REPORT DOCUM AD-A246 475

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to ave... existing data sources gathering and maintaining the data needed, and reviewing the collection of information. Send comments regar... ...ding suggestions for reducing this burden, to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, ... Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 07 Nov 1991 to 01 Jun 1993

4. TITLE AND SUBTITLE
Hewlett-Packard, HP 9000 Series 700/800 Ada Compiler, Version 5.35, HP 9000 Series 800 Model 835 (Host & Target), 911107W1.11228

5. FUNDING NUMBERS
(2)

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

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

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-517-1291

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

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

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

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Hewlett-Packard, HP 9000 Series 700/800 Ada Compiler, Version 5.35, Wright-Patterson AFB, HP 9000 Series 800 Model 835 (Host & Target), ACVC 1.11.

DTIC
S ELECTE D
FEB 26 1992
D

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

15. NUMBER OF PAGES

16. PRICE CODE

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

NSN 7540-01-280-550

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

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 7 November 1991.

Compiler Name and Version: HP 9000 Series 700/800 Ada Compiler Version 5.35

Host Computer System: HP 9000 Series 800 Model 835 HP-UX, Version A.B8.00 (release 8.00)

Target Computer System: HP 9000 Series 800 Model 835 HP-UX, Version A.B8.00 (release 8.00)

Customer Agreement Number: 91-09-04-HPC

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

As a result of this validation effort, Validation Certificate 911107W1.11228 is awarded to Hewlett-Packard. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

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

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

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

Accesion For	
NTIS CRA&I	
DTIC TAB	
Unannounced	
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

92 2 24 002

92-04666

AVF Control Number: AVF-VSR-517-1291
Date VSR complete: 9 December 1991
91-09-04-HPC

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 911107W1.11228
Hewlett-Packard
HP 9000 Series 700/800 Ada Compiler, Version 5.35
HP 9000 Series 800 Model 835 => HP 9000 Series 800 Model 835

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

DECLARATION OF CONFORMANCE

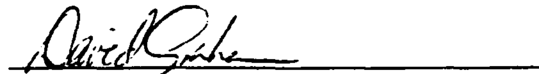
The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer:	Hewlett-Packard Company, California Language Lab
Ada Validation Facility:	ASD/SCEL, Wright Patterson AFB, OH 45433-6503
ACVC Version:	1.11
Ada Implementation:	
Ada Compiler Name and Version:	HP 9000 Series 700/800 Ada Compiler Version 5.35
Host Computer System:	HP 9000 Series 800 Model 835 HP-UX, Version A.B8.00 (release 8.00)
Target Computer System:	HP 9000 Series 800 Model 835 HP-UX, Version A.B8.00 (release 8.00)

Declaration:

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



Date: 10/30/91

Hewlett-Packard Company
David Graham
Ada R&D Section Manager

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 USE OF THIS VALIDATION SUMMARY REPORT 1-1
1.2 REFERENCES. 1-2
1.3 ACVC TEST CLASSES 1-2
1.4 DEFINITION OF TERMS 1-3

CHAPTER 2 IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS 2-1
2.2 INAPPLICABLE TESTS. 2-2
2.3 TEST MODIFICATIONS. 2-4

CHAPTER 3 PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT 3-1
3.2 SUMMARY OF TEST RESULTS 3-1
3.3 TEST EXECUTION. 3-2

APPENDIX A MACRO PARAMETERS

APPENDIX B COMPILATION SYSTEM OPTIONS

APPENDIX C APPENDIX F OF THE Ada STANDARD

CHAPTER 1

INTRODUCTION

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

1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

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

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

1.2 REFERENCES

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

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

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

1.3 ACVC TEST CLASSES

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

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

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

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

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

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

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

1.4 DEFINITION OF TERMS

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

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

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 2 August 1991.

E28005C B28006C C32203A C34006D C35508I C35508J
C35508M C35508N C35702A C35702B B41308B C43004A
C45114A C45346A C45612A C45612B C45612C C45651A
C46022A B49008A B49008B 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) C46012I..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.

C45536A, C46013B, C46031B, C46033B, and C46034B contain length clauses that specify values for 'SMALL that are not powers of two or ten; this implementation does not support such values for 'SMALL.

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.

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

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

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.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

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
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_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
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

The tests listed in the following table check the given file operations for the given combination of mode and access method; this implementation does not support these operations.

Test	File Operation	Mode	File Access Method
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

CE2401H, EE2401D, and EE2401G use instantiations of DIRECT_IO with unconstrained array and record types; this implementation raises USE_ERROR on the attempt to create a file of such types.

