.pr**.

pragma PASSIVE

This pragma directs the compiler to optimize certain tasks into passive tasks. The pragma can be applied to a task or task type declared immediately within a library package specification or body.

pragma PASSIVE has three forms:

pragma PASSIVE;
pragma PASSIVE(SEMAPHORE);
pragma PASSIVE(INTERRUPT, *nnn*);

The statements in the task body may prevent the intended optimization. In such cases, a warning is generated at compile time and the exception TASKING_ERROR is raised at run time.

For additional information about **pragma** PASSIVE and passive tasks, see the section entitled "Passive Tasks" in Chapter 2 of this manual.

pragma PRIORITY

This pragma is implemented as described in Appendix B of the Ada RM. The allowable range for **pragma** PRIORITY is 0 .. 99.

pragma SHARE_CODE

This pragma enables multiple instantiations of the same generic procedure or package body to share object code. A "parent" instantiation is created, and subsequent instantiations of the same types can share the parent's object code, reducing program size and compilation times.

pragma SHARE_CODE takes the name of a generic unit or a generic instantiation as its first argument and either of the identifiers TRUE or FALSE as its second argument. When the first argument is the name of 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 **pragma** SHARE_CODE is allowed only 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. The extent to which code is shared by instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit.

You can substitute the name **pragma** SHARE_BODY for the name **pragma** SHARE_CODE.

pragma SHARED

This pragma is recognized by the implementation but has no effect in the current release.

pragma STORAGE_UNIT

This pragma is recognized by the implementation but has no effect in the current release. PADS does not allow modification of **package** SYSTEM by means of pragmas. You can achieve the same effect by copying the file **system.a** in library **standard** to a local Ada library and recompiling it there with new values. (However, you should *not* redefine STORAGE_UNIT.)

pragma SUPPRESS

This pragma is implemented as described in Appendix B of the Ada RM, except that DIVISION_CHECK and, in some cases, OVERFLOW_CHECK cannot be suppressed.

Using **pragma** SUPPRESS(ALL_CHECKS) is equivalent to writing, at the same point in the program, a **pragma** SUPPRESS for each of the checks listed in Ada RM 11.7.

pragma SUPPRESS(EXCEPTION_TABLES) tells the code generator not to generate, for the enclosing compilation unit, the tables that are normally generated to identify exception regions. This reduces the size of the static data required for a unit but also disables exception handling within that unit.

pragma SYSTEM_NAME

This pragma is recognized by the implementation but has no effect in the current release. PADS does not allow modification of **package** SYSTEM by means of pragmas. You can, however, achieve the same effect by copying the file **system.a** in library **standard** to a local Ada library and recompiling it there with new values.

pragma VOLATILE

This pragma, with its argument, *object*, guarantees that loads and stores to the named object are performed as expected after optimization. For example:

```
memory_flag : integer;
pragma volatile (memory_flag);
```

PREDEFINED PACKAGES AND GENERICS

The following predefined Ada packages, specified by Ada RM Appendix C(22), are provided in the library **standard**:

- **generic function** UNCHECKED_CONVERSION
- **generic package** DIRECT_IO
- **generic package** SEQUENTIAL_IO
- **generic procedure** UNCHECKED_DEALLOCATION
- **package** CALENDAR
- **package** IO_EXCEPTIONS
- **package** LOW_LEVEL_IO
- **package** MACHINE_CODE
- **package** STANDARD
- **package** SYSTEM
- **package** TEXT_I O

Specification of package SYSTEM

```
with UNSIGNED_TYPES;
package SYSTEM is
    pragma SUPPRESS(ALL_CHECKS);
    pragma SUPPRESS(EXCEPTION_TABLES);
    pragma NOT_ELABORATED;
    type NAME is ( umaxv_88k );
    SYSTEM_NAME      : constant NAME := umaxv_88k;
    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 := 0.01;
    -- Other System-Dependent Declarations
    subtype PRIORITY is INTEGER range 0 .. 99;
    MAX_REC_SIZE : integer := 64*1024;
    type ADDRESS is private;
    function ">" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
    function "<" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
    function ">="(A: ADDRESS; B: ADDRESS) return BOOLEAN;
    function "<="(A: ADDRESS; B: ADDRESS) return BOOLEAN;
    function "-" (A: ADDRESS; B: ADDRESS) return INTEGER;
    function "+" (A: ADDRESS; I: INTEGER) return ADDRESS;
    function "-" (A: ADDRESS; I: INTEGER) return ADDRESS;
    function "+" (I: UNSIGNED_TYPES.UNSIGNED_INTEGER) return
        ADDRESS;
    function MEMORY_ADDRESS
        (I: UNSIGNED_TYPES.UNSIGNED_INTEGER) return ADDRESS
            renames "+";
    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;
    type SIG_STATUS_T is array(1..64) of boolean;
    pragma PACK(SIG_STATUS_T);
    SIG_STATUS_SIZE: CONSTANT := 8;
private
    type ADDRESS is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
    NO_ADDR : constant ADDRESS := 0;
    pragma BUILT_IN(">");
    pragma BUILT_IN("<");
    pragma BUILT_IN(">=");
    pragma BUILT_IN("<=");
```

```
    pragma BUILT_IN("-");
    pragma BUILT_IN("+");
end SYSTEM;
```

