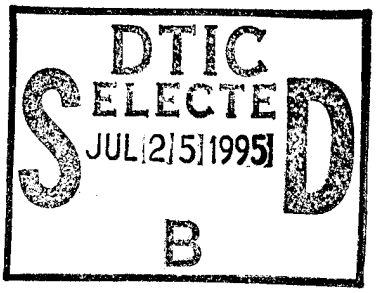
REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 12, 1995	Final

4. TITLE AND SUBTITLE:
Ada Compiler Validation Summary Report, VC# 950606W1.11383
Harris Computer Systems Corporation -- Compiler Name: Harris Ada, Version 2.1

5. FUNDING NUMBERS



6. AUTHOR(S)

Systems Technology Branch, Standard Languages Section

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility
Language Control Facility, 645 C-CSG/SCSL
Area B, Building 676
Wright-Patterson Air Force Base, OH 45433-6503

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office, Defense Information System Agency
Code JEXEV, 701 S. Courthouse Rd., Arlington, VA
22204-2199

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

This Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 6 June 1995.
Host Computer System: Harris NH6202 under PowerUNIX, 2.1
Target Computer System: Harris NH6202 under PowerUNIX, 2.1

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

15. NUMBER OF PAGES
82

16. PRICE

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

AVF Control Number: AVF-VSR-604.0695
Date VSR Completed: 12 June 1995
95-02-28-HAR

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 950606W1.11383
Harris Computer Systems Corporation
Harris Ada, Version 2.1
Harris NH6202 under PowerUNIX, 2.1

(Final)

Prepared By:
Ada Validation Facility
88 CG/SCTL
Wright-Patterson AFB OH 45433-5707

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 6 June 1995.

Compiler Name and Version: Harris Ada, Version 2.1

Host Computer System: Harris NH6202
under PowerUNIX, 2.1

Target Computer System: Same as host

Customer Agreement Number: 95-02-28-HAR

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

As a result of this validation effort, Validation Certificate 950606W1.11383 is awarded to Harris Computer Systems Corporation. This certificate will expire on March 31, 1998.

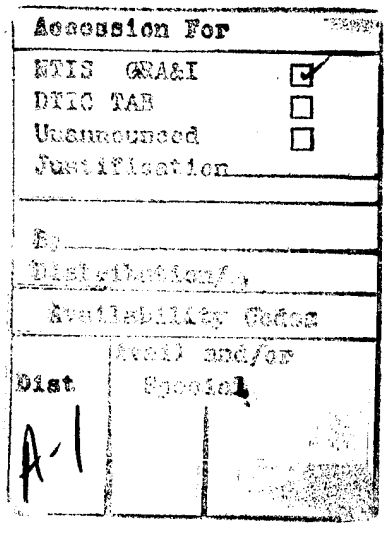
This report has been reviewed and is approved.

Brian P. Andrews

Ada Validation Facility
Brian P. Andrews
AVF Manager
88 CG/SCTL
Wright-Patterson AFB OH 45433-5707

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

Ada Joint Program Office
Donald J. Reifer
Director, AJPO
Defense Information Systems Agency,
Center for Information Management



DECLARATION OF CONFORMANCE

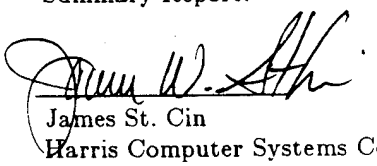
Customer:	Harris Computer Systems Corporation
Certificate Awardee:	Harris Computer Systems Corporation
Ada Validation Facility:	ASD/SCEL, Wright-Patterson AFB
ACVC Version:	1.11

Ada Implementation

Ada Compiler Name and Version:	Harris Ada Version 2.1
Host Computer System:	NH6202
Host Operating system:	PowerUNIX 2.1
Target Computer System:	NH6202
Target Operating System:	PowerUNIX 2.1

Declaration:

I, the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard (ANSI/MIL-STD-1815A. ISO 8652-1987, FIPS 119) as tested in this validation and documented in the Validation Summary Report.



James St. Cin
Harris Computer Systems Corporation
Manager, Contracts

5-9-95
Date

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro95] 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 [Pro95]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

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

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

1.2 REFERENCES

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

[Pro95] Ada Compiler Validation Procedures, Version 4.0, Ada Joint Program Office, January 1995.

[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 removing 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.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
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 [Pro95].
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 22 November 1993.

B27005A	E28005C	B28006C	C32203A	C34006D	C35507K
C35507L	C35507N	C35507O	C35507P	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	C37310A	B41308B
C43004A	C45114A	C45346A	C45612A	C45612B	C45612C
C45651A	C46022A	B49008A	B49008B	A54B02A	C55B06A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
C83026A	B83026B	C83041A	B85001L	C86001F	C94021A
C97116A	C98003B	BA2011A	CB7001A	CB7001B	CB7004A
CC1223A	BC1226A	CC1226B	BC3009B	BD1B02B	BD1B06A
AD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A	CD2A41A
CD2A41E	CD2A87A	CD2B15C	BD3006A	BD4008A	CD4022A
CD4022D	CD4024B	CD4024C	CD4024D	CD4031A	CD4051D
CD5111A	CD7004C	ED7005D	CD7005E	AD7006A	CD7006E
AD7201A	AD7201E	CD7204B	AD7206A	BD8002A	BD8004C
CD9005A	CD9005B	CDA201E	CE2107I	CE2117A	CE2117B
CE2119B	CE2205B	CE2405A	CE3111C	CE3116A	CE3118A
CE3411B	CE3412B	CE3607B	CE3607C	CE3607D	CE3812A
CE3814A	CE3902B				

2.2 INAPPLICABLE TESTS

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

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

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

The following 20 tests check for the predefined type LONG_INTEGER; for this implementation, there is no such type:

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

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

C96005B uses values of type DURATION's base type that are outside the range of type DURATION; for this implementation, the ranges are the same.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

The tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

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 an external sequential file is exceeded; this implementation cannot restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3115A attempts resetting of an external file with OUT_FILE mode, which is not supported with multiple internal files associated with the same external file when they have different modes.

CE3304A checks that SET_LINE_LENGTH and SET_PAGE_LENGTH raise USE_ERROR if they specify an inappropriate value for the external file; there are no inappropriate values for this implementation.

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 15 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
B85005G	B85005H	B91001H	BC1313F	BC3005B	BD2B03A
BD2D03A	BD4003A				

CE3804H was graded passed by Evaluation Modification as directed by the AVO. This test requires that the string "-3.525" can be read from a file using FLOAT_IO and that an equality comparison with the numeric literal '-3.525' will evaluate to TRUE; however, because -3.525 is not a model number, this comparison may evaluate to FALSE (LRM 4.9:12). This implementation's compile-time and run-time evaluation algorithms differ; thus, this check for equality fails and Report.Failed is called at line 81, which outputs the message "WIDTH CHARACTERS NOT READ." All other checks were passed.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

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

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

Ed Kelly
Harris Computer Systems Corporation
2101 West Cypress Creek Road
Ft Lauderdale FL 33309-1892
(305) 973-5340

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 [Pro95].

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

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

a) Total Number of Applicable Tests	3795	
b) Total Number of Withdrawn Tests	104	
c) Processed Inapplicable Tests	70	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	271	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

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 executed on the host computer system. The results were captured on the host 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 following options were used for testing this implementation:

Compiler option/switch		Effect
[-el]	(errors list)	Errors & source to stdout
[-i]	(info)	Suppress information only messages
[-w]	(warnings)	Suppress warning and info messages

Link options		Effect
[-o exec_file]	(output)	Name the generated program "exec_file" instead of the default name, a.out.
[-w]	(warnings)	Suppress warning messages

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

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

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

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	8
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	3221225469
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	HARRIS_POWER
$DELTA_DOC	2.0**(-31)
$ENTRY_ADDRESS	SYSTEM.PHYSICAL_ADDRESS(16)
$ENTRY_ADDRESS1	SYSTEM.PHYSICAL_ADDRESS(17)
$ENTRY_ADDRESS2	SYSTEM.PHYSICAL_ADDRESS(18)
$FIELD_LAST	2147483647
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100000.0
$GREATER_THAN_DURATION_BASE_LAST	10000000.0
$GREATER_THAN_FLOAT_BASE_LAST	3.5E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E+38

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        1.0E+308

$HIGH_PRIORITY          255

$ILLEGAL_EXTERNAL_FILE_NAME1
                        /no/such/file/name

$ILLEGAL_EXTERNAL_FILE_NAME2
                        /this/file/does/not/exist

$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_3'(or_r,r0,r0,r0);

$MACHINE_CODE_TYPE      operand

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2_147_483_648

$MIN_INT                -2147483648

$NAME                   TINY_INTEGER
```

MACRO PARAMETERS

$NAME_LIST	HARRIS_POWER
$NAME_SPECIFICATION1	/jas2/AT/VAL/ppc604//tests/ce/CE2120A
$NAME_SPECIFICATION2	/jas2/AT/VAL/ppc604//tests/ce/CE2120B
$NAME_SPECIFICATION3	/jas2/AT/VAL/ppc604//tests/ce/CE3119A
$NEG_BASED_INT	16#F000000E#
$NEW_MEM_SIZE	3221225469
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	HARRIS_POWER
$PAGE_TERMINATOR	ASCII.LF & ASCII.FF
$RECORD_DEFINITION	type code_1 (op: opcode) is record oprnd_1: operand ; end record ;
$RECORD_NAME	code_1
$TASK_SIZE	32
$TASK_STORAGE_SIZE	10240
$TICK	0.01
$VARIABLE_ADDRESS	FCNDECL.SPACE(1024)
$VARIABLE_ADDRESS1	FCNDECL.SPACE(1024)
$VARIABLE_ADDRESS2	FCNDECL.SPACE(1024)
$YOUR_PRAGMA	EXTERNAL_NAME

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.

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.

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 -2_147_483_648 .. 2_147_483_647;
   type SHORT_INTEGER is range -32_768 .. 32_767;
   type TINY_INTEGER is range -128 ..127;

   type FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;
   type LONG_FLOAT is digits 15 range -1.79769313486232E+308
                                   ..  1.79769313486232E+308;

   type DURATION is delta 2.0**(-14) range -131_072.0 .. 131_072.0;
   ..........
end STANDARD;
```

Table 5-1. ada Options

Option	Meaning	Function
`-a` *name*	(ada source)	Save preprocessed source file in *name*.
`-b`	(object)	Symbolic object listing to stdout.
`-bound`	(bound)	Pass `-bound` to a.ld. The `-M` option must also be specified.
`-c`	(compress)	Compress executable program by removing "dead" routines. The `-M` option must also be specified.
`-d`	(dependencies)	Analyze for dependencies but do not compile.
`-e`	(error)	Process syntax error messages using `a.error` and direct the output to stdout, (i.e., list only the errors).
`-el`	(error listing)	Generate an error listing using `a.error` and direct it to stdout, i.e., list the entire source file.
`-ev`	(error vi)	Process syntax error message using `a.error`, embed them in the source file, and call `vi`.
`-g`[*level*]	(debug level)	Specify the debug level. The level may be n (none), 0 (lines) or 1 (full). (Default is -g0).
`-hapse` *HAPSE*	(hapse)	Select a HAPSE installation.
`-i`	(info)	Suppress informational messages.
`-ld` *file*	(loader)	Pass *file* to `ld`. The `-M` option must also be specified.
`-lib` *library*	(library)	Select a HAPSE library.
`-m`	(map)	Display an object map. The `-M` option must also be specified.
`-multiplexed`	(multiplexed)	Pass `-multiplexed` to a.ld. The `-M` option must also be specified.
`-nshare` *item*	(no sharelib)	Pass "`-nshare` *item*" to `a.ld`. The `-M` option must also be specified. See `a.ld`.
`-nshare_all`	(nshare)	Pass "`-nshare_all`" to `a.ld`. The `-M` option must also be specified. See `a.ld`.
`-o` *output*	(output)	After compilation, call `a.ld`. Rename the executable program to the specified name. The `-M` option must also be specified.
`-pp` *options*	(preprocess)	Pass *options* to `a.pp`.
`-q`	(quick)	Do not invoke code generator; check syntax and semantics and update `ada.lib`.
`-s` *file*	(source)	Compile *file*, where *file* is an Ada source file not ending in `.a`, `.ada`, or `.pp`.
`-share` *item*	(sharelib)	Pass "`-share` *item*" to `a.ld`. The `-M` option must also be specified. See `a.ld`.
`-share_all`	(share)	Pass "`-share_all`" to `a.ld`. The `-M` option must also be specified. See `a.ld`.
`-sm` *mode*	(share mode)	Set the sharing compilation mode to `shared`, `non_shared`, or `both`. (Default -s -sm non_shared).

Table 5-1. ada Options (Cont.)

Option	Meaning	Function
`-u`	(update)	Update the library `ada.lib` even if syntax errors are present.
`-v`	(verbose)	Display information about the compilation.
`-w`	(warnings)	Suppress warning and informational diagnostics.
`-E` `-E` *file* `-E` *directory*	(error output)	Process error messages using `a.error` and direct the output to `stdout` as well as (`-E`) *ada_source.err*, (`-E` *file*) *file*, or (`-E` *directory*) *directory/ada_source.err*.
`-El` `-El` *file* `-El` *directory*	(error listing)	Process source listing with errors using `a.error` as well as (`-El`) *ada_source.err*, (`-El` *file*) *file*, or (`-El` *directory*) *directory/ada_source.err*.
`-H`	(help)	Display this description and stop.
`-K`	(keep)	Keep the `IL` file created by the front end.
`-L`	(list)	Generate a source listing to stdout, even if there are no errors.
`-M` *unit_name* `-M` *ada_source.a*	(main)	After compilation, call `a.ld` to produce an executable program with the named main program or source root name.
`-N`	(No sharing)	Apply pragma SHARE_BODY (FALSE) to all generics (see "Pragma SHARE_BODY" on page 9-15.).
`-O[`*1-3*`]`	(optimize)	Apply pragma OPT_LEVEL to the compilation. (See "Pragma OPT_LEVEL" on page 9-10.) If a number is not specified, then level 2 is assumed. Levels 1-3 correspond to MINIMAL, GLOBAL, and MAXIMAL, respectively, as described in "Optimizations Performed at the Various Levels" on page 10-3. (Default is -O1).
`-P`	(preprocess)	Invoke `a.pp` before compiling source files.
`-Q`*flags*	(qualifier)	Supply qualifier flags to the compiler (see "Back-End Qualifier (-Q) Flags" on page 5-8).
`-R` *library*	(recompile)	Force updating of instantiations.
`-S`	(suppress)	Apply pragma SUPPRESS_ALL
`-T`	(timings)	Display the wall and CPU times used by the compiler.
`-V`	(very verbose)	Display verbose compilation information.