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.

CE3202A expects that function NAME can be applied to the standard input and output files; in this implementation these files have no names, and USE_ERROR is raised. (See section 2.3.)

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 23 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.

B23004A	B24007A	B24009A	B28003A	B32202A	B32202B
B32202C	B37004A	B61012A	B74401F	B74401R	B91004A
B95032A	B95069A	B95069B	BA1101B	BC2001D	BC3009C

B85002A was graded passed by Evaluation Modification as directed by the AVO. This test declares a record type REC2 whose sole component is of an unconstrained record type with a size in excess of 2**32 bytes; this implementation rejects the declaration of REC2. Although a strict interpretation of the LRM requires that this type declaration be accepted (an exception may be raised on the elaboration of the type or an object declaration), the AVO accepted this behavior in consideration that such early error detection is expected to be allowed by the revised language standard.

BA2001E was graded passed by Evaluation Modification as directed by the AVO. The test expects that duplicate names of subunits with a common ancestor will be detected as compilation errors; this implementation detects the errors at link time, and the AVO ruled that this behavior is acceptable.

EA3004D was graded passed by Evaluation and Processing Modification as directed by the AVO. The test requires that either pragma INLINE is obeyed for a function call in each of three contexts and that thus three library units are made obsolete by the re-compilation of the inlined function's body, or else the pragma is ignored completely. This implementation obeys the pragma except when the call is within the package specification. When the test's files are processed in the given order, only two units are made obsolete; thus, the expected error at line 27 of file EA3004D6M is not valid and is not flagged. To confirm that indeed the pragma is not obeyed in this one case, the test was also processed with the files re-ordered so that the re-compilation follows only the package declaration (and thus the other library units will not be made obsolete, as they are compiled later); a "NOT APPLICABLE" result was produced, as expected. The revised order of files was 0-1-4-5-2-3-6.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-10 value as 'SMALL for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal 'SMALLs may be omitted.

CE3202A was graded inapplicable by Evaluation Modification as directed by the AVO. This test applies function NAME to the standard input file, which in this implementation has no name; USE_ERROR is raised but not handled, so the test is aborted. The AVO ruled that this behavior is acceptable pending any resolution of the issue by the ARG.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

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

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

Marianne Mardesich
California Language Lab
19447 Pruneridge Avenue
Cupertino CA 95014
(408) 447-6973

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

3.2 SUMMARY OF TEST RESULTS —

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

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

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	3791
b) Total Number of Withdrawn Tests	95
c) Processed Inapplicable Tests	83
d) Non-Processed I/O Tests	0
e) Non-Processed Floating-Point Precision Tests	201
f) Total Number of Inapplicable Tests	284
g) Total Number of Tests for ACVC 1.11	4170

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 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 computer system, as appropriate. The results were captured on the computer system.

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

Switch	Effect
-B	Produces an output listing.
-e 999	Sets the maximum number of errors to 999.
-W b, -T	Suppresses procedure traceback when exceptions are not caught.

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	255
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"

MACRO PARAMETERS

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

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

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	2147483647
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	HP9000_PA_RISC
$DELTA_DOC	2#1.0#e-31
$ENTRY_ADDRESS	ENTRY_ADDR
$ENTRY_ADDRESS1	ENTRY_ADDR1
$ENTRY_ADDRESS2	ENTRY_ADDR2
$FIELD_LAST	255
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.00
$GREATER_THAN_DURATION_BASE_LAST	100_000_000.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E38

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
$HIGH_PRIORITY	16
$ILLEGAL_EXTERNAL_FILE_NAME1	not_there//not_there/*^
$ILLEGAL_EXTERNAL_FILE_NAME2	not_there/not_there/not_there/././not_there///
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006D1.ADA")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	C
$LESS_THAN_DURATION	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST	-100_000_000.0
$LINE_TERMINATOR	' '
$LOW_PRIORITY	1
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NO_SUCH_TYPE
$MANTISSA_DOC	31
$MAX_DIGITS	15
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648
$MIN_INT	-2147483648
$NAME	SHORT_SHORT_INTEGER