package CALENDAR

package CALENDAR operates as specified in Ada RM 9.6. It uses the clock function in **package** CALENDAR.LOCAL_TIME (located in the file **calendar_s.a**), which uses the operating system service routines GETTIMEOFDAY and LOCALTIME to get the current time.

package MACHINE_CODE

package MACHINE_CODE provides an assembly language interface for the target machine, including the necessary record types needed in the *code* statement (see Ada RM 13.8), an enumeration type containing all the opcode mnemonics, a set of register definitions, and a set of addressing mode functions. Also supplied (for use only in units that **with** MACHINE_CODE) are **pragma** IMPLICIT_CODE and the attribute 'REF. For the specification of the package, see the section entitled "package MACHINE_CODE" in Chapter 3.

Machine code statements take operands of **type** OPERAND, a private type that forms the basis of all machine code address formats for the target.

The general syntax of a machine code statement is

CODE_*n*'(*opcode*, *operand* [, *operand*]);

where *n* indicates the number of operands in the aggregate.

In the following example, CODE_3 is a record 'format' whose first argument is an enumeration value of **type** OPCODE followed by three operands of **type** OPERAND:

```
CODE_3'(add, r10, r11, b'ref);
```

For those opcodes requiring no operands, you must use named notation (see Ada RM 4.3(4)):

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

opcode must specify an enumeration literal (that is, it cannot specify an object, an attribute, or a rename). *operand* can specify only an entity defined in MACHINE_CODE or the 'REF attribute.

The 'REF attribute denotes the effective address of the first storage unit allocated to the object. 'REF is not supported for a package, task unit, or entry. For details, see the section entitled "'REF" later in this appendix.

Arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE.

As an example of machine code insertions, the **procedure** OS_EXTEND requests the operating system to extend the program stack space to a new address:

```
procedure os_extend (new_top : in system.address) is
    -- Extend the stack according to BCS Chapter 5
    pragma implicit_code(off)
    use machine_code;
begin
    code_3'(add, r10, r31, r0);   -- Save sp
    code_3'(add, r31, r2, r0);    -- Set sp to new limit
    code_2'(st, r0, r31+0);       -- Access it; this extends the stack
    code_3'(add, r31, r10, r0)); -- Restore sp
    code_1'(jmp, r1);
end os_extend;
```

package SEQUENTIAL_IO

Sequential I/O is currently implemented for variant records, with one restriction: The maximum size possible for the record is always written. The same is true for direct I/O. For unconstrained records and arrays, the **constant** SYSTEM.MAX_REC_SIZE can be set prior to the elaboration of the generic instantiation of SEQUENTIAL_IO or DIRECT_IO. For example, if unconstrained strings are written, SYSTEM.MAX_REC_SIZE effectively restricts the maximum size of strings. If you know the maximum size of such strings, you can set the SYSTEM.MAX_REC_SIZE prior to instantiating SEQUENTIAL_IO for the string type. You can reset this variable after the instantiation with no effect.

package UNSIGNED_TYPES

The **package** UNSIGNED_TYPES illustrates the definition of and services for the unsigned types supplied in this version of PADS. Use this package at your own risk. We do not warrant its effectiveness or legality, either expressly or by implication.

We plan to withdraw this implementation of UNSIGNED_TYPES if and when the Ada Joint Program Office and the Ada community reach agreement on a practical specification of unsigned types. We will then standardize our implementation based on that accepted version at the earliest practical date.