Table 5-7. a.ld Options

Option	Meaning	Function
`-bound`	(bound)	Set the default task weight to BOUND.
`-c[v|V]`	(compress)	Compress executable image by removing "dead" routines.
`-d`	(default)	Use default supplied libraries. For use with the `-share` and `-nshare` options.
`-hapse` *HAPSE*	(HAPSE)	Select a HAPSE installation.
`-i` *elab_sym*	(initialize)	Insert the elaboration routine specified by *elab_sym* at the elaboration list head.
`-lib` *library*	(library)	Select a HAPSE library.
`-m`	(map)	Produce a link map and direct it to stdout.
`-map` file	(map)	Produce a run-time configuration map file ("file") for this program.
`-multiplexed`	(multiplexed)	Set the default task weight to MULTIPLEXED. (This is the default).
`-n`	(no datarec)	Do not produce a data recording file.
`-nshare` *item*	(no sharelib)	Do not allow units from HAPSE library *item* to come from shared-objects. See `-d`, `-share_all`, `-share`, `-nshare_all`.
`-nshare_all`	(nshare)	Disable the use of any shared-objects, forcing static linking. See `-share_all`, `-nshare`, `-share`.
`-ntrace`	(NightTrace)	Use an `ntrace` run-time which requires the NightTrace daemon, `ntraceud`, for execution. This option cannot be combined with `-trace`.
`-o` *executable_file*	(output)	Use *executable_file* as the name of the output rather than the default, `a.out`.
`-share` *item*	(sharelib)	Force units from HAPSE library *item* to come from a shared-object. See `-d`, `-share_all`, `-nshare`, `-nshare_all`.
`-share_all`	(share)	Enable the use of HAPSE shared-objects as well as system shared libraries. See `-nshare`, `-nshare_all`, `-share`.
`-shmem` *"params"*	(shared memory)	Supplies a quoted string containing shared package configuration parameters. Consult ?Section 6.2.2 for a description of these parameters.
`-trace`	(trace)	Link with a tracing version of the run-time. This option cannot be combined with `-ntrace`.
`-update`	(update)	Update shared-object for the selected HAPSE library. No link of an application program will be performed.
`-v`	(verbose)	Display the linker commands before execution.
`-w`	(warnings)	Suppress warning diagnostics.

Table 5-7. a.ld Options (Cont.)

Option	Meaning	Function
`-A` *"args"*	(analyze)	Invoke `a.analyze` with a list of arguments placed within quotation marks.
`-F`	(files)	Display a list of required files, suppressing linking.
`-H`	(help)	Display this description and stop.
`-O`[*executable_file*]	(optimize)	Perform optimizations at link time by invoking the `a.analyze` optimizer. If *executable_file* is specified, it overrides any name specified by the `-o` option, or `a.out` if no `-o` option is specified. (-O is not used by default).
`-U`	(units)	Display a list of required units in elaboration order, suppressing linking.
`-V`	(verify)	Display the linker commands, but suppress execution.

9
Implementation-Dependent Characteristics

NOTE

This chapter serves as Appendix F of the Ada RM.

Program Structure and Compilation

A "main" program must be a non-generic subprogram without parameters that is either a procedure or a function returning an Ada `STANDARD.INTEGER` (the predefined type). A "main" program cannot be a generic subprogram or an instantiation of a generic subprogram.

Pragmas

The following text discusses implementation-dependent and implementation-defined pragmas. A pragma syntax summary appears in the *HAPSE Pocket Reference*.

Pragma BUILT_IN

NOTE

Pragma `BUILT_IN` is for internal HAPSE use only; it is not applicable to user-defined subprograms.

The implementation-defined pragma `BUILT_IN` specifies that a subprogram is defined to be an implicit operation intrinsic to the compiler and that it, therefore, does not require an explicit subprogram body. The syntax of the pragma is as follows:

```
pragma BUILT_IN (subprogram_name);
```

All calls to the specified subprogram are replaced with code sequences that perform the operations that are implicitly defined for the subprogram, if any such code sequences are required.

Pragma CONTROLLED

The implementation-dependent pragma CONTROLLED is recognized by the implementation but does not have an effect in this release.

Pragma DEBUG

The implementation-defined pragma DEBUG provides a method for specifying the debug level for a compilation unit from within the Ada source code. The format of the pragma is:

```
pragma DEBUG (unit_name, debug_level);
```

where *unit_name* is the name of the compilation unit for which the debug level is being specified, and where *debug_level* is the debug level which should be used for that compilation unit. The possible values for *debug_level* are NONE, LINES, and FULL. This pragma is allowed only immediately following the unit which is specified as the *unit_name* argument. It applies only to the unit which is specified. If applied to a specification, the debug level does not apply to the body or any separate bodies of the unit. If applied to a body, the debug level does not apply to any separate bodies of the unit. If the debug level is desired for any such units, it must be specified for them too.

The pragma is meaningless when applied to a generic unit. If so applied, it will not be applied to any instantiations of that generic. The debug level applied to an instantiation is the debug level of the unit which contains it, or, if the instantiation is library-level, is determined in the same way as for any other library-level unit.

Pragma DEFAULT_HARDNESS

The implementation-defined pragma DEFAULT_HARDNESS sets the default hardness for any memory bound to LOCAL via a pragma MEMORY_POOL. See "Pragma DEFAULT_HARDNESS" on page 20-2 for a complete description.

Pragma DEPRECATED_FEATURE

NOTE

Pragma DEPRECATED_FEATURE is reserved for internal HAPSE use only; it is not intended for use in user-defined code.

This pragma is used to mark packages that have been deprecated and may be significantly changed or completely removed in future releases of HAPSE. Whenever a compilation unit requires a unit (e.g., via a `with` clause) that has been marked with this pragma, a compiler alert (diagnostic) is issued. The alert consists of a standard HAPSE diagnostic

header followed by the exact text of the string that is the single required argument of the pragma.

Pragma ELABORATE

The implementation-dependent pragma ELABORATE is implemented as described in Appendix B of the Ada RM.

Pragma EXTERNAL_NAME

The implementation-defined pragma EXTERNAL_NAME provides a method for specifying an alternative *link name* for variables, functions and procedures. The required parameters are the simple name of the object and a string constant representing the link name. Link names are case-sensitive. Note that this pragma is useful for referencing functions and procedures that have had pragma INTERFACE applied to them, in such cases where the functions or procedures have link names that do not conform to Ada identifiers. The pragma must occur after any such applications of pragma INTERFACE and within the same declarative part or package specification that contains the object. The compiler automatically removes the first leading underscore.

Example:

Use the EXTERNAL_NAME pragma to make an Ada routine visible externally. The link name may then be referenced by other programs to call the Ada routine. Some sample Ada code follows:

```
package stuff is
--
    function call_me return integer;
    pragma EXTERNAL_NAME (call_me, "Ada_call");
--
end stuff;

package body stuff is
--
    function call_me return integer is
    begin
        return 0;
    end call_me;
--
end stuff;
```

The routine call_me is now visible under the link name "Ada_call". Other subprograms can now call this function by referring to the external name given to the routine. For example, an Ada subprogram can call the function as follows:

```
procedure example is
    Ada_value : integer;
    function Ada_call return integer;
```

```
    pragma interface( Ada, Ada_call );
begin
    Ada_value := Ada_call;
end example;
```

In general, calling Ada subprograms in this manner should be avoided because of the complexities of:

- Elaboration order
- Up-level referencing
- Hidden compiler-generated parameters (generics, unconstrained arrays and records)
- Run-time elaboration (tasking, exception)

Pragma FAST_INTERRUPT_TASK

The implementation-defined pragma FAST_INTERRUPT_TASK provides ultra-fast interrupt handling. Use of this pragma causes the task to execute directly at interrupt-level. Use of this pragma requires severe limitations on the form of the task and the actions taken during rendezvous.

The compiler enforces many of the restrictions on the task; however, others cannot be detected and are not enforced by the compiler. For details, see "Pragma FAST_INTERRUPT_TASK" on page 21-4.

Pragma GROUP_CPU_BIAS

The implementation-defined pragma GROUP_CPU_BIAS specifies the CPU bias for all the servers in a given group. See "Pragma GROUP_CPU_BIAS" on page 20-10 for a complete description.

Pragma GROUP_PRIORITY

The implementation-defined pragma GROUP_PRIORITY specifies the operating system priority of all the servers in a given group. See "Pragma GROUP_PRIORITY" on page 20-10 for a complete description.

Pragma GROUP_SERVERS

The implementation-defined pragma GROUP_SERVERS controls the number of servers for a particular group, including the PREDEFINED group. See "Pragma GROUP_SERVERS" on page 20-11 for a complete description.

Pragma IMPLICIT_CODE

The implementation-defined pragma `IMPLICIT_CODE` provides a way to eliminate the stack frame and the copying of parameters when using the `machine_code` package. This pragma takes a single argument (`ON` or `OFF`). When `OFF`, it does not generate code for the argument copies, nor does it generate any return code upon exiting. It can be used as an optimization for writing `machine_code` routines to eliminate the generation of the implicit code. An example demonstrating the use of this pragma is given in "Usage" on page 9-36.

Pragma INLINE

The implementation-dependent pragma `INLINE` is implemented as described in Section 6.3.2 and Appendix B of the Ada RM, however, there are a number of restrictions on inline subprogram expansion. The compiler will issue a warning, not perform the inline expansion, and output code for a subprogram call, if any of these restrictions are violated or exceeded.

The restrictions and limitations on inline subprogram expansion include:

- The body of the subprogram must be compiled before it can be expanded inline. The HAPSE recompilation utility, **`a.make`**, will attempt to compile bodies that define inline subprograms before bodies that use inline subprograms, however, if two bodies contain mutual inline dependencies **`a.make`** will choose, in an arbitrary manner, which to compile first. (See "Inline Dependencies and a.make" on page 5-39.)
- There are a number of Ada constructs that will prevent inline expansion if they appear in the declarations of a subprogram marked with pragma `INLINE`. These constructs include tasks, most generic instantiations, and (inner) subprograms that perform up-level addressing.
- Direct or indirect recursive calls are never inline expanded.
- The actual parameters to the inline expanded subprogram must not contain task objects, must not contain dependent arrays, and must have complete type declarations.
- Subprograms marked with pragma `INTERFACE` are never inline expanded.

The uncontrolled use of inline expansion can adversely affect the performance of the HAPSE compiler itself. Inline expansion can be controlled by using HAPSE configuration management described in "Inline Parameters" on page 12-4.

Subprograms that contain machine-code insertion statements are always inline expanded if they are marked with pragma `INLINE`, regardless of any configuration limits.

WARNING

It is not recommended practice to inline machine-code procedures, as the compiler does not track register uses and definitions made by machine-code procedures. Inlining of machine-code procedures is supported, but the user should exercise caution.

Pragma INTERFACE

The implementation-dependent pragma INTERFACE is recognized by the implementation and supports calls to C, Fortran, Ada and other language functions. The Ada specifications can be either functions or procedures. For C and Fortran, all parameters must have mode IN. See Chapter 11 of this manual for examples.

For C, the types of parameters as well as the result type for functions must be scalar, access, or the predefined type SYSTEM.ADDRESS. Record and array objects can be passed by reference using the ADDRESS attribute. Users must ensure that representation of records and arrays are in the form expected by the called subprogram. The default link name is the symbolic representation of the simple name converted to lower case. For more information about C, see the *Harris C Reference Manual.*

For Fortran, all parameters are passed by reference. The parameter types must have the type ADDRESS defined in the package SYSTEM. The user is responsible for passing the arguments by reference, typically by using the ADDRESS attribute. For example:

```
c := add (a'address,b'address);
```

The result type for a Fortran function must be a scalar type. Care should be taken when using tasking and Fortran functions. Since Fortran is not reentrant, it is recommended that an Ada controller task be used to control concurrent access to Fortran functions. The default link name is the symbolic representation of the simple name converted to lower case with a trailing underscore (_) character. For more information about Fortran, see the *Hf77 Fortran Reference Manual.*

For Ada, the types and modes of parameters are unrestricted. The default link name is the symbolic representation of the simple name converted to lower case.

For other languages UNCHECKED, which allows all parameter types and modes, should be specified. The user must ensure that the called subprograms conform to the standard calling sequence for scalar and address data types. The user is responsible for forming alternative calling sequences when passing non-scalar or non-address arguments (e.g., passing a character string, etc.). The default link name is the symbolic representation of the simple name converted to lower case.

The link name of interface routines can be changed via the implementation-defined pragma EXTERNAL_NAME.

Care should be taken when using Ada tasking and interfacing to subprograms written in non-reentrant languages, such as Fortran. In such cases, an Ada controller task may be used to manage concurrent access to such routines.

Pragma INTERFACE_NAME

The implementation-defined pragma INTERFACE_NAME provides a method for specifying external names for variables, functions, and procedures. The required parameters are the simple name of the object and a string constant representing the external name. The pragma may not be applied to objects that have been explicitly initialized. For all practical purposes, this pragma is equivalent to the EXTERNAL_NAME pragma.

Pragma INTERFACE_OBJECT

The implementation-defined pragma INTERFACE_OBJECT provides an interface to objects defined externally from the Ada compilation model, or an object defined in another language. For example, a variable defined in the run-time system may be accessed via the pragma. This pragma has two required parameters, the first being the simple name of an Ada variable to be associated with the foreign object. The second parameter is a string constant that defines the link name of the object. The variable declaration must occur before the pragma and both must occur within the same declarative part or package specification.

An example is the C-language variable errno in the following code.

```
declare
    errno : integer ;
    pragma interface_object ( errno, "errno" ) ;
begin
    ...
end ;
```

Pragma INTERFACE_SHARED_OBJECT

The implementation-defined pragma INTERFACE_SHARED_OBJECT provides an interface to objects defined in other languages which exist in shared memory segments. Specifically, this allows for the sharing of data between Ada objects and Fortran or C objects defined within the same process or in a separate process.

Pragma INTERFACE_SHARED_OBJECT associates an Ada variable with a shared memory segment. It has two required parameters. The first parameter is the simple name of the Ada variable to be associated with the foreign object. The second parameter is a string constant that defines the external link name of the object as defined in the other language. The variable declaration must occur before the pragma and both must occur within the same declarative part or package specification.

Variables marked with the pragma must have a static size. It is recommended that an explicit length clause be specified for composite objects to ensure conformance with the size as defined by the other language. Additionally, record representation clauses may be used to define the layout of records to match the other language definitions.

The association of the shared memory segment to the Ada variable is effected at program start-up time, by the HAPSE run-time system. However, specific control over the config-

uration of the shared memory is defined externally from the Ada compilation model and requires user intervention. The Power UNIX **shmdefine** utility has been provided to aid the user in defining the configuration of shared memory segments. The utility produces a link-ready file and a loader command file which must be included in the link of any Ada program using pragma INTERFACE_SHARED_OBJECT. To include these files in the link process, the user should invoke the HAPSE prelinker, **a.ld**, adding the names of these files to the end of the command line. See Chapter 11 of this manual for an example application of the pragma. See also **shmdefine(1)**.

Pragma LINK_OPTION

The implementation-defined pragma LINK_OPTION allows a command to the **linker ld(1)** to be specified in an Ada compilation unit. The pragma has one required parameter, a string within quotes containing the command to be passed to **ld**. The string specified will be passed directly to **ld** by the **a.ld** tool. For example, if a compilation unit references the system **curses** library, the pragma:

```
pragma LINK_OPTION("-lcurses");
```

can be used in the compilation unit to link the **curses** library with the resulting Ada program. An example usage of this pragma may be found in the HAPSE **harrislib** library's MATH package. The body of this package contains a LINK_OPTION pragma that causes all programs that specify the package MATH in a with clause to be automatically linked with the system math library.

Pragma LIST

The implementation-dependent pragma LIST is implemented as described in Appendix B of the Ada RM.

Pragma MAP_FILE

The implementation-defined pragma MAP_FILE causes **a.ld** to automatically emit a map file at link time. See "Pragma MAP_FILE" on page 20-1 for a complete description.