$NAME_LIST	HP9000_PA_RISC
$NAME_SPECIFICATION1	/ACVC/mnt/root5/notrace/Test/ACVC1B/Run/X2120A
$NAME_SPECIFICATION2	/ACVC/mnt/root5/notrace/Test/ACVC1B/Run/X2120B
$NAME_SPECIFICATION3	/ACVC/mnt/root5/notrace/Test/ACVC1B/Run/X3114A
$NEG_BASED_INT	16#FF_FF_FF_FD#
$NEW_MEM_SIZE	1048576
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	HP9000_PA_RISC
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	RECORD NULL; END RECORD;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	32768
$TICK	0.010
$VARIABLE_ADDRESS	VARIABLE_ADDR;
$VARIABLE_ADDRESS1	VARIABLE_ADDR1;
$VARIABLE_ADDRESS2	VARIABLE_ADDR2;
$YOUR_PRAGMA	EXPORT_OBJECT

APPENDIX B

COMPILATION SYSTEM OPTIONS

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

NAME
ada - Ada compiler

SYNOPSIS
ada [options] [files] libraryname

Remarks:
This command requires installation of optional Ada software (not included with the standard HP-UX operating system) before it can be used.

DESCRIPTION
ada is the HP-UX Ada compiler. It accepts several types of file arguments:

(1) Arguments whose names end with .ad?, where ? is any single alphanumeric character, are taken to be Ada source files.

(2) The libraryname argument names an Ada library that must have been previously created using the ada.mklib(1) command. An Ada library is an HP-UX directory containing files that are used by the various components of the Ada compilation system. There is no required or standard suffix on the name of an Ada library.

The named source files are compiled and each successfully compiled unit is placed in the specified Ada library by the compiler. When binding, or binding and linking, information is extracted from the Ada library to perform the bind and/or link operation. The Ada library must always be specified.

To ensure internal data structure integrity of the Ada libraries, libraries are locked for the duration of any operations being performed on them. During compilation, libraryname is normally locked for updating, and other Ada libraries can be locked for reading. During binding/linking, Ada libraries are only locked for reading.

If ada cannot obtain a lock after a suitable number of retries, it displays an informational message and terminates.

Users are strongly discouraged from placing any additional files in Ada library directories. User files in Ada libraries are subject to damage by, or might interfere with, proper operation of ada and related tools.

(3) All other file arguments, including those whose names

end with .o or .a are passed on to the linker ld(1) to be linked into the final program. It is not possible to link-only with the ada(1) command.

(4) Although shown in the preferred order above, options, files, and libraryname arguments can appear in any order.

Environment Variables

The environment variable ADA_PATH is associated with all components of the Ada compilation system. It must be set properly and exported before any component of the Ada compilation system (including ada) can be used.

Normally this variable is defined and set in the system-wide shell startup files /etc/profile (for sh(1) and ksh(1)) and /etc/csh.login (for csh(1)). However, it can be set by a user, either interactively or in a personal shell startup file, .profile (for sh(1) and ksh(1)) or .cshrc (for csh(1)).

ADA_PATH must contain the path name of the directory in which the Ada compiler components have been installed.

The value of this variable must be a rooted directory (that is, it must begin with a /) and the directory specification must not end with a /.

ADAOPTS
The environment variable ADAOPTS can be used to supply commonly used (or default) arguments to ada. ADAOPTS is associated directly with ada, and is not used by any other component of the Ada compilation system.

Arguments can be passed to ada through the ADAOPTS environment variable, as well as on the command line. ada(1) picks up the value of ADAOPTS and places its contents before any arguments on the command line. For example (in sh(1) notation),

```
$ ADAOPTS="-v -e 10"
$ export ADAOPTS
$ ada -L source.ada test.lib
```

is equivalent to

```
$ ada -v -e 10 -L source.ada test.lib
```

Compiler Options
The following options are recognized:

-a string Store the supplied annotation string in the library with the compilation unit. This

string can later be displayed by the unit manager. The maximum length of this string is 80 characters. The default is no string.

-b Display abbreviated compiler error messages (default is to display the long forms).

-c Suppress link phase and if binding occurred, preserve the object file produced by the binder. This option only takes effect if linking would normally occur. Linking normally occurs when binding has been requested.

Use of this option causes an informational message to be displayed on standard error indicating the format of the ld(1) command that should be used to link the program. It is recommended that the user supply additional object (.o) and archive (.a) files and additional library search paths (-lx) only in the places specified by the informational message.

If the -c option is given along with the -d or -D option, the binder must assume the name for the eventual executable file, in order to determine what to name the debug/profiling information file (see -d and -D). If no -o option is given, the debug/profiling information file will be named a.out.cui, and you must make sure the eventual executable file is named a.out. If a -o outfile option is given, the debug/profiling information file will be named outfile.cui, and you must make sure the eventual executable file is named outfile. If the executable is not named as expected, neither ada.probe(1) nor ada.tune(1) will work correctly.