The package is supplied in comment form because the actual package cannot be expressed in normal Ada – the types are not symmetric about 0, as is required by the Ada RM. This package is supplied and is accessible through the Ada WITH*n* statement, as if it were present in source form.

Example:

```
with unsigned_types;
procedure foo( xxx: unsigned_types.unsigned_integer) is ...
```

Note: Use package UNSIGNED_TYPES at your own risk.

Specification of package UNSIGNED_TYPES

```
-- package unsigned_types is
--
-- type unsigned_integer is range 0 .. (2**32 - 1); -- 0..4294967295
--function "=" (a, b: unsigned_integer) return boolean;
--function "/="(a, b: unsigned_integer) return boolean;
--function "<" (a, b: unsigned_integer) return boolean;
--function "<="(a, b: unsigned_integer) return boolean;
--function ">" (a, b: unsigned_integer) return boolean;
--function ">="(a, b: unsigned_integer) return boolean;
--function "+" (a, b: unsigned_integer) return unsigned_integer;
--function "-" (a, b: unsigned_integer) return unsigned_integer;
--function "+" (a   : unsigned_integer) return unsigned_integer;
--function "-" (a   : unsigned_integer) return unsigned_integer;
--function "*" (a, b: unsigned_integer) return unsigned_integer;
--function "/" (a, b: unsigned_integer) return unsigned_integer;
--function "mod"(a, b: unsigned_integer) return unsigned_integer;
--function "rem"(a, b: unsigned_integer) return unsigned_integer;
--function "**" (a, b: unsigned_integer) return unsigned_integer;
--function "abs"(a, b: unsigned_integer) return unsigned_integer;
--
-- type unsigned_short_integer is range 0 .. (2**16 - 1);-- 0..65535
--function "=" (a, b: unsigned_short_integer) return boolean;
--function "/="(a, b: unsigned_short_integer) return boolean;
--function "<" (a, b: unsigned_short_integer) return boolean;
--function "<="(a, b: unsigned_short_integer) return boolean;
--function ">" (a, b: unsigned_short_integer) return boolean;
--function ">="(a, b: unsigned_short_integer) return boolean;
--function "+" (a, b: unsigned_short_integer)
-- return unsigned_short_integer;
--function "-" (a, b: unsigned_short_integer)
-- return unsigned_short_integer;
--function "+" (a   : unsigned_short_integer)
-- return unsigned_short_integer;
--function "-" (a   : unsigned_short_integer)
-- return unsigned_short_integer;
--function "*" (a, b: unsigned_short_integer)
-- return unsigned_short_integer;
--function "/" (a, b: unsigned_short_integer)
-- return unsigned_short_integer;
--function "mod"(a, b: unsigned_short_integer)
-- return  unsigned_short_integer;
--function "rem"(a, b: unsigned_short_integer)
-- return  unsigned_short_integer;
--function "**" (a, b: unsigned_short_integer)
```

```
-- return  unsigned_short_integer;
--function "abs"(a, b: unsigned_short_integer)
-- return  unsigned_short_integer;
--
-- type unsigned_tiny_integer is range 0 .. (2**8 - 1); -- 0..255
--function "=" (a, b: unsigned_tiny_integer) return boolean;
--function "/="(a, b: unsigned_tiny_integer) return boolean;
--function "<" (a, b: unsigned_tiny_integer) return boolean;
--function "<="(a, b: unsigned_tiny_integer) return boolean;
--function ">" (a, b: unsigned_tiny_integer) return boolean;
--function ">="(a, b: unsigned_tiny_integer) return boolean;
--function "+" (a, b: unsigned_tiny_integer)
-- return unsigned_tiny_integer;
--function "-" (a, b: unsigned_tiny_integer)
-- return unsigned_tiny_integer;
--function "+" (a   : unsigned_tiny_integer)
-- return unsigned_tiny_integer;
--function "-" (a   : unsigned_tiny_integer)
-- return unsigned_tiny_integer;
--function "*" (a, b: unsigned_tiny_integer)
-- return unsigned_tiny_integer;
--function "/" (a, b: unsigned_tiny_integer)
-- return unsigned_tiny_integer;
--function "mod"(a, b: unsigned_tiny_integer)
-- return  unsigned_tiny_integer;
--function "rem"(a, b: unsigned_tiny_integer)
-- return  unsigned_tiny_integer;
--function "**" (a, b: unsigned_tiny_integer)
-- return  unsigned_tiny_integer;
--function "abs"(a, b: unsigned_tiny_integer)
-- return  unsigned_tiny_integer;
-- end unsigned_types;
```

IMPLEMENTATION-DEFINED ATTRIBUTES

This section describes the attributes defined by PADS.