Pragma MEMORY_POOL

The implementation-defined pragma MEMORY_POOL is used to change physical memory pool attributes from their default values for a memory pool. See "Memory Attributes" on page 20-11 and "Pragma MEMORY_POOL" on page 20-13 for a complete description.

Pragma MEMORY_SIZE

The implementation-dependent pragma MEMORY_SIZE is recognized by the implementation but has no visible effect. The implementation restricts the argument to the predefined value in the package SYSTEM.

Pragma OPT_FLAGS

The implementation-defined pragma OPT_FLAGS provides a method for overriding the optimization parameters defined by a HAPSE library's configuration. (See "Configuring HAPSE" on page 12-2). By specifying a configuration value for an optimizer parameter using this pragma, the given value is observed by the HAPSE compiler when the enclosing unit is compiled (regardless of the value specified for the optimizer parameter(s) in the library's configuration).

The OPT_FLAGS pragma takes a single string literal as an argument. This string, enclosed in quotes, should contain all of the optimizer flags to be overridden for the compilation, along with the value to be observed. The literal string argument must take the form:

"flag = value, flag = value, flag = value ..."

Seven flags are recognized by the HAPSE compiler. Many of these flags are described in detail in Chapter 12 of this manual and are configurable not only via the pragma, but also as parameters to the **a.config** tool. The optimizer flags are:

```
Objects_to_optimize
Loops_to_optimize
Unroll_limit
Growth_limit
Optimize_for_space
Opt_class
Noreorder
```

For example, the line:

```
Pragma OPT_FLAGS ("Growth_limit = 200, Unroll_limit = 5");
```

will optimize a total of 200 objects, and will use a loop unrolling limit of 5 for the compilation unit whose declarative part contains the preceding pragma. These values will override any values given by a local or system configuration record for the compilation.

Compilation units that do not contain this pragma will observe the optimizer flag values given by the library's configuration. Additionally, all optimizer flags that are not given as arguments when the pragma is utilized will use the values from the library's configuration.

Pragma OPT_LEVEL

The implementation-defined pragma OPT_LEVEL controls the level of optimization performed by the compiler. This pragma takes one of the following as an argument: MINI-

MAL, GLOBAL, or MAXIMAL. The default is MINIMAL (equivalent to **ada -O1**). The pragma is allowed within any declarative part. The specified optimization level will apply to all code generated for the specifications and bodies associated with the immediately closing declarative part. For more about optimization see Chapter 10 and Chapter 12.

Pragma OPTIMIZE

The implementation-dependent pragma OPTIMIZE is recognized by the implementation but does not have an effect in this release. See the **-O** option for **ada** for code optimization options, or the implementation-defined pragma OPT_LEVEL.

Pragma PACK

The implementation-dependent pragma PACK causes the compiler to choose a non-aligned representation for elements of composite types. Application of the pragma causes objects to be packed to the bit level.

Pragma PAGE

The implementation-dependent pragma PAGE is implemented as described in Appendix B of the Ada RM.

Pragma POOL_CACHE_MODE

The implementation-defined pragma POOL_CACHE_MODE defines the cache mode for a memory pool. See "Pragma POOL_CACHE_MODE" on page 20-22 for a complete description.

Pragma POOL_LOCK_STATE

The implementation-defined pragma `POOL_LOCK_STATE` defines the lock state of a memory pool. See "Pragma POOL_LOCK_STATE" on page 20-22 for a complete description.

Pragma POOL_PAD

The implementation-defined pragma `POOL_PAD` sets the pad for a STACK memory pool. See "Pragma POOL_PAD" on page 20-23 for a complete description.

Pragma POOL_SIZE

The implementation-defined pragma `POOL_SIZE` permits the setting of the size for a `STACK` or `COLLECTION` memory pool. See "Pragma POOL_SIZE" on page 20-22 for a complete description.

Pragma PRIORITY

The implementation-dependent pragma `PRIORITY` is implemented as described in Appendix B of the Ada RM. Priorities range from 0 to 255, with 255 being the most urgent. See "Pragma TASK_PRIORITY" on page 9-17 for a related pragma.

Pragma QUEUING_POLICY

The implementation-defined pragma `QUEUING_POLICY` sets the entry queuing policy. See "Pragma QUEUING_POLICY" on page 20-2 for a complete description.

Pragma RUNTIME_DIAGNOSTICS

The implementation-defined pragma `RUNTIME_DIAGNOSTICS` controls whether or not the run-time emits warning diagnostics. See "Pragma RUNTIME_DIAGNOSTICS" on page 20-1 for a complete description.

Pragma SERVER_CACHE_SIZE

The implementation-defined pragma `SERVER_CACHE_SIZE` sets the size of the server cache. See "Pragma SERVER_CACHE_SIZE" on page 20-2 for a complete description.

Pragma SHARED

The implementation-dependent pragma SHARED is recognized by the implementation but has no effect on the current release. See "Pragma VOLATILE" on page 9-18 for information about the implementation-defined pragma VOLATILE.

Pragma SHARED_PACKAGE

The implementation-defined pragma SHARED_PACKAGE provides for the sharing and communication of data declared within the specification of library-level packages. All variables declared in the specification of a package marked pragma SHARED_PACKAGE (henceforth referred to as a shared package) are allocated in shared memory that is created and maintained by the implementation (the variables declared in the body (if any) of such packages are not shared). The pragma can be applied only to library-level package specifications. Each package specification nested in a shared package will also be shared and all objects declared in the nested packages reside in the same shared memory segment as the outer package.

The implementation restricts the kinds of objects that can be declared in a shared package. Unconstrained or dynamically sized objects cannot be declared in a shared package and access type objects cannot be declared in a shared package. Additionally, generic instantiations cannot be declared within a shared package. If any of these restrictions are violated, a warning message is issued and the package is not shared. These restrictions apply to nested packages as well. Note that if a nested package violates one of the preceding restrictions, it prevents the sharing of all enclosing packages as well.

Packages that require initialization should not be marked with the pragma unless the user is prepared to deal with concurrency issues. The compiler no longer rejects the pragma in these cases; however, every program that uses the shared package will initialize it during program elaboration. Initialization can occur as a result of an explicit initialization by the user (e.g., `a : integer := 54 ;`) or implicitly due to an object's representation (an array or record with gaps). The compiler issues a warning message in either case.

Task objects are allowed within shared packages, however, the tasks as well as the data defined within those tasks are not shared.

Pragma SHARED_PACKAGE accepts as an optional argument, "*params*", that, if specified, must be a string constant containing a comma-separated list of system shared segment configuration parameters, as defined by the following:

`key=`*name*
: Identifies the system shared segment key to be used in subsequent `shmget` system calls, which are done automatically by the implementation in configuring the shared segment. *name* is considered to be a file name which will be translated to a shared segment key using the `ftok(3C)` service. By default, HAPSE applies "`key=`{*absolute HAPSE library path*}*/.shmem/package_name*" to the shared package and creates an empty file by that name. Note that relative path names may be specified and would cause key translation to be dependent on the user's current working directory when program execution is initiated. If *name* is a numeric literal (a decimal integer or Ada octal- or hexadecimal-based lit-

eral), HAPSE interprets this as the actual system key, and does not translate it using the `ftok` service.

`ipc=(IPC_CREAT, IPC_EXCL, IPC_PRIVATE)`
Allows the user to specify details about the initialization of the shared segment. By default, HAPSE applies `ipc=(IPC_CREAT)` to the shared package, thereby creating the shared segment if it did not previously exist. If any `ipc` parameters are given, they entirely replace the default `ipc` specification.

`SHM_RDONLY` Specifies that the segment is available only for `READ` operations. HAPSE defaults shared package segments to `READ/WRITE`.

CAUTION

The current shared memory implementation does not allow the use of the `'LOCK` and `'UNLOCK` attributes with a `SHM_RDONLY` shared memory segment. Any use of these attributes with a package marked `SHM_RDONLY` will raise `PROGRAM_ERROR` at run time.

`mode=`*n* Where *n* is assumed to be an octal number defining the access to the shared segment. By default, HAPSE applies `mode=644` to the shared package, (owner read/write, group read, other read). The specified value for mode is ORed into the *shmflgs* parameter that HAPSE uses for the **`shmget(2)`** call. Additional bits can be supplied via `mode` to control caching, etc. (e.g., "`mode = 8#200644#`" would specify `SHM_COPYBACK`, as well as the 644 mode). For more information, see the *Power UNIX Programming Guide*.

`SHM_LOCAL` Requests that pages for the shared segment be allocated from the local memory pool. If a program attempts to attach to a segment which has been allocated from local memory on a different CPU, then the attachment will fail. See **`shmget(2)`**.

`SHM_LOCK` Specifies that virtual memory pages be locked into physical memory at program start-up time. Doing this makes these pages immune to swapping.

`SHM_HARD` When used in conjunction with `SHM_LOCAL`, specifies that pages for the shared segment must be allocated from the local memory pool. If pages are not available from local memory then the signal `SIGSEGV` is delivered to the process. See **`shmget(2)`**.

`no_bsem` Prohibits the use of the shared package lock attributes `'LOCK` and `'UNLOCK`. In shared packages marked with this parameter, binary semaphore space is not initialized in the shared memory segment. Any attempt to make use of the lock attributes in a shared package marked with `no_bsem` will raise `PROGRAM_ERROR` at run time. Unlike `RDONLY` shared packages, packages marked by `no_bsem` have `READ/WRITE` capability.

`bind=n` Where *n* is assumed to be an octal number. The segment will be attached to the physical memory address specified by *n*.

WARNING

If the **shmbind(2)** attempt fails due to EBUSY or EREGSTALE, the implementation will ignore the error and continue, assuming that another program has already bound the segment to the desired location. Shared memory segments bound to physical memory should be freed manually by the user via **ipcrm(1)**.

CAUTION

By default, every shared package that is available for READ/WRITE has a binary semaphore initialized which starts 12 bytes before the end of the segment and extends to the end of the segment. If a shared package is bound to a device using the bind= parameter, be aware that the contents of these bytes may change if the 'LOCK and 'UNLOCK attributes are utilized. The only exceptions are those shared packages which are defined as SHM_RDONLY or those marked by the no_bsem parameter. In these cases, the semaphore space is not initialized, but it is still present.

A detailed explanation of the IPC and SHM flags, and access modes may be found in "Ada Language Program Communication" on page 11-4 of this manual and the following man pages: **shmbind(2)**, **shmget(2)**, **ipcs(1)**, **ipcrm(1)**, and **chmod(1)**.

The SHARED_PACKAGE pragma must appear within the specification of the library-level package. The pragma may also be repeated in the package body to allow the user to override the shared memory configuration parameters that were associated with the pragma in the specification. Additionally, these configuration parameters, as defined before, may also be specified at link time to **a.ld**, via the **-shmem** "*params*" option, where "*params*" is defined as before with the addition that the first item in the list must be the name of a shared package. If this option is used, then it replaces all previous information that may have been provided with all pragmas for that package.

A more detailed example of shared package usage is given in "Ada Language Program Communication" on page 11-4.

With the valid application of pragma SHARED_PACKAGE to a library-level package, the following assumptions can be made about the objects declared in the package:

- The lifetime of such objects is greater than the lifetime defined by the complete execution of a single program.
- The lifetime of such objects is guaranteed to extend from the elaboration of the shared package by the first *concurrent program* until the termination of execution of the last *concurrent program*.

In the preceding assumptions, a *concurrent program* is defined to be any Ada program that elaborates the body of a shared package, whose span of execution, from elaboration of such a package to termination, overlaps that of another such program.

In actuality, the shared memory segments created by these programs remain even after the last *concurrent program* has exited. The values of objects within these segments remain valid until the segment is destroyed, or until the system is rebooted. Segments may be explicitly removed through the shared memory service `shmctl`, to which an interface is provided in the HAPSE package `shared_memory_support`. Alternatively, the user may obtain information about active shared memory segments through the **`ipcs(1)`** utility. These segments may be removed via the **`ipcrm(1)`** utility.

Programs that attempt to reference the contents of objects declared in shared packages that have not been implicitly or explicitly initialized are technically erroneous as defined by the RM (3.2.1(18)). This implementation, however, does not prevent such references and, in fact expects them.

The preceding discussion describes the intent that several Ada programs may begin, continue, and complete their execution simultaneously, with the contents of the variables in the shared packages consistent with the execution of those programs.

The association of a system shared memory segment with the shared package occurs during the elaboration of the package body. If this association should fail due to system shared memory constraints, access, or improper use of shared memory configuration parameters, an error message is issued and the PROGRAM_ERROR exception is raised.

HAPSE has provided semaphore operations so that programs can define critical sections to reference and update variables within the shared packages. See "Implementation-Dependent Attributes" on page 9-18 for a description of the implementation-defined attributes `P'LOCK` and `P'UNLOCK`.

Example:

```
-- The following illustrates proper usage of the "params" argument
-- supplied to the Pragma SHARED_PACKAGE.

     package shared_pack is
          dummy : integer;
          procedure do_nothing;
          pragma shared_package ("key=3, ipc=IPC_CREAT, no_bsem, SHM_LOCK");
     end shared_pack;

     package body shared_pack is
          procedure do_nothing is
          begin
               null;
          end do_nothing;
     end shared_pack;
```

Pragma SHARE_BODY

The implementation-defined pragma SHARE_BODY is used to indicate whether or not an instantiation is to be shared. The pragma may reference the generic unit or the instantiated unit. When it references a generic unit, it sets sharing on/off for all instantiations of the

generic, unless overridden by specific SHARE_BODY pragmas for individual instantiations. When it references an instantiated unit, sharing is on/off only for that unit. The default is to share all generics that can be shared, unless the unit uses pragma INLINE. See "Shared Generics" on page 7-6 for a complete list of restrictions.

Pragma SHARE_BODY 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. The form of this pragma is

```
pragma SHARE_BODY (generic_name, boolean_literal)
```

Note that a parent instantiation is independent of any individual instantiation; therefore, recompilation 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.

Sharing generics causes a slight execution time penalty because all type attributes must be indirectly referenced (as if an extra calling argument were added). However, it substantially reduces compilation time in most circumstances and reduces program size.

Pragma SHARE_MODE

The implementation-defined pragma SHARE_MODE is provided to allow users to set the *share_mode* for a compilation unit from within the Ada source code. The format of the pragma is:

```
pragma SHARE_MODE (share_mode);
```

where *share_mode* is one of: SHARED, NON_SHARED, or BOTH.

This pragma is allowed immediately within the outermost declarative context of a library-level compilation unit. If applied to a specification, it also applies to the body (and any separate bodies) corresponding to that compilation unit. If applied to a body, it also applies to any separate bodies corresponding to that compilation unit, but not the specification. The pragma may be specified for the specification, body, and separate bodies of the same library-level package or subprogram.

If the pragma is applied to a generic, it also applies to all instantiations of that generic that exist in the same HAPSE library.

Pragma STORAGE_UNIT

The implementation-dependent pragma STORAGE_UNIT is recognized by the implementation but has no visible effect. The implementation restricts the argument to the predefined value in the package SYSTEM.

Pragma SUPPRESS

The implementation-dependent pragma SUPPRESS in the single parameter form is supported and applies from the point of occurrence to the end of the innermost enclosing block. DIVISION_CHECK and OVERFLOW_CHECK for floating-point types will reduce the amount of overhead associated with checking, but is not fully suppressible. The double parameter form of the pragma, with a name of an object, type, or subtype is recognized, but has no effect.