When ld is later used to actually link the program, the following conditions must be met:

1. The .o file generated by the binder must be specified before any HP-UX archive is specified (either explicitly or with -l).

2. If -lc is specified when linking, any .o file containing code that uses stdio(3S) routines must be specified before -lc is specified.

-d

Cause the compiler to store additional information in the Ada library for the units being compiled for use by the Ada debugger (see ada.probe(1)) or Ada profiler (see ada.tune(1)). Only information required for debugging or profiling is saved; the source is not saved (see -D).

Cause the binder to produce a debug/profiling information file for the program being bound so that the resulting program can be manipulated by the Ada debugger. (Note: if you intend to use the Ada profiler you should use the binder option -W b,-p instead of using -d at bind time.) The binder will produce a debug/profiling information file named a.out.cui (unless -o is used to specify an alternate name). If the debug information file name would be truncated by the file system on which it would be created, an error is reported.

Only sources compiled with the -d or -D option contribute information to the debug/profiling information file produced by the binder. The default is not to produce the debug/profiling information (see the Ada Tools Manual for more details). See -c for information on the enteraction between the -o, -d, -D, and -c options.

See the WARNINGS section for information regarding debugging of optimized programs.

-e nnn

Stop compilation after nnn errors (legal range 0..32767, default 50).

-i

Cause any pending or existing instantiations of generic bodies in this Ada library, whose actual generic bodies have been compiled or recompiled in another Ada library, to be compiled (or recompiled) in this Ada library.

This option is treated as a special "source" file and the compilation is performed when the option is encountered among the names of any actual source files.

Any pending or existing instantiations in the same Ada library into which the actual generic body is compiled (or recompiled), do not need this option. Such pending or existing instantiations are automatically compiled (or recompiled) when the actual

generic body is compiled into the same Ada library.

Warning: Compilation (or recompilation) of instantiations either automatically or by using this option only affects instantiations stored as separate units in the Ada library (see -u). Existing instantiations which are "in-line" in another unit are not automatically compiled or recompiled by using this option. Units containing such instantiations must be explicitly recompiled by the user if the actual generic body is recompiled.

-k Cause the compiler to save an internal representation of the source in the Ada library for use by the Ada cross referencer ada.xref(1). By default, the internal representation is not saved.

-lx Cause the linker to search the HP-UX library named either /lib/libx.a (tried first) or /usr/lib/libx.a (see ld(1)).

-m nnn The supplied number is the size in Kbytes to be allocated at compile time to manipulate library information. The range is 500 to 32767. The default is 500. The default size should work in almost all cases. In some extreme cases involving very large programs, increasing this value will improve compilation time.

-n Cause the output file from the linker to be marked as shareable (see -N). Do not use the option -n when also profiling with the option -W b,-p. For details refer to chatr(1) and ld(1).

-o outfile Name the output file from the linker outfile instead of a.out. In addition, if used with the -c option, name the object file output by the binder outfile.o instead of a.out.o. If debugging is enabled (with -d or -D), name the debug information file output by the binder outfile.cui instead of a.out.cui.

If -c is not specified, the temporary object files used in the link operation are deleted, whether or not the link succeeded.

-q Cause the output file from the linker to be marked as demand loadable (see -Q). For

details refer to chatr(1) and ld(1).

-r nnn — Set listing line length to nnn (legal range 60..255, default 79). This option applies to the listing produced by both the compiler and the binder (see -B, -L, and -W b,-L).

-s — Cause the output of the linker to be stripped of symbol table information (see ld(1) and strip(1)).

Use of this option prevents ada.probe(1) and ada.tune(1) from functioning correctly.

-t c,name — Substitute or insert subprocess c with name where c is one or more of a set of identifiers indicating the subprocess(es). This option works in two modes: 1) if c is a single identifier, name represents the full path name of the new subprocess; 2) if c is a set of (more than one) identifiers, name represents a prefix to which the standard suffixes are concatenated to construct the full path name of the new subprocesses.

The possible values of c are the following:

- b binder (standard suffix is adabind)
- c compiler (standard suffix is adacomp)
- 0 same as c
- l linker (standard suffix is ld)

-u — Cause instantiations of generic program unit bodies to be stored as separate units in the Ada library (see -i).