'TASK_ID

For a task object or a value T, T'TASK_ID yields the unique task ID associated with the task. The value of this attribute is of the type SYSTEM.TASK_ID.

'REF

The 'REF attribute denotes the effective address of the first of the storage units allocated to the object. 'REF is not supported for a package, task unit, or entry. This attribute has two forms: X'REF and SYSTEM.ADDRESS'REF(N). X'REF, used only in machine code procedures, designates an operand within a code statement. SYSTEM.ADDRESS'REF(N) can be used anywhere to convert an integer expression to an address.

X'REF

This attribute generates a reference to the entity to which it is applied.

In X'REF, X must be either a constant, variable, procedure, function, or label. The attribute returns a value of the **type** MACHINE_CODE.OPERAND and can only be used to designate an operand within a code statement.

The instruction generated by the code statement in which the attribute occurs can be preceded by additional instructions needed to facilitate the reference (for example, loading a base register). If the declarative section of the procedure contains **pragma** IMPLICIT_CODE (OFF) and additional code is required, a warning is generated.

References may also cause the generation of run-time checks. You can use **pragma** SUPPRESS to eliminate these checks.

Example:

```
CODE_1'(BSR, PROC'REF);
CODE_2'(ld, r11, X.ALL(Z)'REF);
```

For further information, see the section entitled "Ada Entities as Operands" in Chapter 3 of this manual.

SYSTEM.ADDRESS'REF(N)

In SYSTEM.ADDRESS'REF(N), SYSTEM.ADDRESS must be the **type** SYSTEM.ADDRESS; N must be an expression of **type** UNIVERSAL_INTEGER. The attribute returns a value of **type** SYSTEM.ADDRESS, which represents the address designated by N.

The effect of this attribute is similar to the effect of an unchecked conversion from integer to address. You should, however, use SYSTEM.ADDRESS'REF(N) in the following circumstances (and in these circumstances, N must be static):

- Within any of the run-time configuration packages. Use of UNCHECKED_CONVERSION within an address clause would require the generation of elaboration code, and the configuration packages are not elaborated.
- In any instance where N is greater than INTEGER'LAST. Such values are required in address clauses that reference the upper portion of memory. UNCHECKED_CONVERSION in these instances would require that the expression be specified as a negative integer.

- To place an object at an address. The *integer_value* in the following example is converted to an address for use in the address representation clause. The form avoids UNCHECKED_CONVERSION and is also useful for 32-bit unsigned addresses:

```
--place an object at an address
for object use at ADDRESS'REF (integer_value)
--to use unsigned addresses
for VECTOR use at SYSTEM.ADDRESS'REF(16#808000d0#);
TOP_OF_MEMORY : SYSTEM.ADDRESS := SYSTEM.ADDRESS'REF(16#FFFFFFFF#);
```

RESTRICTIONS ON MAIN PROGRAMS

In PADS, a main program must be a nongeneric subprogram that is either a procedure or a function returning an Ada STANDARD.INTEGER (the predefined type). A main program may be neither a generic subprogram nor an instantiation of a generic subprogram.

GENERIC DECLARATIONS

In PADS, a generic declaration and the corresponding body need not be part of the same compilation, nor must they exist in the same Ada library. If a single compilation contains two versions of the same unit, an error is generated.

SHARED OBJECT CODE FOR GENERIC SUBPROGRAMS

The PADS compiler generates code for a generic instantiation that can be shared by other instantiations of the same generic, thus reducing the size of the generated code and increasing compilation speed.

Shared code instantiations do entail some overhead because the generic actual parameters must be accessed indirectly and, in the case of a generic package instantiation, declarations in the package must also be accessed indirectly. In addition, unshared instantiations permit greater optimization because exact actual parameters are known. You must therefore determine whether space or time is more critical in a specific application.

[illegible] impractical in some cases. If the generic has a formal private type [illegible] generated to accommodate an instantiation with an arbitrary [illegible] extremely inefficient.

pragma SHARE_CODE lets you control whether an instantiation generates unique code or shares code with other similar instantiations.