Pragma SUPPRESS_ALL

The implementation-defined pragma SUPPRESS_ALL gives permission to the implementation to suppress all run-time checks. Pragma SUPPRESS_ALL does not have any parameters. It is allowed to appear immediately within a declarative part or immediately within a package specification. Its effects are equivalent to a list of SUPPRESS pragmas, each naming a different check.

Pragma SYSTEM_NAME

The implementation-dependent pragma SYSTEM_NAME is recognized by the implementation but has no visible effect. The implementation provides only one enumeration value for SYSTEM_NAME in the package SYSTEM.

Pragma TASK_CPU_BIAS

The implementation-defined pragma TASK_CPU_BIAS is used to set the CPU assignments for a given BOUND task. See "Pragma TASK_CPU_BIAS" on page 20-8 for a complete description of task CPU assignments.

Pragma TASK_PRIORITY

The implementation-defined pragma TASK_PRIORITY is used to set the scheduling priority 1) for a given task within the server group and 2) for entry queuing. It also sets the operating system priority for BOUND tasks. See "Priorities" on page 18-3 and "Pragma TASK_PRIORITY" on page 20-7 for a complete description of task priorities.

Pragma TASK_QUANTUM

The implementation-defined pragma TASK_QUANTUM is used to set the task time-slice duration for a given task. See "Pragma TASK_QUANTUM" on page 20-9 for a complete description of task time-slicing.

Pragma TASK_WEIGHT

The implementation-defined pragma TASK_WEIGHT specifies the weight of a task. See "Pragma TASK_WEIGHT" on page 20-5 for a complete description.

Pragma VOLATILE

The implementation-defined pragma VOLATILE accepts a list of simple variable names, which the compiler assumes to occupy volatile storage bases. All accesses of these variables will result in memory references. Pragma VOLATILE should be used on any variable that may be accessed concurrently by different threads of a program, e.g., a variable shared between tasks.

An example of a use of pragma VOLATILE follows:

```
pragma VOLATILE ( name1, name2 );
```

Implementation-Dependent Attributes

HAPSE has defined the following attributes for use in conjunction with the implementation-defined pragma SHARED_PACKAGE:

```
P'KEY
P'SHM_ID
P'LOCK
P'UNLOCK
```

where the prefix P denotes a package marked with pragma SHARED_PACKAGE.

The 'KEY attribute is an overloaded function without parameters that returns the key used to identify the system shared segment associated with the package. One specification of the function returns the predefined type `string`, and returns a value specifying the file name used in the key translation (`ftok(3C)`). If an integer literal key was specified in the pragma SHARED_PACKAGE parameters, this function returns a null string. The other specification of the function returns the predefined type `universal_integer`, and returns a value specifying the translated integer key. The latter form of the function will raise the predefined exception PROGRAM_ERROR if the shared package body has not yet been elaborated.

The 'SHM_ID attribute is a function without parameters that returns the shared memory segment identifier for the system shared memory segment associated with the shared package P. This identifier corresponds to the identifier which is returned by the shared memory service `shmget` upon creation of the shared package.

The 'SHM_ID attribute will raise PROGRAM_ERROR if the call to `shmget` failed when the segment associated with the shared package P was created.

The 'LOCK and 'UNLOCK attributes are procedures without parameters that manipulate the "state" of a shared package. HAPSE defines all shared packages to have two states: LOCKed and UNLOCKed. Upon return from the 'LOCK procedure, the state of the package will be LOCKed. If upon invocation, 'LOCK finds the state already LOCKed, it will wait until it becomes UNLOCKed before altering the state and returning. 'UNLOCK sets the state of the package to UNLOCKed and then returns. At the point of unlocking the package, if another process waiting in the 'LOCK procedure has a more favorable operating system priority, the system will immediately schedule its execution.

Note that if 'LOCK is waiting, it may be interrupted by the HAPSE run-time system's time slice for tasks which may cause another task within the process to become active. Eventually, HAPSE will again transfer control to the 'LOCK procedure in the original task, and it will continue waiting or return to the task.

The state of the package is meaningful only to the 'LOCK and 'UNLOCK attribute procedures that set and query the state. A LOCK state does not prevent concurrent access to objects in the shared package. These attributes provide indivisible operations only for the setting and testing of implicit semaphores that could be used to control access to shared package objects.

CAUTION

The current shared memory implementation does not allow the use of the 'LOCK and 'UNLOCK attributes with a SHM_RDONLY shared memory segment or a shared package marked with the no_bsem parameter.

HAPSE provides the package, shared_memory_support. This package contains Ada type, subprogram definitions, and interfaces to aid the user in manually interfacing to the shared memory system services.

This includes:

- System defines and records layouts as defined by the C programming language include files, <sys/shm.h> and <sys/ipc.h>.
- Interface specifications to shared memory system calls: shmbind, shmget, shmat, shmctl, shmdt.
- Interface specifications to the system's binary semaphore operators: bsemget, bsemfree, lockbinsem, unlockbinsem, clockbinsem.

Specification of Package System

```
package system is
--
   type name is (HARRIS_POWER) ;

   system_name              : constant name := HARRIS_POWER ;
```

```
-- System-Dependent Named Numbers

min_int                 : constant := -2_147_483_647 - 1 ;
max_int                 : constant :=  2_147_483_647 ;
max_nonbinary_modulus   : constant :=  2_147_483_648 ;
max_binary_modulus      : constant :=  4_294_967_296 ;
max_digits              : constant :=  15 ;
max_mantissa            : constant :=  31 ;
fine_delta              : constant :=  2.0 ** (-31) ;
tick                    : constant :=  0.01 ;

-- Storage-related Declarations

type address is private ;

no_addr       : constant address ;

storage_unit : constant :=  8 ;

memory_size  : constant :=  3_221_225_469 ;

-- Other System-Dependent Declarations

subtype priority is integer range 0 .. 255 ;

subtype   max_rec_size : integer := 268435455;    -- 16#0FFFFFFF#

--
pragma suppress( elaboration_check );
--

--
-- The following functions provide conversions to "address"
-- from integer values.
--
-- Note: the function "physical_address" was misnamed, and will
-- be deleted in subsequent releases.  Use "logical_address".
--
function logical_address  (i : integer) return address ;
function physical_address (i : integer) return address renames
   logical_address ;

--
-- The following function should ONLY be used to supply a
-- machine address to an object's address clause statement.
--
-- The parameter is an integer describing the physical machine
-- address of the object.
--
-- Optimization of subsequent references to the object with the
-- address clause will result if the parameter to this function
-- is an integer literal.
--
-- The form of the function that takes a string argument is provided
-- to ease the specification of machine addresses which have the
-- high bit set.  For example,

--
--    for clock use at machine_address("16#96002000#") ;
--
-- In this case, the specified argument must be a string literal
-- which may have the high bit set, but is otherwise acceptable
-- to integer'value().
--
```

```
   function machine_address (i : integer) return address ;
   function machine_address (i : string) return address ;

   function ">"       (a, b : address) return boolean ;
   function "<"       (a, b : address) return boolean ;
   function ">="      (a, b : address) return boolean ;
   function "<="      (a, b : address) return boolean ;

   function "-"       (a, b : address) return integer ;
   function "+"       (a : address ; incr : integer) return address ;
   function "-"       (a : address ; decr : integer) return address ;

   function addr_gt   (a, b : address) return boolean renames ">" ;
   function addr_lt   (a, b : address) return boolean renames "<" ;
   function addr_ge   (a, b : address) return boolean renames ">=" ;
   function addr_le   (a, b : address) return boolean renames "<=" ;

   function addr_diff (a, b : address) return integer renames "-" ;
   function incr_addr (a : address ; incr : integer) return address
      renames "+" ;
   function decr_addr (a : address ; decr : integer) return address
      renames "-" ;
em ;
```

Restrictions on Representation Clauses

Pragma PACK

Pragma PACK is fully supported. Objects and components are packed to the nearest and smallest bit boundary when pragma PACK is applied.

Length Clauses

The specification T'SIZE is fully supported for all scalar and composite types, except for floating point. For floating-point, access, and task types, the supplied static expression must conform to an existing supported machine representation.

Type	Size
floating point	32 or 64
access	32
task	32

T'SIZE applied to a composite type will cause compression of scalar component types and the gaps between the components. T'SIZE applied to a composite type whose components are composite types does not imply compression of the inner composite objects. To achieve such compression, the implementation requires explicit application of T'SIZE or pragma PACK to the inner composite type.

Composite types which contain components that have had T'SIZE applied to them, will adhere to the specified component size, even if it causes alignment of components on non-STORAGE_UNIT boundaries.

The size of a non-component object of a type whose size has been adjusted, via T'SIZE or pragma PACK, will be exactly the specified size; however, the implementation will choose an alignment for such objects that provides optimal performance.

Record Representation Clauses

The simple expression following the keywords "at mod" in an alignment clause specifies the STORAGE_UNIT alignment restrictions for the record and must be one of the following values: 1, 2, 4 or 8.

The simple expression following the keyword "at" in a component clause specifies the STORAGE_UNIT (relative to the beginning of the record) at which the following range is applicable. The static range following the keyword range specifies the bit range of the component. Components may overlap word boundaries (4 STORAGE_UNITs). Components that are themselves composite types must be aligned on a STORAGE_UNIT boundary.

A component clause applied to a component that is a composite type does not imply compression of that component. For such component types, the implementation requires that T'SIZE or pragma PACK be applied, if compression beyond the default size is desired.

Address Clauses

Address clauses are supported only for variables, constants, and task entries.

For variables and constants, both logical and machine addresses are supported. A *logical address* refers to a virtual memory address in the execution program's address space. A *machine address* refers to a physical memory address.

Logical address clauses

- The function LOGICAL_ADDRESS is defined in the package SYSTEM to provide conversion from INTEGER values to ADDRESS values for logical addresses only.
- Both static and variable logical addresses are supported.
- The value supplied to the address clause must be a valid logical address in the user's program.

Machine address clauses

- When a machine address is desired, the expression supplied on the address clause must be an invocation of the function MACHINE_ADDRESS, found in package SYSTEM. Any other expression supplied to the address clause will cause it to be interpreted as a logical address.

- Both static and variable machine addresses are supported.
- If the argument to `MACHINE_ADDRESS` is an integer literal, then static address translation can occur, thereby removing any additional overhead involved in accessing the variable at run time.
- In order to use machine address clauses, you must have the `P_SHMBIND` privilege.

WARNING

It is the user's responsibility to ensure that the supplied address is a valid physical memory address.

Memory copies done through address clauses will require a bus access for each word.

Machine addresses clauses are implemented via system shared memory segments, which are bound to the specified physical memory address at elaboration time. These shared memory segments are removed at the end of normal execution of a program.

Interrupts

Interrupt entries (signals and hardware interrupts) are supported. This feature allows Ada programs to bind an operating system signal to an interrupt entry by using a `for` clause with a signal number. Two tasks may not posses a binding to the same interrupt source at the same time; if this is attempted, program_error will be raised. Interrupt entries must not have any parameters and can be called explicitly by the program. See the chapter on task interrupt entries in the run-time section for more information.

Other Representation Implementation-Dependencies

Restrictions to the ADDRESS Attribute

The `ADDRESS` attribute is not supported for the following entities: static constants, packages, tasks, and entries. Application of the attribute to these entities generates a compile time warning and returns a value of 0 at run time.

Restrictions to Alignment of Types

Like the size, the alignment of a data type is a property of the type. If a type requires alignment, then the address of any object of the type modulo the alignment value is required to

be zero. This is the same as saying that the address must be a multiple of the alignment value.

For example, if an object is word (4-byte) aligned, its address must be a multiple of 4 bytes. The address, 16#7fff439c# is a multiple of 4 bytes, so it is word-aligned. The address, 16#7fff439b# is not, however, because 16#b# mod 4 is 3.

Ideally, representation specifications and length clauses would size and align any data type to any number of bits. However, there are actually underlying hardware and compiler implementation choices that impose some restrictions. For the Series 6000 architecture, HAPSE imposes restrictions on the alignment of several specific types or classes of types. These restrictions are outlined in the following sections.

Access and Task Type Alignment

In order to provide the fastest access possible when dereferencing access types, HAPSE requires that all objects of access types be aligned on a word (4-byte) address boundary. Because HAPSE implements task types using an access type, it also requires that task types be aligned on a word boundary. The compiler issues an error message if you try to align a task or access type on any other alignment boundary.

FLOAT and LONG_FLOAT Type Alignment

In the current version of the HAPSE Ada compiler, the interface between the front end and code generator imposes an alignment restriction of at least byte alignment. For this reason, objects of any floating-point type and objects of composite types containing floating-point objects must be byte-aligned as a minimum. In order to provide more efficient code and avoid misaligned access exceptions on the Series 6000 architecture, the HAPSE Ada compiler will try to provide for optimal alignment of float and long float objects. The optimal alignment for single-precision floats is 4 bytes, and for long floats is 8 bytes. You may use pragma PACK or a representation specification to force the alignment to be as small as one-byte, but the compiler will reject any representation specification that tries to bit align a floating-point object.

Integer, Fixed-Point, and Enumeration Type Alignment

The HAPSE Ada compiler imposes no minimal alignment restrictions on any integer type, or any type whose underlying implementation is that of an integer, such as fixed-point and enumeration types. Objects of these types may be packed to the bit level resulting in bit alignment.

In order to maximize performance and avoid misaligned access exceptions on Series 6000 architectures, the HAPSE Ada compiler will try to align objects of integer, fixed-point, and enumeration types on their optimal alignment boundary. This boundary is a multiple of the size of the type in question. For example, integers will be given 4-byte optimal alignment and tiny_integer objects will be one-byte aligned.

The optimal alignment may be overridden with pragma PACK or by utilizing a representation specification to force smaller alignments.

Composite Type Alignment

The minimal and optimal alignments for record and array types is determined by taking the maximum of the alignment restrictions imposed by the component types involved. For example, if a record type contains an array of LONG_FLOAT objects, then the array and the enclosing record will be given an optimal alignment restriction of 64 bits. This may be reduced to a minimal alignment restriction of 8 bits by applying pragma PACK or a length clause to the array type.

Note that although array components may be packed to a size which is not a multiple of their alignment restriction, the HAPSE Ada compiler will force padding to be allocated after each component such that the next component begins on an alignment boundary. For example, consider the following types:

```
type BITTY_REC is
    record
        BOAT     : float ;
        SINKING : boolean ;
    end record ;

for BITTY_REC'size use 33 ;

type BR_ARRAY is array (1 .. 5) of BITTY_REC ;

X : BR_ARRAY ;
```

At first glance, it might seem that BR_ARRAY'size should be 165 (33 * 5) bits, but because the float element, BOAT, requires at least 8-bit alignment, BR_ARRAY will be sized to 200 (40 * 5) bits. This is because alignment padding is allocated between each element, as shown in Figure 9-1.

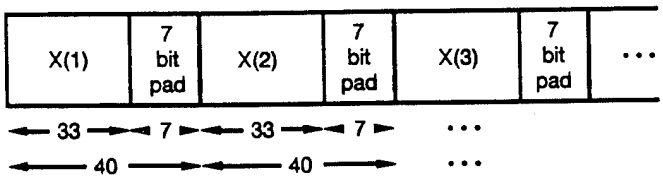


Figure 9-1. 8-Bit Alignment for Composite Type

If the BITTY_REC type did not have the length clause of 33 bits on it, this padding would occur differently, because the compiler would choose to optimally align each element of a BR_ARRAY. The float element of a BITTY_REC, BOAT, forces an optimal alignment of 32 bits, so each element of a BR_ARRAY would be aligned on a word boundary, as shown in Figure 9-2.