If -u is not specified and the actual generic body has already been compiled when an instantiation of the body is compiled, the body generated by the instantiation is stored "in-line" in the same unit as its declaration.

If -u is specified or the actual generic body has not already been compiled when an instantiation of the body is compiled, the body generated by the instantiation is stored as a separate unit in the Ada library.

The -u option may be needed if a large number of generic instantiations within a given unit result in the overflow of a compiler internal table.

Specifying -u reduces the amount of table space needed, permitting the compiler to complete. However it also increases the number of units used within the Ada library, and also introduces a small amount of overhead at execution time in units which instantiate generics.

-v   Enable verbose mode, producing a step-by-step description of the compilation, binding, and linking process on standard error.

-w   Suppress warning messages.

-x   Perform syntactic checking only. The libraryname argument must be supplied, although the Ada library is not modified (see -X and -G).

-B   Causes the compiler to produce a compilation listing, suppressing page headers and the error summary at the end of the compilation listing. This is useful for avoiding mismatches caused by page headers when comparing a compilation listing with a previous compilation listing of the same program. This option cannot be specified in conjunction with the -L option.

-C   Suppress runtime checks except for stack overflow. Use of this option may result in erroneous (in the Ada sense) program behavior. In addition, some checks (such as those automatically provided by hardware) might not be suppressed (see -R).

If a generic body, or a procedure or function that may be included "inline" (see the +O i and -I options), is compiled with this option in effect, this option setting is "remembered". When the generic is instantiated, or the procedure or function is included "inline", the compiler will produce code for that instantiation or inlining that is consistent with the "remembered" option again being in effect (i.e. the runtime checking level specified to the compiler which is compiling the instatiation or inlined procedure or function is ignored for the duration of the instantiation or inlining and the runtime checking level specified when the generic body or orignal procedure or function was compiled is used instead).

See the Users Guide for more information.

-D Cause the compiler to store additional information in the Ada library for the units being compiled, for use by the Ada debugger (see ada.probe(1)) or Ada profiler (see ada.tune(1)). In addition to saving information required for debugging and profiling, an internal representation of the actual source is saved. This permits accurate source level debugging and profiling at the expense of a larger Ada library if the actual source file changes after it is compiled (see -d). By default, neither debug/profiling information nor source information is stored.

Cause the binder to produce a debug/profiling information file for the program being bound so that the resulting program can be manipulated by the Ada debugger (Note: if you intend to use the Ada profiler you should use the binder option -W b,-p instead of using -D at bind time). The binder wil produce a debug profiling information file named a.out.cui (unless -o is used to specify an alternate name). If the debug information file name would be truncated by the file system on which it is to be created, an error is reported.

Only sources compiled with the -d or -D option contribute information to the debug/profiling information file produced by the binder. The default is not to produce the debug/profiling information (see the Ada Tools Manual for more details). See -c for information on the interaction between the -o, -d, -D, and -c options.

See the WARNINGS section for information regarding debugging of optimized programs.

-G Generate code but do not update the library. This is primarily intended to allow one to get an assembly listing (with -S) without changing the library. The libraryname argument must be supplied, although the Ada library is not modified (see -x and -X).

-H Causes the compiler to produce informational messages of interest to users of the INTERRUPT_MANAGER package (which is described by the appropriate Reference Manual for the

Ada Programming Language, Appendix F). Three types of messages are produced:

1. Information that an Ada runtime system routine will be called from the generated code for the indicated construct. The name of the Ada runtime system routine will be indicated.

2. Information that an Ada type support subprogram (TSS) will be called from the generated code for the indicated construct. The name of the Ada type that the TSS supports will be indicated.

3. Information indicating the size in bytes of the parameter block, for a task entry for which an address clause has been specified.

The first two kinds of messages are of interest when compiling the bodies of signal handlers for use with the INTERRUPT_MANAGER package. The third kind of message is of interest when compiling the specifications of tasks which contain entries which will be called by the INTERRUPT_MANAGER package.

If you generate a complete compiler listing with the -B or -L options, the informational messages appear in the compiler listing at the appropriate source lines. If you do not generate a complete compiler listing, only the source lines that apply to the informational message appear with the informational message. The information normally produced by -H will be suppressed if -b is also specified.

-I	Suppress all inlining. No procedures or functions are expanded inline and pragma inline is ignored. This also prevents units compiled in the future (without this option in effect) from inlining any units compiled with this option in effect.

-L	Write a program listing with error diagnostics to standard output. This option cannot be specified in conjunction with the -B option.