This pragma is allowed only in the following places: immediately within a declarative part, immediately within a package specification, or after a library unit in a compilation but before any subsequent compilation unit. **pragma** SHARE_CODE takes the following form:

pragma SHARE_CODE (*generic*, *boolean_literal*)

You can apply **pragma** SHARE_CODE to a generic declaration or to individual instantiations. When **pragma** SHARE_CODE references a generic unit, it sets sharing on or off for all instantiations of that generic unless overridden by specific SHARE_CODE pragmas for individual instantiations. When it references an instantiated unit, **pragma** SHARE_CODE sets sharing on or off for that unit alone. The default is to share all generics that can be shared unless the unit uses **pragma** INLINE.

The compiler shares code by default if the generic formal type parameters are restricted to integer, enumeration, or floating-point. To override the default, use the **pragma** SHARE_CODE(*name*, FALSE). If there are formal subprogram parameters, instantiations are not shared unless you specify **pragma** SHARE_CODE(*name*, TRUE).

Generics are shared by default if a parent is visible, except in the following cases:

- When generic formal types other than integer, enumeration, SYSTEM.ADDRESS or floating-point are used
- When **pragma** INLINE is applied to a generic subprogram or instantiation or to a subprogram visible at the library level within a generic package or instantiation
- When the representations of the actual type parameters are not the same for each of the instantiations
- When the generic has a formal **in out** parameter and the subtype of the corresponding actual is not the same as the subtype of the formal parameter

Note that a parent instantiation (the instantiation that creates the shareable body) is independent of any individual instantiation. Therefore, reinstantiation of a generic with different parameters has no effect on other compilations that reference it. The unit that caused compilation of a parent instantiation need not be referenced in any way by subsequent units that share the parent instantiation.

The unit SHARED_IO in the library **standard** instantiates all Ada I/O generic packages for the most commonly used base types. Thus, any instantiation of an Ada I/O generic package shares one of the parent instantiation generic bodies unless the following pragma is specified:

```
pragma SHARE_CODE ( generic, FALSE );
```

REPRESENTATION CLAUSES

This section describes the PADS implementation of representation clauses.

Representation Clauses

PADS supports bit-level representation clauses.

Representation Pragmas

The language-defined **pragma** PACK is the only representation pragma supported by PADS.

Length Clauses

PADS supports all length clauses.

Enumeration Representation Clauses

PADS supports enumeration representation clauses.

Record Representation Clauses

Representation clauses are based on the target machine's word, byte, and bit order numbering, so that VADS compilers are consistent with machine architecture manuals for both 'big-endian' and 'little-endian' machines. Bits within a STORAGE_UNIT are also numbered according to the target machine manuals. You need not understand the default layout for records and other aggregates, since the use of record representation clauses gives you fine control over the layout. You can align record fields correctly with structures and other aggregate types from other languages by specifying the location of each element explicitly. On the MC88100, PADS operates in the big-endian type ordering configuration.

Figure B-1 illustrates MC88100 addressing and bit numbering.