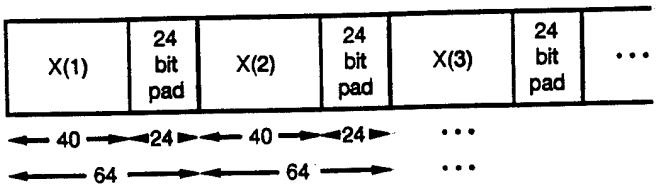


Figure 9-2. 32-Bit Alignment for Composite Type

The added pad between each element allows optimal code generation when accessing the float elements and avoids misaligned access exceptions as well.

Note that dynamic record types contain internal fields that may also increase the minimum alignment restrictions to 32 bits. In most cases, these types may not have representation specifications applied to them.

Conventions for Implementation-Generated Names

Implementation-generated names do not exist.

Unchecked Programming

Restrictions on UNCHECKED_CONVERSION

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

Implementation of UNCHECKED_CONVERSION

The following describes the transfer of data between the source and target operands when performing unchecked conversion. When possible, the implementation may optimize the conversion operation such that transfer of data does not actually occur.

Justification

The implementation considers all objects of simple types to be "right justified" within the storage allocated, and all objects of composite types to be "left justified". If, for alignment reasons, an object is placed in storage which is larger than the object's `'SIZE` value, the significant bits of an object of a simple type will be placed in the low order bits of storage,

right justified, with any padding in the high order bits. Likewise, should an object of a composite type be allocated storage which is larger than the type's `'SIZE`, the significant bits will be placed in the high order bits, and any padding will be placed in the low order bits.

Simple Type to Simple Type Conversions

For all access, task and scalar types, unchecked conversion is implemented using the most efficient MOVE instruction to move a 1, 2, 4 or 8 byte object to its destination. However, a `BIT_MOVE` will be used if the type has explicitly been given a `'SIZE` which is not a power of two.

If the sizes of the source and target differ, then the smallest size is used.

If the target has a larger size than the source, the source is moved to the low order bits of the target. If the target type is signed, then the high bit of the source is sign extended through the high bits of the target. Otherwise, the high order bits of the target are zero-filled.

If the target has a smaller size than the source, the low order bits of the source are copied to the target.

Composite Type to Composite Type Conversions

All conversions logically occur by moving bits from the source to the target, starting at the highest order bit of the source and target.

If the sizes of the source and target differ, then the smallest size is used.

If the target has a larger size than the source, the source is moved to the high order bits of the target, and the low order bits of the target are zero filled.

If the target has a smaller size than the source, the high order bits of the source are copied to the target.

Simple Type to Composite Type Conversions

Conversions from simple types to composite types are implemented by moving the low order, right justified, bits of the source to the high order, left justified, bits of the target.

If the sizes of the source and target differ, then the smallest size is used.

If the target has a larger size than the source, the source is moved to the high order bits of the target, and the low order bits of the target are zero-filled.

If the target has a smaller size than the source, the low order bits of the source are copied to the target.

Composite Type to Simple Type Conversions

Conversions from composite types to simple types are implemented by moving the high order, left justified, bits of the source to the low order, right justified, bits of the target.

If the sizes of the source and target differ, then the smallest size is used.

If the target has a larger size than the source, the source is moved to the low order bits of the target. If the target type is signed, then the high bit of the source is sign extended through the high bits of the target. Otherwise, the high order bits of the target are zero filled.

If the target has a smaller size than the source, the high order bits of the source are copied to the target.

UNCHECKED_DEALLOCATION

UNCHECKED_DEALLOCATION is supported. Currently, UNCHECKED_DEALLOCATION does not have an effect on access objects that designate tasks.

Implementation Characteristics of I/O Packages

Implementation-Dependent Characteristics of DIRECT I/O

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 as 268_435_455 storage units.

Implementation-Dependent Characteristics of SEQUENTIAL I/O

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 as 268_435_455 storage units.

Form Parameters

The HAPSE implementation of the standard Ada I/O packages support form parameters of the following syntax and semantics for the OPEN and CREATE subprograms:

form_parameters ::= [*form_specification* {, *form_specification*}]
form_specification ::= *form_name* => *form_value*

The following list defines the supported *form_name* and *form_values*:

`Append => True | False`

If `True`, the file is opened in append mode. If `False`, the file will be opened in truncate mode, which is the default.

`USE_ERROR` is raised if specified to the `CREATE` subprogram.

```
Owner => read | write | execute | read_write | ...
Group => read | write | execute | read_write | ...
Other => read | write | execute | read_write | ...
```

The file being created will have the permissions as defined by *form_name* and *form_value*. Note that *form_value* may be any combination of `read`, `write`, or `execute`, separated by an underscore (e.g., `write_read_execute`).

`USE_ERROR` is raised if specified to the `OPEN` subprogram.

`File_Descriptor => n`

This specifies that the high level `file_type` be associated with an existing open file descriptor, as specified by *n*. *n* should be of a form consistent with `integer'image`.

`USE_ERROR` is raised if specified to the `CREATE` subprogram.

`Page_Terminators => True | False`

If `False`, then page terminators are not output to the external file. If **`ASCII.FF`** is encountered while reading from the external file, it is interpreted as a character **`ASCII.FF`** and not as a page terminator. `USE_ERROR` will be raised upon explicit calls to `TEXT_IO.NEW_PAGE` or to `TEXT_IO.SET_LINE` when the current line number exceeds the specified argument.

If `True`, page termination on output will result in **`ASCII.FF`** being written to the external file. Encountering **`ASCII.FF`** on input is interpreted as a new page (e.g., `TEXT_IO.GET` would never see an **`ASCII.FF`** returned to it). True is the default.

`Terminal_Input => Lines | Characters`

If `Lines`, terminal input shall be done in canonical mode. This is the default.

If `Characters`, terminal input shall be done in non-canonical mode, such that the minimum input count is 1 character and the minimum input time is 0 seconds.

This form specification has no effect if the associated `file_type` is not used for terminal input.

`Echo => True | False`

If `True`, echoing of characters is done on input operations to the associated terminal device. This is the default.

If `False`, echoing of characters is not done on input operations to the associated terminal device for non-canonical processing. `USE_ERROR` is raised if the non-canonical processing has not been specified.

`File_Structure => Regular | Fifo`

If `Fifo`, then the file being created will be a named FIFO file. Otherwise, the file being created will be a regular file, which is the default.

`USE_ERROR` is raised if specified to the `OPEN` subprogram.

`Blocking => Tasks | Program`

If all the tasks in the running program have *task_weight* `BOUND`, then the *form_value* must be `Tasks`; otherwise, `USE_ERROR` is raised.

If all the tasks in the running program have *task_weight* `MULTIPLEXED`, then the *form_value* must be `Program`; otherwise, `USE_ERROR` is raised.

If the running program has tasks of both `BOUND` and `MULTIPLEXED` *task_weight*, then the *form_value* must be `Program`; otherwise, `USE_ERROR` is raised. This use of `Program` blocking behavior is intended to indicate that if a task blocks while performing I/O on the associated file, other tasks in the program may be blocked. The actual blocking behavior depends on the *task_weight* of a blocked task.

Machine-Code Insertions

WARNING

It is not recommended practice to inline machine-code procedures, as the compiler does not track register uses and definitions made by machine-code procedures. Inlining of machine-code procedures is supported, but the user should exercise caution.

PowerPC-604 (Series 6000)

The general definition of the package MACHINE_CODE provides an assembly language interface for the target machine including the record types needed in the code statement, an enumeration type containing all of 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) is the implementation defined attribute 'REF.

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

The general syntax for a machine-code statement is

code_n' (*opcode*, *operand* [, *operand*]);

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

```
code_3' (ai, r4, r4, +1);
```

The *opcode* must be an enumeration literal (i.e., it cannot be an object, an attribute, or a rename). Valid *opcodes* are listed in "PowerPC-604 Instruction Set" on page 9-31. An *operand* can only be an entity defined in MACHINE_CODE.

The 'REF attribute denotes the effective address of the first of the storage units allocated to the object. For a label, it refers to the address of the machine code associated with the corresponding body or statement. The attribute is of type OPERAND defined in the package MACHINE_CODE and is allowed only within a machine code procedure. 'REF is supported only for simple objects and labels.

PowerPC-604 Instruction Set

The MACHINE_CODE package supports the following PowerPC-604 opcodes:

add	add_dot	addc	addc_dot	addco
addco_dot	adde	adde_dot	addeo	addeo_dot
addi	addic	addic_dot	addis	addme
addme_dot	addmeo	addmeo_dot	addo	addo_dot
addze	addze_dot	addzeo	addzeo_dot	and_dot
and_r	andc	andc_dot	andi_dot	andis_dot
b	ba	bc	bca	bcctr
bcctrl	bcl	bcla	bclr	bclrl
bctr	bctrl	bdnz	bdnza	bdnzeq
bdnzeqa	bdnzeql	bdnzeqla	bdnzeqlr	bdnzeqlrl
bdnzf	bdnzfa	bdnzfl	bdnzfla	bdnzflr
bdnzflrl	bdnzge	bdnzgea	bdnzgel	bdnzgela

bdnzgelr	bdnzgelrl	bdnzgt	bdnzgta	bdnzgtl
bdnzgtla	bdnzgtlr	bdnzgtlrl	bdnzl	bdnzla
bdnzle	bdnzlea	bdnzlel	bdnzlela	bdnzlelr
bdnzlelrl	bdnzlr	bdnzlrl	bdnzlt	bdnzlta
bdnzltl	bdnzltla	bdnzltlr	bdnzltlrl	bdnzne
bdnznea	bdnznel	bdnznela	bdnznelr	bdnznelrl
bdnzng	bdnzngа	bdnzngl	bdnzngla	bdnzngir
bdnzngirl	bdnznl	bdnznla	bdnznll	bdnznlla
bdnznllr	bdnznllrl	bdnzns	bdnznsa	bdnznsl
bdnznsla	bdnznslr	bdnznslrl	bdnznu	bdnznua
bdnznul	bdnznula	bdnznulr	bdnznulrl	bdnzso
bdnzsoa	bdnzsol	bdnzsola	bdnzsolr	bdnzsolrl
bdnzt	bdnzta	bdnztl	bdnztla	bdnztlr
bdnztlrl	bdnzun	bdnzuna	bdnzunl	bdnzunla
bdnzunlr	bdnzunlrl	bdz	bdza	bdzeq
bdzeqa	bdzeql	bdzeqla	bdzeqlr	bdzeqlrl
bdzf	bdzfa	bdzfl	bdzfla	bdzflr
bdzflrl	bdzge	bdzgea	bdzgel	bdzgela
bdzgelr	bdzgelrl	bdzgt	bdzgta	bdzgtl
bdzgtla	bdzgtlr	bdzgtlrl	bdzl	bdzla
bdzle	bdzlea	bdzlel	bdzlela	bdzlelr
bdzlelrl	bdzlr	bdzlrl	bdzlt	bdzlta
bdzltl	bdzltla	bdzltlr	bdzltlrl	bdzne
bdznea	bdznel	bdznela	bdznelr	bdznelrl
bdzng	bdzngа	bdzngl	bdzngla	bdzngir
bdzngirl	bdznl	bdznla	bdznll	bdznlla
bdznllr	bdznllrl	bdzns	bdznsa	bdznsl
bdznsla	bdznslr	bdznslrl	bdznu	bdznua
bdznul	bdznula	bdznulr	bdznulrl	bdzso
bdzsoa	bdzsol	bdzsola	bdzsolr	bdzsolrl
bdzt	bdzta	bdztl	bdztla	bdztlr
bdztlrl	bdzun	bdzuna	bdzunl	bdzunla
bdzunlr	bdzunlrl	beq	beqa	beqctr
beqctrl	beql	beqla	beqlr	beqlrl
bf	bfa	bfctr	bfctrl	bfl
bfla	bflr	bflrl	bge	bgea

bgectr	bgectrl	bgel	bgela	bgelr
bgelrl	bgt	bgta	bgtctr	bgtctrl
bgtl	bgtla	bgtlr	bgtlrl	bl
bla	ble	blea	blectr	blectrl
blel	blela	blelr	blelrl	blr
blrl	blt	blta	bltctr	bltctrl
bltl	bltla	bltlr	bltlrl	bne
bnea	bnectr	bnectrl	bnel	bnela
bnelr	bnelrl	bng	bnga	bngctr
bngctrl	bngl	bngla	bnglr	bnglrl
bnl	bnla	bnlctr	bnlctrl	bnll
bnlla	bnllr	bnllrl	bns	bnsa
bnsctr	bnsctrl	bnsl	bnsla	bnslr
bnslrl	bnu	bnua	bnuctr	bnuctrl
bnul	bnula	bnulr	bnulrl	bso
bsoa	bsoctr	bsoctrl	bsol	bsola
bsolr	bsolrl	bt	bta	btctr
btctrl	btl	btla	btlr	btlrl
bun	buna	bunctr	bunctrl	bunl
bunla	bunlr	bunlrl	clrlslwi	clrlslwi_dot
clrlwi	clrlwi_dot	clrrwi	clrrwi_dot	cmp
cmpi	cmpl	cmpli	cmplw	cmplwi
cmpw	cmpwi	cntlzw	cntlzw_dot	crand
crandc	crclr	creqv	crmove	crnand
crnor	crnot	cror	crorc	crset
crxor	dcbf	dcbi	dcbst	dcbt
dcbtst	dcbz	divw	divw_dot	divwo
divwo_dot	divwu	divwu_dot	divwuo	divwuo_dot
eciwx	ecowx	eieio	eqv	eqv_dot
extlwi	extlwi_dot	extrwi	extrwi_dot	extsb
extsb_dot	extsh	extsh_dot	fabs	fabs_dot
fadd	fadd_dot	fadds	fadds_dot	fcmpo
fcmpu	fctiw	fctiw_dot	fctiwz	fctiwz_dot
fdiv	fdiv_dot	fdivs	fdivs_dot	fmadd
fmadd_dot	fmadds	fmadds_dot	fmr	fmr_dot
fmsub	fmsub_dot	fmsubs	fmsubs_dot	fmul