-M main — Invoke the binder after all source files named in the command line (if any) have been successfully compiled. The argument main specifies the entry point of the Ada program; main must be the name of a parameterless Ada-library-level procedure.

The library-level procedure main must have been successfully compiled into (or linked into) the named Ada library, either by this invocation of ada or by a previous invocation of ada (or ada.umgr(1)).

The binder produces an object file named a.out.o (unless -o is used to specify an alternate name), only if the option -c is also specified. The object file produced by the binder is deleted unless the option -c is specified. Note that the alternate name is truncated, if necessary, prior to appending .o.

-N — Cause the output file from the linker to be marked as unshareable (see -n). For details refer to chatr(1) and ld(1).

-O — Invoke the optimizer with full optimizations. See the description of +O under the DEPENDENCIES section for more information.

See the WARNINGS section for information regarding debugging of optimized programs.

-P nnn — Set listing page length to nnn lines (legal range 10..32767 or 0 to indicate no page breaks, default 66). This length is the total number of lines listed per listing page. It includes the heading, header and trailer blank lines, listed program lines, and error message lines. This option applies to the listing produced by both the compiler and the binder (see -L and -W b,-L).

-Q — Cause the output file from the linker to be marked as not demand-loadable (see -q). For details refer to chatr(1) and ld(1).

-R — Suppress all run-time checks. However, some checks (such as those automatically provided by hardware) might not be suppressed. Use of this option may result in erroneous (in the Ada sense) program behavior (see -C).

If a generic body, or a procedure or function

that may be included "inline" (see the +O i and -I options), is compiled with this option in effect, this option setting is "remembered". When the generic is instantiated, or the procedure or function is included "inline", the compiler will produce code for that instantiation or inlining that is consistent with the "remembered" option again being in effect (i.e. the runtime checking level specified to the compiler which is compiling the instatiation or inlined procedure or function is ignored for the duration of the instantiation or inlining and the runtime checking level specified when the generic body or orignal procedure or function was compiled is used instead).

-S
: Write an assembly listing of the code generated to standard output. This output is not in a form suitable for processing with as(1).

-W c,arg1[,arg2,...,argN]
: Cause arg1 through argN to be handed off to subprocess c. Each argi is of the form -argoption[,argvalue], where argoption is the name of an option recognized by the subprocess and argvalue is a separate argument to argoption where necessary. The values that c can assume are those listed under the -t option as well as d (driver program).

To pass the -r (preserve relocation information) option to the linker, use:

```
-W l,-r
```

The following sends the options -m 10 -s 2 to the binder:

```
-W b,-m,10,-s,2
```

Note that all the binder options can be supplied with one -W, (more than one -W can also be used) and that any embedded spaces must be replaced with commas. Note that -W b is the only way to specify binder options.

The -W d option specification allows additional implementation-specific options to be recognized and passed through the compiler driver to the appropriate subprocess. For example,

-W d,-O,eo

sends the option -O eo to the driver, which sends it to the compiler so that the e and o optimizations are performed. Furthermore, a shorthand notation for this mechanism can be used by inserting + in front of the option as follows:

+O eo

This is equivalent to -W d,-O,eo. Note that for simplicity this shorthand is applied to each implementation-specific option individually, and that the argvalue (if any) is separated from the shorthand argoption with white space instead of a comma.

-X — Perform syntactic and semantic checking. The libraryname argument must be supplied, although the Ada library is not modified (see -x and -G).

Binder Options

The following options can be passed to the binder using -W b,...:

-W b,-b — At execution time, the Ada STANDARD_INPUT and STANDARD_OUTPUT files will block if data is not available (STANDARD_INPUT) or if data cannot currently be written (STANDARD_OUTPUT) or if the data to be read or written lies in a locked region. All tasks will be suspended when an I/O operation on one of these standard files blocks. This option is the default if the program contains no tasks (see -W b,-B for more details).

-W b,-k — Keep uncalled subprograms when binding. The default is to remove them.

-W b,-m,nnn — Series 300/400: Set the initial program stack size to nnn units of 1024 bytes (legal range 1..32767, default 10 units = 10 * 1024 bytes = 10240 bytes).

Series 600/700/800: Set the maximum stack limit of the program stack to nnn units of 1024 bytes (legal range 512..32767, defaults to a system-defined limit).