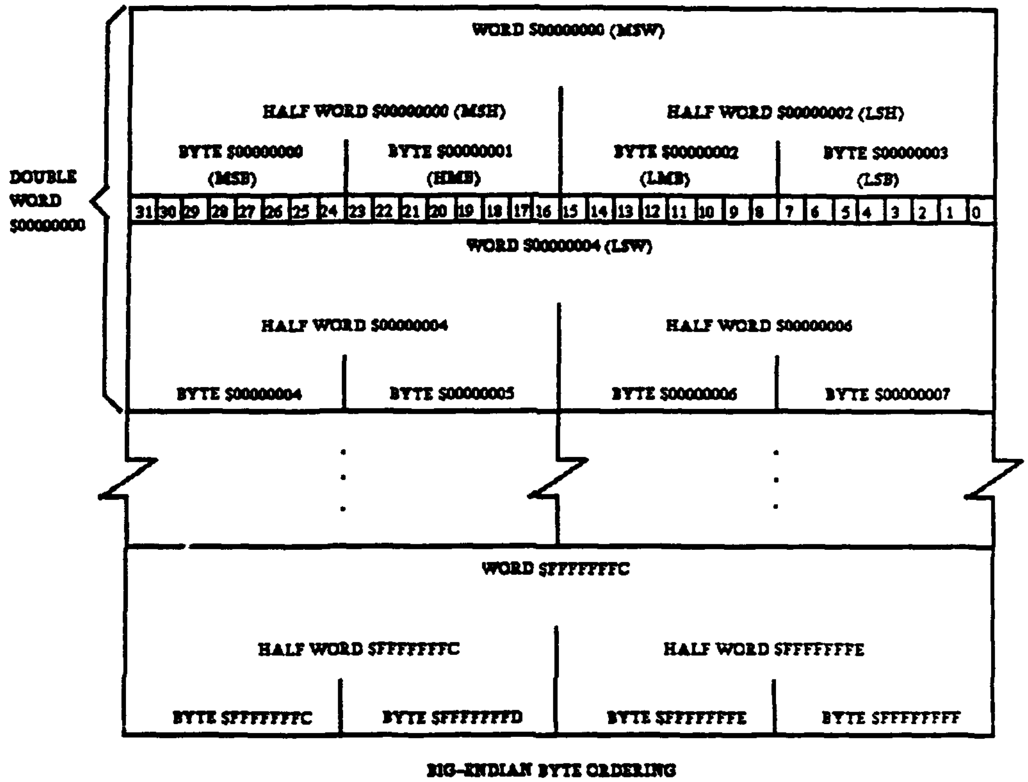

Figure B-1: MC88100 Addressing and Bit Numbering

The only restrictions on record representation clauses are the following:

- If a component does not start and end on a storage unit (byte) boundary, it must be stored within 4 consecutive bytes.
- A component that is itself a record must occupy a power of 2 bits. Components that are of a discrete type or packed array can occupy an arbitrary number of bits, subject to the preceding restriction.

Address Clauses

PADS supports address clauses for objects and entries.

Note: Use with caution code that references memory-mapped devices using a **for use at** clause to locate an object at the I/O address. The default optimization of the compiler eliminates redundant moves to and from memory. If this causes problems, compile with **pragma** OPTIMIZE_CODE(OFF).

Interrupt Entries

PADS allows task entries to be associated with operating system signals. The operating system handles all interrupts and faults initially and then returns control to the user program as a signal.

The available signals are described in *UMAX V Programmers Guide*. Due to restrictions in the operating system, some of the signals cannot be caught. Although an attempt to assign an entry to these signals does not result in an error, the operating system will not deliver the signal to the program.

The Ada run-time system discourages attempts to catch the timer-related signals.

The following example program shows you how to attach to the CTRL-c or interrupt-from-keyboard signal:

```
with iface_intr;
with system;    use system;
with text_io;
task interrupt is
   entry SIGINT;
      for SIGINT use at address'ref(iface_intr.sigint); -- interrupt
end;
task body interrupt is
begin
   loop
      accept SIGINT do
         text_io.put_line("SIGINT");
      end;
   end loop;
end;
```

Signal handlers are set up for the following signals by the PADS run-time system:

```
#define   SIGFPE     8    /* floating point exception */
#define   SIGSEGV   11    /* segmentation violation */
#define   SIGTRAP    5    /* trace trap */
#define   SIGALRM   14    /* alarm clocks */
```

If a task entry is attached to SIGFPE, NUMERIC_ERROR exceptions are not raised correctly. If a task entry is attached to SIGSEGV, STORAGE_ERROR exceptions may not be raised correctly. If a task entry is attacked to SIGALRM, delay statements and time slicing do not work correctly.

Use of signal handlers is complicated when non-Ada routines are involved. For further information, see Chapter 4 of this manual.

Change of Representation

PADS supports change of representation.

The package SYSTEM

For the specification of **package** SYSTEM, see the section entitled "Predefined Packages and Generics" earlier in this appendix. The specification is also available on line in the file **system.a** in the release library **standard**. The **pragmas** SYSTEM_NAME, STORAGE_UNIT, and MEMORY_SIZE are recognized by the implementation but have no effect. PADS does not allow SYSTEM to be modified by means of pragmas. However, you can achieve the same effect by recompiling **package** SYSTEM with altered values. Note that such recompilation causes other units in the library **standard** to become out of date. Consequently, you should recompile SYSTEM in some library other than **standard**.

Representation Attributes

PADS supports the 'ADDRESS attribute for the following entities:

- Variables
- Constants
- Procedures
- Functions

If the prefix of an 'ADDRESS attribute is an object that is not aligned on a storage unit boundary, the attribute yields the address of the storage unit containing the first bit of the object. This is consistent with the definition of the 'FIRST_BIT attribute.

All other Ada representation attributes are fully supported.

Representation Attributes of Real Types

PADS supports these attributes. See the section entitled "Predefined Packages and Generics" earlier in this appendix.

Machine Code Insertions

PADS supports machine code insertions. See Chapter 3 of this manual for details.