fmul_dot	fmuls	fmuls_dot	fnabs	fnabs_dot
fneg	fneg_dot	fnmadd	fnmadd_dot	fnmadds
fnmadds_dot	fnmsub	fnmsub_dot	fnmsubs	fnmsubs_dot
fres	fres_dot	frsp	frsp_dot	frsqrte
frsqrte_dot	fsel	fsel_dot	fsqrt	fsqrt_dot
fsqrts	fsqrts_dot	fsub	fsub_dot	fsubs
fsubs_dot	icbi	inslwi	inslwi_dot	insrwi
insrwi_dot	isync	lbz	lbzu	lbzux
lbzx	lfd	lfdu	lfdux	lfdx
lfs	lfsu	lfsux	lfsx	lha
lhau	lhaux	lhax	lhbrx	lhz
lhzu	lhzux	lhzx	li	lis
lmw	lswi	lswx	lwarx	lwbrx
lwz	lwzu	lwzux	lwzx	mcrf
mcrfs	mcrxr	mfcr	mfctr	mfdar
mfdbatl	mfdbatu	mfdec	mfdsisr	mfear
mffs	mffs_dot	mfibatl	mfibatu	mflr
mfmsr	mfpvr	mfsdr1	mfspr	mfsprg
mfsr	mfsrin	mfsrr0	mfsrr1	mftb
mftbl	mftbu	mfxer	mr	mtcrf
mtctr	mtdar	mtdbatl	mtdbatu	mtdec
mtdsisr	mtear	mtfsb0	mtfsb0_dot	mtfsb1
mtfsb1_dot	mtfsf	mtfsf_dot	mtfsfi	mtfsfi_dot
mtibatl	mtibatu	mtlr	mtmsr	mtsdr1
mtspr	mtsprg	mtsr	mtsrin	mtsrr0
mtsrr1	mttb	mttbu	mtxer	mulhw
mulhw_dot	mulhwu	mulhwu_dot	mulli	mullw
mullw_dot	mullwo	mullwo_dot	nand	nand_dot
neg	neg_dot	nego	nego_dot	nop
nor	nor_dot	not_r	not_dot	or_dot
or_r	orc	orc_dot	ori	oris
rfi	rlwimi	rlwimi_dot	rlwinm	rlwinm_dot
rlwnm	rlwnm_dot	rotlw	rotlw_dot	rotlwi
rotlwi_dot	rotrwi	rotrwi_dot	sc	slw
slw_dot	slwi	slwi_dot	sraw	sraw_dot
srawi	srawi_dot	srw	srw_dot	srwi

srwi_dot	stb	stbu	stbux	stbx
stfd	stfdu	stfdux	stfdx	stfiwx
stfs	stfsu	stfsux	stfsx	sth
sthbrx	sthu	sthux	sthx	stmw
stswi	stswx	stw	stwbrx	stwcx_dot
stwu	stwux	stwx	sub	sub_dot
subc	subc_dot	subco	subco_dot	subf
subf_dot	subfc	subfc_dot	subfco	subfco_dot
subfe	subfe_dot	subfeo	subfeo_dot	subfic
subfme	subfme_dot	subfmeo	subfmeo_dot	subfo
subfo_dot	subfze	subfze_dot	subfzeo	subfzeo_dot
subi	subic	subic_dot	subis	subo
subo_dot	sync	tlbie	tlbiex	tlbsync
trap	tw	tweq	tweqi	twge
twgei	twgt	twgti	twi	twle
twlei	twlge	twlgei	twlgt	twlgti
twlle	twllei	twllt	twllti	twlng
twlngi	twlnl	twlnli	twlt	twlti
twne	twnei	twng	twngi	twnl
twnli	xor_dot	xor_r	xori	xoris

Register Set

Registers - The full set of 32 general-purpose registers for the.PowerPC-604 architecture is supported (R0 through R31), plus a full set of 32 floating-point registers (F0 through F31), 8 control registers (CRF0 through CRF7), 16 segment registers (SR0 through SR15), and 44 special-purpose registers (XER, LR, CTR, DSISR, DAR, DEC, SDR1, SRR0, SRR1, SPRG0, SPRG1, SPRG2, SPRG3, ASR, EAR, TB, TBU, PVR, IBAT0U, IBAT0L, IBAT1U, IBAT1L, IBAT2U, IBAT2L, IBAT3U, IBA,T3L, DBAT0U, DBAT0L, DBAT1U, DBAT1L, DBAT2U, DBAT2L, DBAT2L, DBAT3U, DBAT3L, MMC,R0, PMC1, PMC2, SIA, SDA, HID0, IABR, DABR, PIR).

Addressing Modes

All of the PowerPC-604 addressing modes are supported by the compiler. They are accessed through the following functions provided in MACHINE_CODE.

Address Mode	Assembler Notation	Ada Function Call
External Name	lo16(*name*)	ext_lo (<*name*>)
	hi16(*name*)	ext_hi (<*name*>)
	uhi16(*name*)	ext_uhi(<*name*>)
	label	lo14(<name>)
	label	lo24(<name>)
Absolute	lo16(*xxx*)	absol_lo (<*disp*>)
	hi16(*xxx*)	absol_hi (<*disp*>)
	uhi16(*xxx*)	absol_uhi(<*disp*>)
	I14	lo14(<disp>)
	I24	lo24(<disp>)
Register Indirect	0(R*n*)	indr (<*addr_reg*>)
with Displacement	I16(R*n*)	disp (<*reg*>, <*disp*>)
with External	lo16(*name*)(R*n*)	lo16 (<*name*>, <addr_reg>)
Immediate Data	D, SI, UI	"+" (<*integer*>)
	SI	"-" (<*integer*>)
	SI	immed(<*integer*>)

Usage

The following example uses machine code to move a block of data.

```
with machine_code;
with system;

procedure move (dest, src : in system.address; length : in positive) is
--
     use machine code;
     pragma implicit_code(off); -- See Section 9.3.4
     pragma opt_flags ("noreorder");
--
begin
--
-- PowerPC-604 input arguments set up as:
-- r3 <= dest; r4 <= src; r5 <= length;
--
     -- save CTR in r8 and restore it before returning
     code_2' (mfspr, r8, CTR) ;
     code_3' (andil_dot, r7, r5, +3) ;

     <<backward_unaligned>>
```

```
      code_3' (sriq, r6, r5, immed(2));-- calculate number of words
      code_1' (mtctr, r6);             -- put # of words in CTR
      -- check for partial word
      <<bu_partial>>
         code_3' (cmpi, crf0, r7, +0);
         code_2' (ble, crf0, bu_loop'ref);
         -- move partial word
         code_2' (lbzu, r6, disp(r4, -1));
         code_2' (stbu, r6, disp(r3, -1));
         code_3' (ai, r7, r7, -1);
         code_1' (b, bu_partial'ref);
      <<bu_loop>>
         code_3' (lsi, r6, r4, +4);
         code_3' (ai, r4, r4, -4);
         code_3' (stsi, r6, r3, +4);
         code_3' (ai, r3, r3, -4);
         code_1' (bdn, bu_loop'ref);   -- CTR -= 1; b if /= 0

      <<done>>
      code_1' (mtctr, r8) ;            -- restore CTR
      code_0' (op => br);              -- Return.

--
end move;
```

NOTE

The `IMPLICIT_CODE` pragma is used to optimize the code by eliminating the stack frame and the copy of the `IN` parameters. The `br` instruction is necessary when the `IMPLICIT_CODE` pragma is utilized to act as the `return` statement.

20
Configuration Control

Overview

Although Chapter 9 discusses all pragmas, it focuses on pragmas that influence the software development environment, compiling, and linking. This chapter discusses pragmas that affect configuration of the whole run-time system, task execution, and memory utilization. It also provides some information about the underlying implementation of tasking and memory resources.

General Pragmas

The following pragmas affect the run-time system as a whole.

Pragma RUNTIME_DIAGNOSTICS

The implementation-defined pragma RUNTIME_DIAGNOSTICS controls whether or not the run-time emits warning diagnostics.

```
pragma RUNTIME_DIAGNOSTICS (boolean) ;
```

boolean A static boolean enumeration literal. **TRUE** means run-time diagnostics will be emitted. **FALSE** means run-time diagnostics will not be emitted. The default is **TRUE**. At run-time, you can specify this value via a call to RUNTIME_CONFIGURATION.SET_RUNTIME_DIAGNOSTICS.

Pragma MAP_FILE

The implementation-defined pragma MAP_FILE may occur in any declarative part. It causes **a.ld** to automatically emit at link time a map file containing an ASCII description of pragma entries and comments that define the layout of the file. This file is useful in conjunction with the **a.map** tool defined in Chapter 5. Similar functionality is available via **a.ld -map**. If this pragma is absent, then no map file will be produced.

```
pragma MAP_FILE (file_name) ;
```

file_name A static string of non-zero length specifying the name of the map file.

Pragma QUEUING_POLICY

The implementation-defined pragma QUEUING_POLICY sets the entry queuing policy.

```
pragma QUEUING_POLICY (policy_identifier) ;
```

policy_identifier The keyword **FIFO_QUEUING** means the entry queuing policy as defined in the Ada RM sections 9.5.3 and 9.7.1. **PRIORITY_QUEUING** means the entry queuing policy as defined in Ada 95 RM section D.4. The default is **FIFO_QUEUING**.

Pragma SERVER_CACHE_SIZE

The implementation-defined pragma SERVER_CACHE_SIZE sets the size of the server cache. The server cache contains execution servers that are currently unneeded by the application, but which can be placed back into service when they become necessary. These include the anonymous servers for terminated BOUND tasks, as well as servers from server groups which were reduced in size by the run-time.

```
pragma SERVER_CACHE_SIZE (cache_size) ;
```

cache_size A static, non-negative number specifying the maximum number of servers allowed in the cache (i.e., the server cache size). The default is 8. This value can also be set at run-time via a call to RUNTIME_CONFIGURATION.SET_SERVER_CACHE_SIZE.

Pragma DEFAULT_HARDNESS

The implementation-defined pragma DEFAULT_HARDNESS sets the default hardness for any memory bound to LOCAL via a pragma MEMORY_POOL. It can be overridden by individual MEMORY_POOL pragmas. See "Pragma MEMORY_POOL" on page 20-13.

```
pragma DEFAULT_HARDNESS (hardness) ;
```

hardness The keyword **HARD** or **SOFT** specifying the default hardness for memory. The default is acquired from the environment at program start-up time.

Note that when a Series 6000 system is configured without GLOBAL memory, the operating systems treats LOCAL memory as if it were GLOBAL. In such cases, user requests to bind virtual memory to LOCAL memory will be rejected since the system considers it GLOBAL.

Task and Group Configuration Concepts

Task Names and Default Settings

To make good use of task pragmas, it is necessary to understand some terminology.

ENVIRONMENT task
: At start-up, the run-time creates this one task that performs library-level package elaboration and executes the main program.

DEFAULT pseudo task
: This non-executing pseudo task sometimes provides default task-attribute values for other tasks. The user may change these default values with task pragmas or with calls to routines in package `RUNTIME_CONFIGURATION`.

ghost task
: An automatically generated overhead task.

For any actual task (excluding objects of task types) or ADMIN or TIMER ghost task, if a configuration pragma is omitted for that task, the value specified for the DEFAULT pseudo task is used instead.

For objects of task types, the following steps indicate the search order for configuration pragma values.

1. If the object is a variable and the pragma exists for that variable, that pragma is used.
2. If the pragma exists for its task type, that pragma is used.
3. If the task type is a derived type, the pragma of the nearest ancestor type is used if found.
4. If no such pragma is found, the DEFAULT pseudo task is checked for the pragma, and that pragma is used if found.
5. If no pragma has been found, the default value is used.

The same steps take place simultaneously for SHADOW, COURIER, INTR_COURIER, and PROXY ghost tasks.

Task Specifiers in Task Pragmas

The following task specifiers appear in task pragmas..

task_specifier ::= {*ordinary_task* | *ghost_task* | `ENVIRONMENT` | `SPEC`}

ordinary_task ::= {*task_type_name* | *task_variable_name* | `DEFAULT`}

ghost_task ::= {*companion_ghost_task* | `ADMIN` | `TIMER`}

companion_ghost_task ::= {*shadow_ghost* | *courier_ghost* | *intr_courier_ghost* | *proxy_ghost*}

shadow_ghost ::= *ordinary_task*, **SHADOW**, *task_entry*

courier_ghost ::= *ordinary_task*, **COURIER**, *task_entry*

intr_courier_ghost ::= *ordinary_task*, **INTR_COURIER**, *task_entry*

proxy_ghost ::= *ordinary_task*, **PROXY**

task_entry ::= {*entry_name* | **DEFAULT**}

task_specifier Value	Effect
SPEC or (omitted)	The pragma must occur in the declarative part of a task specification. It then applies to all tasks identified with that specification.
task type	The pragma applies to all task objects of that task type, regardless of where the task objects are actually declared, unless overridden for derived types or task variables.
task variable	The pragma must appear in the same declarative part as the declaration of the task variable. In this case, the pragma affects the task attribute of the specified task, regardless of any other task pragmas associated with defaults or task specifications.
task	The pragma applies only to the task itself.
DEFAULT	The pragma sets the task attribute to the specified value for the DEFAULT pseudo task, and therefore for all tasks, unless otherwise specified for a task.
ENVIRONMENT	The pragma sets the task attribute to the specified value for the ENVIRONMENT task.
ghost task	The pragma is associated with the specified ghost task. Ghost task specifiers are described in Chapter 17.

Group Names and Default Settings

To make good use of group pragmas, it is necessary to understand some terminology.

server group

Server groups allow users to restrict the resources their tasks use. These groups are designated by simple identifiers and are defined when they are used. However, they are not Ada program entities. They cannot be referenced anywhere except within the appropriate pragmas. In fact, they exist in a namespace which is separate from the Ada language's namespaces. This separate namespace is completely flat. That is, there is no hierarchical nesting to the namespace based on the units in which these pragmas appear. The same group name can be specified in two separate and unrelated units, and it will indicate the same group.

PREDEFINED group
: At start-up, the run-time creates this one predefined group that includes and executes the ENVIRONMENT task. By default, the DEFAULT pseudo task is also in this group.

DEFAULT group
: This pseudo group provides default group-attribute values for other groups that omit any group configuration pragmas. The user may change these default values with group pragmas or with calls to routines in package `RUNTIME_CONFIGURATION`.

To add tasks to any group, see "Pragma TASK_WEIGHT" on page 20-5.

Group Specifiers in Group Pragmas

The following server group specifiers appear in group pragmas.

group_spec ::= {**DEFAULT** | **PREDEFINED** | *group_name*}

group_spec Value	Effect
DEFAULT	The pragma sets the group attribute for the DEFAULT pseudo group, and therefore for all groups, to the specified value.
PREDEFINED	The pragma sets the group attribute for the PREDEFINED group to the specified value.
group	The pragma applies only to the group itself.

Task Attributes

Users can control the execution of tasks: specifically, tasks' scheduling priority, time-slice duration, physical CPU binding, and weight. Control may be static through implementation-defined pragmas, and may be changed dynamically via supplied routines in the `RUNTIME_CONFIGURATION` package.

Pragma TASK_WEIGHT

The implementation-defined pragma TASK_WEIGHT specifies the weight of a task.

?The weight of a task specifies whether the task is bound to a server or if it, along with some set of other tasks, is served by a group of servers.

```
pragma TASK_WEIGHT (weight[, task_specifier ]) ;
```

weight	::= { *bound_spec* | *multiplexed_spec* | *passive_spec* }
bound_spec	::= **BOUND**
multiplexed_spec	::= **MULTIPLEXED**, *group_spec*
passive_spec	::= **PASSIVE**, *passivity*
passivity	::= {*general* | *server* | *ipl_server*}
general	::= **GENERAL**
server	::= **SERVER**
ipl_server	::= **IPL_SERVER**, *ipl_value*

There are three possible weights: BOUND, MULTIPLEXED, and PASSIVE.

BOUND tasks are served by an anonymous group, distinct from all other groups, containing a single server.

MULTIPLEXED tasks are served by named groups, specified by *group_spec*, and are associated with multiple servers. These server groups are configured via other pragmas. (See "Group Attributes" on page 20-10.)

PASSIVE tasks do not have designated servers on which to execute. They do not execute at all unless called via an entry call. When they are called, the server executing the caller switches to execute the PASSIVE task and continues executing the passive task until it blocks waiting for another entry call. It then resumes execution of the caller. For passive tasks, the passivity must also be specified.