-W b,-p — Cause the binder to link the program in the special manner required by ada.tune(1) and to place additional information in the

debug/profiling information file for use ada.tune(1). Note that neither -d nor -D needs to be specified when binding if this option is specified, as this option causes the same binder action as -d and -D plus the additional actions noted above.

If you specify the -c option when binding, you will need to specify special options and object files when linking with ld(1) in order for ada.tune(1) to function correctly. When you specify -c, the ld(1) command you need to use will be displayed as an informational message by the binder. Consult the Ada Tools Manual for your implementation for further information.

-W b,-s,nnn

Cause round-robin scheduling to be used for tasking programs. Set the time slice to nnn tens of milliseconds (legal range 1..32767 or 0 to turn off time slicing). By default, round-robin scheduling is enabled with a time slice of 1 second (nnn = 100).

The time slice granularity is specified under the DEPENDENCIES section.

-W b,-t,nnn

Set the total task 'STORAGE_SIZE to nnn units of 1024 bytes for each task which does not have a length clause.

Cause this amount of space to be allocated for the total task data area which includes both the task stack and the overhead for the Ada runtime system.

Series 300/400: The Ada runtime system overhead is approximately 3600 bytes, so the minimum usable value of nnn is 4. The default is 32 units, equal to 32 * 1024 bytes = 32768 bytes, less 3600 bytes leaves 29168 bytes for the task stack.

Series 600/700/800: The Ada runtime system overhead is approximately 5400 bytes, so the minimum usable value of nnn is 6. The default is 32 units, equal to 32 * 1024 bytes = 32768 bytes, less 5400 bytes leaves 27368 bytes for the task stack.

If insufficient space is allocated either TASKING_ERROR or STORAGE_ERROR will occur during task elaboration or activation.

The legal range is 1..32767 units.

-W b,-w Suppress warning messages from the binder.

-W b,-x Perform consistency checks without producing an object file, and suppress linking. The -W b,-L option can be used to obtain binder listing information when this option is specified (see -W b,-L below).

-W b,-B At execution time, the Ada STANDARD_INPUT and STANDARD_OUTPUT files will not block if data is not available (STANDARD_INPUT) or if data cannot currently be written (STANDARD_OUTPUT) or if the data to be read or written lies in a locked region. The task attempting the I/O operation that cannot currently be completed will be suspended (and the I/O operation retried later), but other tasks will continue to run as appropriate. This option is the default if the program contains tasks (see -W b,-b for more details).

-W b,-L Write a binder listing with warning/error diagnostics to standard error.

-W b,-S,t Specifies which HP-UX timer should be used to implement task time-slicing. If time-slicing is not also enabled, this option has no effect.

The argument t is a single character that specifies which timer to use. The legal values of t and their meanings are:

a or A : Use the timer which generates SIGALRM for time-slicing.

p or P : Use the timer which generates SIGPROF for time-slicing.

v or V : Use the timer which generates SIGVTALRM for time-slicing.

The default is to use the timer which generates SIGVTALRM for time-slicing.

-W b,-T Suppress procedure traceback in response to runtime errors and unhandled exceptions. This also causes traceback tables to be excluded from the final executable file.

Locks

To ensure the integrity of their internal data structures,

Ada libraries and families are locked for the duration of operations that are performed on them. Normally Ada families are locked for only a short time when libraries within them are manipulated. However, multiple Ada libraries might need to be locked for longer periods during a single operation. If more than one library is locked, ada places an exclusive lock on one library so it can be updated, and a shared lock on the other(s) so that they can remain open for read-only purposes.

An Ada family or library locked for updating cannot be accessed in any way by any part of the Ada compilation system except by the part that holds the lock. An Ada family or library locked for reading can be accessed by any part of the Ada compilation system desiring to read from the Ada family or library.

If ada cannot obtain a lock after a suitable number of retries, it displays an informational message and terminates.

Under some circumstances, an Ada family or Ada library might be locked, but the locking program(s) might have terminated (for example, due to system crash or network failure). If you determine that the Ada family or Ada library is locked but should not be locked, you can remove the lock.

Use ada.unlock(1) to unlock an Ada library and ada.funlock(1) to unlock an Ada family. However, unlocking should be done with care. If an Ada family or Ada library is actually locked by a tool, unlocking it will permit access by other tools that might find the contents invalid or that might damage the Ada family or Ada library.

EXTERNAL INFLUENCES

International Code Set Support
Single-byte character code sets are supported within file names.

DIAGNOSTICS

The diagnostics produced by ada are intended to be self-explanatory. Occasional messages might be produced by the linker.