Interface to Other Languages

PADS supports interface to other languages. See Chapter 4 of this manual and the section entitled "Pragmas and Their Effects" earlier in this appendix for details.

Unchecked Programming

PADS provides both UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION.

Unchecked Storage Deallocations

Any object that is allocated can be deallocated. No checks are currently performed on released objects. However, when an object is deallocated, its access variable is set to null. Subsequent deallocations using the null access variable are ignored.

Unchecked Type Conversions

The predefined **generic function** UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

PARAMETER PASSING

Parameters are passed in registers or by pushing values (or addresses) on the stack. Extra information is passed for records ('CONSTRAINED) and for arrays (dope vector address).

Registers r2 through r9 are used to pass parameters. Parameters of 64-bit floating-point type are passed in a register pair. Other parameters of scalar type, access type, or the type SYSTEM.ADDRESS are passed in a single register. If all parameter registers have been used, a parameter is transmitted in storage by pushing its value on the stack.

Likewise, a function result of scalar type, access type, or the **type** SYSTEM.ADDRESS is returned in register r2 or in the pair r2, r3, as appropriate.

Small results are returned in registers; large results with known targets are passed by reference. Large results of anonymous target and known size are passed by reference to a temporary created in the caller. Large results of anonymous target and unknown size are returned by copying the value down from a temporary created by the callee so that the space used by the temporary can be reclaimed.

The compiler assumes the following calling conventions, defined in Object Compatibility Standard (OCS):

1. Caller copies first 8 argument words into r2-r9.
2. Caller pushes additional arguments on stack.
3. Caller calls callee.
4. Callee builds display and allocates space for local variables.
5. Callee stores any registers it modifies in the set r14 .. r25.
6. Callee executes.
7. Callee restores registers saved in Step 5.
8. If callee is a function, callee leaves result in r2 (or in the pair r2, r3 for a 64-bit floating-point result).
9. Callee deallocates local storage.
10. Callee returns to caller.
11. Caller copies back any **out** parameters or function values.
12. Caller deallocates the space used for arguments on the stack.

Note: Compilers for other languages may follow calling conventions other than those expected by PADS. Use the debugger, **a.db**, to verify that the call interface is the expected one.

When calling C routines (defined with **pragma** INTERFACE (C, *Ada_subprogram*)), the caller allocates stack space for each parameter passed in a register in accordance with the 88open Consortium Ltd. Object Compatibility Standard (OCS).

When compiler conventions are not compatible, or when interfacing to assembly language, you can build a call interface explicitly using machine code insertions. For further information, see Chapter 3 of this manual.

CONVERSION AND DEALLOCATION

The predefined **generic function** UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

There are no restrictions on the types with which **generic function** UNCHECKED_DEALLOCATION can be instantiated. No checks are performed on released objects.

PROCESS STACK SIZE

The stack limit for the main program is set in the CONFIGURATION_TABLE structure in the **package** V_USR_CONF. The default value is

MAIN_TASK_STACK_SIZE => 256000

The stack limit for tasks is also set in the configuration table. Its default value is

DEFAULT_TSK_STACK_SIZE => 10_240

For information on how to modify these values for your program, see Appendix C of the *Parallel Ada Development System User's Guide*.

TYPES, RANGES, AND ATTRIBUTES

This section describes the PADS implementation of the following types:

- Numeric literals
- Enumeration types
- Discrete types
- The **type** STRING
- Integer types
- Floating-point types
- Fixed-point types
- Array types

Numeric Literals

PADS uses unlimited precision arithmetic for computations with numeric literals.

Enumeration Types

PADS allows an unlimited number of literals within an enumeration type.

Attributes of Discrete Types

PADS defines the image of a character that is not a graphic character as the corresponding 2- or 3-character identifier from **package** ASCII of Ada RM, Appendix C. The identifier is in upper case without enclosing apostrophes. For example, the image for a carriage return is the 2-character sequence CR (ASCII.CR).

The type STRING

Except for memory size, PADS places no specific limit on the length of the predefined **type** STRING. Any type derived from the **type** STRING is similarly unlimited.

By default, strings are represented with a single character in each byte of memory. Thus, storage for string objects is automatically minimized.

Integer Types

Table B-1 summarizes the attributes of the predefined integer types.