BOUND tasks can be thought of as special-cases of the MULTIPLEXED case. The sequence:

```
pragma TASK_WEIGHT (BOUND, t);
```

is roughly equivalent to:

```
pragma GROUP_SERVERS (1, anon_group_spec);
pragma TASK_WEIGHT (MULTIPLEXED, anon_group_spec, t);
```

The weight of default tasks can be overridden by certain options to the `a.ld` tool.

`a.ld -bound` overrides as:

```
pragma TASK_WEIGHT (BOUND, DEFAULT);
```

`a.ld -multiplexed` overrides as:

```
pragma TASK_WEIGHT(MULTIPLEXED, PREDEFINED, DEFAULT);
pragma GROUP_SERVERS(1, PREDEFINED);
```

In the absence of any such option to `a.ld`, by default, the following pragmas apply:

```
pragma TASK_WEIGHT(MULTIPLEXED, PREDEFINED, DEFAULT);
```

In other words, by default, the task weight for all tasks, including the environment task, is MULTIPLEXED.

This pragma will not be accepted for any ghost tasks. SHADOW ghost tasks have no associated weight. COURIER, INTR_COURIER, and PROXY ghost tasks are always BOUND. The ADMIN and TIMER ghost tasks, if they exist, are always BOUND.

For information about group specifiers, see "Group Specifiers in Group Pragmas" on page 20-5. For information about task specifiers, see "Task Specifiers in Task Pragmas" on page 20-3. For information about GENERAL, IPL, and IPL_SERVER tasks, see "Passive Tasks" on page 18-5.

Pragma TASK_PRIORITY

As discussed in the chapters relating to "task scheduling", the *task scheduling priority* of a task determines how the Ada run-time selects tasks for execution within a group. Similarly, the *operating system scheduling priority* determines how the real-time kernel selects task groups for execution.

The implementation-defined pragma `TASK_PRIORITY` is primarily used to set the *task scheduling priority*. For a BOUND task, it also sets the *operating system scheduling priority* of the BOUND task's anonymous group. See "Task Names and Default Settings" on page 20-2 to find out how a task without an explicit pragma `TASK_PRIORITY` setting gets its scheduling priority.

```
pragma TASK_PRIORITY (scheduling_priority [, task_specifier ]) ;
```

scheduling_priority (

A required integer expression, possibly a program variable, specifying the scheduling priority. It is in the range `0..MAX_PRIORITY`, as defined by the package `RUNTIME_CONFIGURATION`. This value can also be set at run-time via a call to `RUNTIME_CONFIGURATION.SET_TASK_PRIORITY`.

Values greater than `MAX_PRIORITY` will be truncated to `MAX_PRIORITY` by the run-time executive.

Values less than 0 are considered to be values relative to `MAX_PRIORITY+1`. The following pragmas are equivalent:

```
pragma TASK_PRIORITY
    (RUNTIME_CONFIGURATION.MAX_PRIORITY) ;
pragma TASK_PRIORITY (-1) ;
```

As previously mentioned, for a BOUND task, this pragma sets the *operating system scheduling priority* of the BOUND task's anonymous group. The sequence:

```
pragma TASK_WEIGHT (BOUND, t) ;
pragma TASK_PRIORITY (prio, t) ;
```

is equivalent to:

```
pragma GROUP_SERVERS (1, anon_group_spec) ;
```

```
pragma TASK_WEIGHT (MULTIPLEXED, anon_group_spec, t) ;
pragma GROUP_PRIORITY (prio, anon_group_spec) ;
pragma TASK_PRIORITY (prio, t) ;
```

Unless otherwise specified, the default value of *scheduling_priority* for the DEFAULT pseudo task is RUNTIME_CONFIGURATION.PRIORITY_OF_ENVIRONMENT. Similarly, unless otherwise specified, the default value of *scheduling_priority* for the ADMIN ghost task is also the priority of the environment.

Unless otherwise specified, the default value of *scheduling_priority* for the other ghost tasks is as follows:

```
pragma TASK_PRIORITY(DEFAULT, SHADOW, DEFAULT, -1);
pragma TASK_PRIORITY(DEFAULT, COURIER, DEFAULT, -1);
pragma TASK_PRIORITY(DEFAULT, INTR_COURIER, DEFAULT, -1);
pragma TASK_PRIORITY(DEFAULT, PROXY, -1);
pragma TASK_PRIORITY(TIMER, -1);
```

For information about task specifiers, see "Task Specifiers in Task Pragmas" on page 20-3.

Pragma TASK_CPU_BIAS

By default, tasks are automatically distributed across all available CPUs on the system. However, applications are also provided precise control over task distribution with the concept of the CPU bias.

A CPU bias is a mask in which the relative bit number identifies a CPU # (LSB corresponds to CPU #0). For example:

CPU Bias	Effect
2#00000100#	Task will be bound to CPU #2
2#01000010#	Task allowed to execute on CPUs #6 & #1
2#11111111#	Task allowed to execute on all 8 CPUs

Note that when more than 1 bit is set in a CPU bias, the kernel will continually employ CPU load-balancing techniques and migrate the task to the least busy CPU specified in the bias.

If the application is utilizing physical "local memory" pools, the kernel's load-balancing algorithms will not automatically migrate tasks to CPUs without direct access to those physical pools. However, explicit task CPU bias specifications will allow such migration. See "Pragma MEMORY_POOL" on page 20-13 for information about the implementation-defined pragma, MEMORY_POOL, and physical "local memory" utilization.

NOTE

Specifying a CPU bias of zero forces the task to use the "default task bias" as described below.

See **cpu_bias(2)** for more information on CPU biases.

The implementation-defined pragma TASK_CPU_BIAS provides for the binding of BOUND tasks to individual CPUs or a set of CPUs, associating a CPU bias with one or more BOUND tasks. This is necessary because pragma GROUP_CPU_BIAS is not available for BOUND tasks. See "Task Names and Default Settings" on page 20-2 to find out how a task without an explicit pragma TASK_CPU_BIAS setting gets its CPU bias.

```
pragma TASK_CPU_BIAS (cpu_bias [, task_specifier ]) ;
```

cpu_bias A required CPU bias, possibly a program variable, specifying CPUs that are valid for the machine configuration where the application will run. This value can also be set at run-time via a call to RUNTIME_CONFIGURATION.SET_TASK_CPU_BIAS.

By default, the CPU bias for all tasks, including the environment task, is inherited from the bias of the process that spawned the environment task. Normally this bias specifies all CPUs on the system. See the **run(1)** system command for more details.

The sequence:

```
pragma TASK_WEIGHT (BOUND, t) ;
pragma TASK_CPU_BIAS (bias, t) ;
```

is equivalent to:

```
pragma GROUP_SERVERS (1, anon_group_spec) ;
pragma TASK_WEIGHT (MULTIPLEXED, anon_group_spec, t) ;
pragma GROUP_CPU_BIAS (bias, anon_group_spec) ;
```

For information about task specifiers, see "Task Specifiers in Task Pragmas" on page 20-3.

Pragma TASK_QUANTUM

Apart from activation, rendezvous, abort, delay, termination, and priority, task scheduling is also dependent on its time slice. A task's time slice is determined by its quantum attribute. A *quantum* is the length of time a task actually spends executing on a CPU. As a task executes, its quantum is decremented by the kernel based on the 60Hz system clock interrupt. The quantum is reset when it is decremented to zero. At this time, another task may be scheduled to run if its quantum is lower. When a task is preempted due to interrupts or higher priority tasks, its quantum is not reset. When a task blocks for some other reason (rendezvous, for example) its quantum is reset when it becomes able to run.

The implementation-defined pragma, TASK_QUANTUM, provides for the specification of a task's quantum value. See "Task Names and Default Settings" on page 20-2 to find out how a task without an explicit pragma TASK_QUANTUM setting gets its CPU quantum. If the default task quantum for the DEFAULT pseudo task has not been explicitly set via a TASK_QUANTUM pragma, then the task gets the value 6.

```
pragma TASK_QUANTUM (quantum [, task_specifier ] ) ;
```

quantum A non-zero number, possibly a program variable, of 60Hz clock ticks. The value can be set at run time via a call to RUNTIME_CONFIGURATION.SET_TASK_QUANTUM.

By default, the quantum value for all tasks is inherited from the quantum of the process that spawned the environment task. Normally this value is the system-wide default of 6. See the `run(1)` system command for more details.

For information about task specifiers, see "Task Specifiers in Task Pragmas" on page 20-3.

Group Attributes

Users can control the operating system scheduling priority, physical CPU binding, and number of servers in a group. Control may be static through implementation-defined pragmas or through the run-time configuration package, and may be changed dynamically via supplied routines that interface to the run-time executive.

Pragma GROUP_PRIORITY

The implementation-defined pragma GROUP_PRIORITY specifies the *operating-system scheduling priority* of all the servers in a given group. It does not specify the *task scheduling priority* of particular tasks within the group. If this pragma is not specified for a particular group, the group acquires the *operating-system scheduling priority* of the environment that spawned it.

```
pragma GROUP_PRIORITY (scheduling_priority, group_spec) ;
```

scheduling_priority

A static integer expression specifying the operating system scheduling priority. It is in the range 0..MAX_PRIORITY, as defined by the package RUNTIME_CONFIGURATION. This value can also be set at run-time via a call to RUNTIME_CONFIGURATION.SET_GROUP_PRIORITY.

Values greater than MAX_PRIORITY will be truncated to MAX_PRIORITY by the run-time executive.

Values less than 0 are considered to be values relative to MAX_PRIORITY+1.

For information about group specifiers, see "Group Specifiers in Group Pragmas" on page 20-5 .

Pragma GROUP_CPU_BIAS

The implementation-defined pragma GROUP_CPU_BIAS specifies the CPU bias for all the servers in a given group. If this pragma is not specified for a particular group, the default bias is acquired from the environment, which indicates any CPUs.

```
pragma GROUP_CPU_BIAS (cpu_bias, group_spec) ;
```

cpu_bias A static CPU bias specifying CPUs that are valid for the machine configuration where the application will run. At run time, a call to RUNTIME_CONFIGURATION.SET_GROUP_CPU_BIAS can be used to set this value.

For information about group specifiers, see "Group Specifiers in Group Pragmas" on page 20-5.

Pragma GROUP_SERVERS

The implementation-defined pragma GROUP_SERVERS controls the number of servers for a particular group, including the PREDEFINED group.

```
pragma group_servers (group_size, group_spec) ;
```

group_size A static non-negative number indicating the quantity of servers in a group. If, for any group, no GROUP_SERVERS pragma is specified, then the default size for that group is 1. At run time, a call to RUNTIME_CONFIGURATION.SET_GROUP_SERVERS can be used to set this value.

For information about group specifiers, see "Group Specifiers in Group Pragmas" on page 20-5.

Specification of RUNTIME_CONFIGURATION Package

?TODD

Memory Attributes

On Series 6000 systems there are two kinds of physical memory pools:

- Global memory (1 pool)
- Local memory (up to 4 pools, 1 per CPU board)

An example configuration of a 6-processor Series 6000 is shown in Figure 20-1.

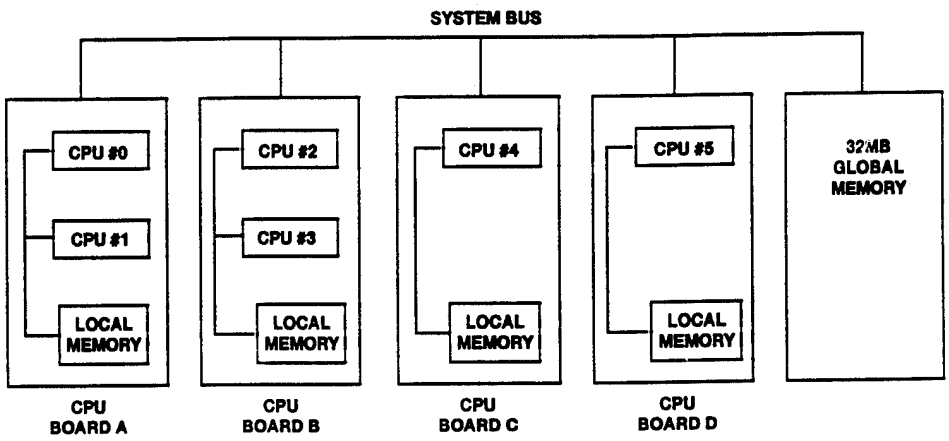


Figure 19-1. Example Configuration for 6-Processor Series 6000 System

Global memory is available to all CPUs via a system-wide bus. Local memory is available to CPUs via a local bus physically located on the same CPU board as the local memory. Accessing local memory from a foreign board CPU is allowed but is extremely costly and should be prevented in all time-critical areas. With the judicious use of pragma `MEMORY_POOL`, `TASK_CPU_BIAS`, and `GROUP_CPU_BIAS`, an Ada application can take full advantage of all the CPU and memory resources of a Series 6000 system.

WARNING

The `lwarx` and `stwcx.` instructions do not guarantee indivisibility of operations when foreign local memory is involved; do not attempt such operations with data structures bound to foreign local memory.

NOTE

The HAPSE package `INDIVISIBLE_OPERATIONS` in `harrislib` utilizes the `lwarx` and `stwcx.` instructions. Do not attempt such operations with data structures bound to foreign local memory.

Combining local memory usage with hardware interrupt handling can cause erroneous program behavior in isolated cases. All such cases are outlined in "Local Memory and Hardware Interrupts" on page 21-8 dealing with hardware interrupt entries.

Pool Specifiers

The following memory pool specifiers appear in memory pool pragmas.

pool_spec	::= {*text_pool* | *stack_pool* | *data_pool* | *collection_pool* | *default_pool*}
sizeable_spec	::= {*stack_pool* | *collection_pool*}
paddable_spec	::= {*stack_pool*}
default_pool	::= **DEFAULT**
text_pool	::= **TEXT**
stack_pool	::= **STACK**, {*task_specifier*}
data_pool	::= **DATA**, {**PKG** | **DEFAULT**}
collection_pool	::= **COLLECTION**, {**DEFAULT** | *access_type*}

pool_spec — This parameter specifies the particular pool of memory whose parameters are to be altered.

DEFAULT — This value means the *memory_spec* is applied to all memory in the program for which a specific memory pool was not already specified.

TEXT — For the entire text image (machine instructions), specify a value.

STACK — For a specific task, an object of a task type, the environment task, or the DEFAULT pseudo task, the stack may be allocated out of dynamic pools bound to local or global memory.

In this form the pragma may occur in any declarative part. If the second parameter is **ENVIRONMENT**, then the pragma affects the environment task's stack. If the second parameter is **SPEC**, then the pragma must be immediately enclosed by a task specification and will affect all associated tasks. If the second parameter is **DEFAULT**, the pragma applies to all stack frames for all tasks not marked with their own explicit pragma MEMORY_POOL specification. If the second parameter is not any of these three keywords, then it must be the name of a task type or a task variable in the same declarative part.

DATA — For the static memory associated with a specific package, or for all other packages, specify a value.

In this form the pragma must occur in the immediate declarative part of a library-level package specification, a library-level package body, or a library-level subprogram. If the second parameter is **PKG**, it must occur in the package specification or body. When in a package specification, the

pragma affects all static data for the package specification and for the package body, unless another pragma is applied to the body. When in a package body, the pragma affects all static data for the package body, regardless of any pragmas associated with the package specification. When the second parameter is **DEFAULT**, the pragma affects all other static data.