Table B-1: Attributes of Integer Types

Name of Attribute	AttributeValue of INTEGER	AttributeValue of SHORT_INTEGER	AttributeValue of TINY_INTEGER
SIZE	32	16	8
FIRST	-2_147_483_648	-32_768	-128
LAST	2_147_483_647	32_767	127

Operation of Floating-Point Types

Table B-2 summarizes the attributes of PADS floating-point types.

Table B-2: Attributes of Floating-Point Types

Name of Attribute	Attribute Value of FLOAT	Attribute Value of SHORT_FLOAT
SIZE	64	32
FIRST	-1.79769313486232E+308	-3.40282E+38
LAST	1.79769313486232E+308	3.40282E+38
DIGITS	15	6
MANTISSA	51	21
EPSILON	8.88178419700125E-16	9.53674316406250E-07
EMAX	204	84
SMALL	1.94469227433161E-62	2.58493941422821E-26
LARGE	2.57110087081438E+61	1.93428038904620E+25
SAFE_EMAX	1021	125
SAFE_SMALL	2.22507385850720E-308	1.17549435082229E-38
SAFE_LARGE	2.24711641857789E+307	4.25352755827077E+37
MACHINE_RADIX	2	2
MACHINE_MANTISSA	53	24
MACHINE_EMAX	1024	128
MACHINE_EMIN	-1021	-125
MACHINE_ROUNDS	TRUE	TRUE
MACHINE_OVERFLOWS	TRUE	TRUE

Fixed-Point Types

PADS provides fixed-point types mapped to the supported integer sizes.

Operation of Fixed-Point Types

Table B-3 summarizes the attributes of the PADS fixed-point type DURATION.

Table B-3: Attributes of type DURATION

Name of Attribute	Attribute Value for DURATION
SIZE	32
FIRST	-2147483.648
LAST	2147483.647
DELTA	1.0E-03
MANTISSA	31
SMALL	1.0E-3
LARGE	2147483.647
FORE	8
AFT	3
SAFE_SMALL	1.0E-3
SAFE_LARGE	2147483.647
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

Array Types

PADS array bound limits are:

INTEGER'FIRST: –2,147,483,648
INTEGER'LAST: 2,147,483,647

INPUT/OUTPUT

The PADS I/O system is implemented using UMAX V operating system services. Both formatted and binary I/O are available. There are no restrictions on the types with which DIRECT_IO and SEQUENTIAL_IO can be instantiated, except that the element size must be less than a maximum specified by the variable SYSTEM.MAX_REC_SIZE. Since you can set this variable to any value prior to the generic instantiation, you can use any element size. DIRECT_IO can be instantiated with unconstrained types, but each element is padded out to the maximum possible for that type or to SYSTEM.MAX_REC_SIZE, whichever is smaller. No checking, other than normal static Ada type checking, is done to ensure that values from files are read into correctly sized and typed objects.

PADS file and terminal input-output are identical in most respects, differing only in the frequency of buffer flushing. Output is buffered (buffer size is 1024 bytes). The buffer is always flushed after each write request if the destination is a terminal. The **procedure** FILE_SUPPORT.ALWAYS_FLUSH (FILE_PTR) causes the buffer associated with FILE_PTR to be flushed after all subsequent output requests. Refer to the source code for **file_spprt_b.a** in the **standard** library. Note that the **limited private type** FILE_TYPE, defined in TEXT_IO, is derived from the **type** FILE_PTR. Currently, you must convert between them using UNCHECKED_CONVERSION, because the derivation happens in the private part of the specification of TEXT_IO. For example, the following procedure stops buffering for standard output:

```
with text_io;
with file_support;
with unchecked_conversion;
procedure dont_buffer(file: text_io.file_type) is
   function cvt is new unchecked_conversion(
      source => text_io.file_type,
      target => file_support.file_ptr);
begin
   file_support.always_flush(cvt(file));
end;
```

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 a 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 before instantiating DIRECT_IO to provide an upper limit on the record size. The maximum size supported is 1024 * 1024 * 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 a 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 SEQUENTIAL_IO to provide an upper limit on the record size. SEQUENTIAL_IO imposes no limit on MAX_REC_SIZE.

Implementation-Defined Values of the Input/Output Packages

The PADS-defined values in the input/output packages are as follows:

- In **package** TEXT_IO
  type COUNT **is range** 0..INTEGER'LAST;
  subtype FIELD **is** INTEGER **range** 0..INTEGER'LAST;
- In **package** DIRECT_IO
  type COUNT **is range** 0..2_147_483_647;