COLLECTION For the memory associated with an access type with a `'storage_size` clause, specify a value. When a pragma is applied to a COLLECTION, that collection is allocated from heap memory and can never be deallocated. It is recommended that this be done only in library-level packages.

In this form the pragma must occur in the same declarative part as the specification of the supplied *access_type*. The *access_type* must have a `'storage_size` length clause associated with it before the pragma is encountered.

Pragma MEMORY_POOL

The implementation-defined pragma MEMORY_POOL is used to change physical memory pool attributes from their default values for a memory pool. The pragma affects the mapping of abstract memory to physical memory, specifically: static data (packages), procedural data (stacks), and pointer data (collections).

```
pragma MEMORY_POOL (pool_spec, memory_spec) ;
```

memory_spec	::= {*global_spec* | *local_spec*}
global_spec	::= **GLOBAL**
local_spec	::= **LOCAL**, *mp_cpu_bias* [, *hardness*]
mp_cpu_bias	::= {*cpu_bias* | **HOME**}
hardness	::= {**HARD** | **SOFT**}

pool_spec See "Pool Specifiers" on page 20-13 for more information.

memory_spec specifies new values for memory pool attributes. If the MEMORY_POOL pragma is not specified for a particular pool (or for the DEFAULT pool), the default value for the *memory_spec* for that pool depends on the machine configuration on which it runs. For a system with a single CPU board, its value is the value specified by the environment. For a system with multiple CPU boards, its value is GLOBAL.

GLOBAL Uses physical global memory. Note that when a Series 6000 system is configured without GLOBAL memory, the operating systems treats LOCAL memory as if it were GLOBAL. In such cases, user requests to bind virtual

memory to LOCAL memory will be rejected since the system considers it GLOBAL.

`LOCAL` — Uses physical local memory.

cpu_bias — Identifies which physical local memory pool to utilize. Distinct physical local memory pools are identified by specifying a CPU bias which contains a (partial) list of CPU numbers corresponding to a CPU board. A CPU bias is a mask in which the relative bit number identifies a CPU # (LSB corresponds to CPU #0). Refer to Figure 20-3. Note that the *cpu_bias* must specify at least one CPU (cannot be zero). The `CPU_BIAS` is used to locate a CPU board's local memory pool.

The *cpu_bias* is searched starting with the LSB (least significant bit) and the first CPU specified by the bias determines which CPU board is selected.

For example, assume that a user provides a *cpu_bias* with bits that specified CPUs existing on two different CPU boards. In that case, the CPU board selected would be the board that holds the lowest numbered CPU.

`HOME` — Allocates the memory pool from the LOCAL memory associated with the CPU on which the appropriate task is running. For TEXT and DATA memory pools, the appropriate task is the ENVIRONMENT task and the allocation occurs before the ENVIRONMENT task executes any Ada code. For COLLECTION memory pools, the appropriate task is the task that elaborates the access type associated with the memory pool and the allocation occurs at the time of that elaboration. For STACK memory pools, the appropriate task is the one that will be using the stack during its execution, and the allocation occurs when that task is created. In any of these cases, if a task migrates to another CPU after the allocation occurs, the memory will not also migrate.

hardness — Controls usage of physical global memory if insufficient physical local memory is available. The default value for an unspecified hardness is defined by the `DEFAULT_HARDNESS` pragma.

Figure 20-2 depicts how memory mapping might occur given a specific configuration and applications of pragma `MEMORY_POOL`.

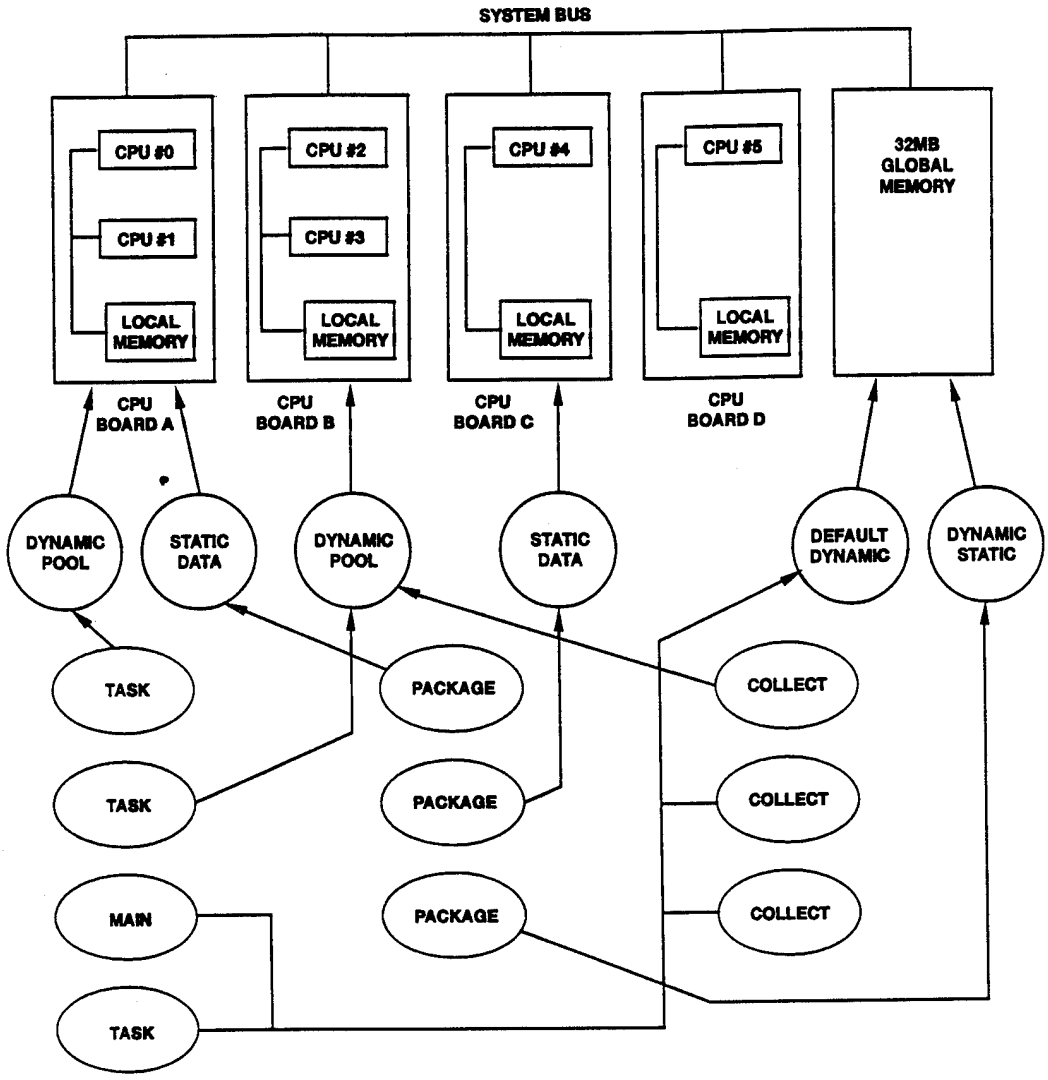


Figure 19-2. Memory Usage on a 6-Processor Series 6000 System

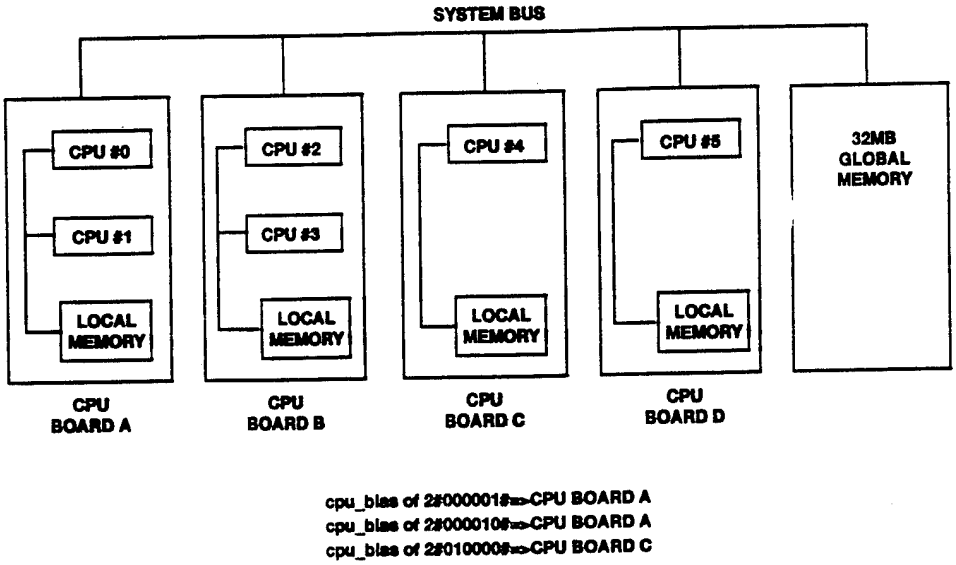


Figure 19-3. Relationship of cpu_bias to CPU Board on a 6-Processor Series 6000 System

For memory pools associated with physical local memory, the pool will be bound to the physical local memory associated with the highest logical CPU number found in the supplied *cpu_bias*. Since it is possible to specify a *cpu_bias* referencing multiple CPUs not sharing the same physical local memory, the preceding algorithm can be used to predict the results. Consider the following example:

```
 1 package defaults is
 2    pragma memory_pool (data, default, global) ;
 3    pragma memory_pool (stack, environment, global) ;
 4    pragma pool_size (stack, environment, 1024*1024) ;
 5    pragma pool_size (stack, default, 2048) ;
 6    pragma pool_size (collection, default, 20*1024*1024) ;
 7    pragma memory_pool (collection, default, global) ;
 8 end defaults ;
 9
10 package aero is
11    ...
12    pragma memory_pool (data, pkg, local, 2#010000#, hard, ncache) ;
13
14    task type aeronautics is
15       pragma task_weight (bound, spec) ;
16       pragma task_cpu_bias (2#010000#) ;
17       pragma memory_pool (stack, spec, local, 2#010000#) ;
18    end aeronautics ;
19    for aeronautics'storage_size use 4096*8 ;
20 end aero ;
21
22 package eom is
23    task equations is
24       ...
25       pragma task_weight (bound, spec) ;
26       pragma task_cpu_bias (2#000001#) ;
27    end equations ;
28 end eom ;
```

In the preceding example, a program including all three packages would have these attributes, assuming the machine configuration shown in Figure 20-2:

- Static data is allocated out of global memory (line 2)
- The environment task's stack frame allocated out of global memory (line 3)
- The size of the environment task's stack frame is 1 MB (line 4)
- The default task stack frame size is 2 KB (line 5)
- The size of the default heap is 20 MB (line 6)
- The default heap is allocated as global memory (line 7)
- Static data for package aero (both for specification and body) is allocated to local memory on CPU board C (line 12) (configuration-dependent)
- The weight of tasks of type aeronautics is BOUND (line 15)
- Tasks of type aeronautics execute on CPU #4 (line 16)
- Stacks for tasks of type aeronautics are allocated out of local memory on CPU board C (line 17) (configuration-dependent)
- The stack size of tasks of type aeronautics is 4 KB (line 19)
- The weight of task equations is BOUND (line 25)
- Task equations executes on CPU #0 (line 26)

Pragma POOL_CACHE_MODE

The implementation-defined pragma POOL_CACHE_MODE defines the cache mode for a memory pool.

```
pragma POOL_CACHE_MODE (pool_spec, cache_mode) ;
```

pool_spec	See "Pool Specifiers" on page 20-13 for more information.
cache_mode	The keyword **COPYBACK** or **NCACHE**. **COPYBACK** means use the operating system's COPYBACK cache mode. **NCACHE** means use the operating system's NCACHE cache mode. The default is the value specified for the DEFAULT pool. If there is no DEFAULT pool, this parameter value is **COPYBACK**.

The optional *cache_mode* sets the specified system cache attribute on the associated memory pool (see the system service **memadvise(2)** for more information).

In COPYBACK cache mode, only a single task is usually modifying a semi-private data area at any given point in time and other tasks will not read the update immediately. This mode does not cause a cache flush or memory bus access until another CPU reads the data.

Pragma POOL_LOCK_STATE

The implementation-defined pragma POOL_LOCK_STATE defines the lock state of a memory pool.

```
pragma POOL_LOCK_STATE (pool_spec, lock_state) ;
```

pool_spec — See "Pool Specifiers" on page 20-13 for more information.

lock_state — The keyword **LOCKED** or **UNLOCKED**. **LOCKED** means the memory pages are physically locked in memory and cannot be swapped out by the operating system. **UNLOCKED** means the memory pages can be swapped out by the operating system. The default is the value specified for the DEFAULT pool. It there is no DEFAULT pool, the default is **UNLOCKED**.

If a program specifies that text is to be locked in memory and local memory is specified by the user via this pragma, then task migrations to foreign CPU boards will be inhibited. Locking occurs when:

- FAST_INTERRUPT_TASKs are present
- plock (TXT_LOCK) system call is present

WARNING

When locking pages in memory and using local memory pools and using hardware interrupt entries, the user must specify additional information to the hardware interrupt entry address clause. See "Local Memory and Hardware Interrupts" on page 21-8.

Pragma POOL_SIZE

The implementation-defined pragma POOL_SIZE permits the setting of the size for a STACK or COLLECTION memory pool.

```
pragma POOL_SIZE (sizeable_spec, size_spec) ;
```

size_spec ::= {*size* | **UNLIMITED**}

sizeable_spec — See "Pool Specifiers" on page 20-13 for more information.

size — A static non-negative number that controls the amount of space allocated for an Ada program's use.

UNLIMITED — A value that is allowed only for the COLLECTION, DEFAULT and STACK, ENVIRONMENT memory pools.

This pragma, if specified for the STACK, DEFAULT pool, will not affect the size of the STACK, ENVIRONMENT pool. This is the only pragma where such a statement is true. The implementation is this way so that the stack size for the ENVIRONMENT task can con-

tinue to be **UNLIMITED**, which is its default value. This value can always be overridden explicitly, though.

If this pragma is not specified for the ENVIRONMENT task's STACK pool, the default value is **UNLIMITED**. If this pragma is not specified for a task type's STACK pool, the default value is the task type's 'storage_size value if it exists, and 20480 otherwise. If no POOL_SIZE pragma is valid for a task object or a task other than the ENVIRONMENT task, the default value for that real task is 20480. The default values for ghost tasks are as follows:

Table 20-1. Stack Pool Sizes for Ghost Tasks

Shadow Type	Default Stack Size
SHADOW	N/A
COURIER	10240
INTR_COURIER	4096
PROXY	10240
ADMIN	12800
TIMER	12800

If this pragma is unspecified for the COLLECTION, DEFAULT pool, its value is **UNLIMITED**. If this pragma is unspecified for any other COLLECTION pool, then its default value is the value of the 'storage_size attribute for the collection.

Pragma POOL_PAD

The implementation-defined pragma POOL_PAD sets the pad for a STACK memory pool.

```
pragma POOL_PAD (paddable_spec, size) ;

paddable_spec ::= {stack_pool}
```

size A non-negative number that controls the amount of additional pad after the stack size. This value has no meaning when the stack size for the same pool is **UNLIMITED**.

This additional space is intended only for use by the run-time system or for signal handlers. For ADMIN ghost tasks, the default is 12800; otherwise, it is 4096.