If a program listing (-B or -L) and/or generated code listing (-S) is requested from the compiler, this listing as well as compiler error messages are written to standard output.

If neither a program listing nor a generated code listing is requested from the compiler, erroneous source lines and compiler error messages are written to standard error.

If a binder listing is requested from the binder (with -W b,-L), the binder listing as well as binder error messages are written to standard error.

If a binder listing is not requested from the binder, binder error messages are written to standard error.

Errors detected during command line processing or during scheduling of the compiler, binder, or linker, are written to standard error. If any compiler, binder, or linker errors occur, ada writes a one-line summary to standard error immediately before terminating.

WARNINGS

Options not recognized by ada are not passed on to the linker. The option -W l,arg can also be used to pass options to the linker explicitly.

ada does not generate an error or warning if both optimization and debugging are requested. However, ada.probe is only capable of limited debugging of optimized code. Certain ada.probe commands may give misleading or unexpected results. For example, object values may be stored in registers; therefore the value displayed from memory may be incorrect. For this reason, the ability to examine or modify objects and expressions may be impaired. Dead-code elimination or code motion may affect single step execution or prevent breakpoints from being set on specific source lines.

DEPENDENCIES

Series 300/400

The compiler option -H is not supported on the Series 300/400.

The binder option -W b,-m behaves differently on the Series 300/400 versus the Series 600/700/800. See the section Binder Options for more information.

The time slice granularity for round-robin scheduling is 20 milliseconds.

The following options are specific to the Series 300/400:

+O what — Selectively invoke optimizations. The what argument must be specified, and indicates which optimizations should be performed. Note that the option -O is equivalent to +O eioE.

The what argument can be a combination of the letters e, i, o, p, E, and P. Either e

or p, but not both, can be specified. Similarly, either E or P, but not both, can be specified. All other combinations are permitted, but only one of each letter, at most, can be specified.

e   Same as p (below).

i   Permit procedures and functions not declared with pragma inline to be expanded inline at the compiler's discretion. Only procedures and functions in the current source file are considered.

Procedures and functions declared with pragma inline are always considered candidates for inline expansion unless -I is specified; this optimization only causes additional procedures and functions to be considered.

o   Peephole optimizations are performed on the final object code.

p   Optimizations are performed to remove unnecessary checks, optimize loops, and remove dead code.

E   Same as P (below).

P   Optimizations are performed on common subexpressions and register allocation.

+h type   Bind/link the program to use the specified type of hardware floating-point assist for user code floating-point operations (see +H). The two types currently supported are 68881 (the MC 68881 math coprocessor) and 68882 (the MC 68882 math coprocessor). The code generated is the same for either type. This is the default if the host system provides a MC 68881 or a MC 68882 coprocessor. This option is ignored if floating point operations were compiled inline with the +i type option.

+i type   Compile the program to inline the specified type of hardware floating-point assist for

user code floating-point operations (see +I). The two types currently supported are 68881 (the MC 68881 math coprocessor) and 68882 (the MC 68882 math coprocessor). The code generated is the same for either type. This is the default if the host system provides a MC 68881 or a MC 68882 coprocessor. Once inlined with this option, the bind/link options +h type and +H are ignored.

+H
: Bind/link the program to use software floating-point routines for user code floating-point operations (see +h). This is the default if the host system does not provide a MC 68881 or a MC 68882 coprocessor. This option is ignored if floating point operations were compiled inline with the +i type option.

+I
: Compile the program to make calls to a math library for user code floating point operations (see +i). The +h or +H options are then used at bind/link time to specify whether hardware or software is used for floating point operations. This is the default if the host system does not provide a MC 68881 or a MC 68882 coprocessor.

Unlike other Series 300/400 compilers, it is not possible to link-only using the ada(1) command. If separate linking is desired, use the ld(1) command.

A successful bind produces a (non-executable) .o file. The .o file is normally deleted unless the option -c is specified.

Series 600/700/800

The binder option -W b,-m behaves differently on the Series 300/400 versus the Series 600/700/800. See the section Binder Options for more information.

The time slice granularity for round-robin scheduling is 10 milliseconds.

The following options are specific to the Series 600/700/800:

+DAarchitecture
: Generate code for the architecture specified. architecture is required. The default code generated for the Series 800 is PA_RISC_1.0. The default code generated

for the Series 700 is PA_RISC_1.1. The default code generation may be overriden using the ADAOPTS environment variable or the command line option +DA. Defined values for architecture are:

1.0 Precision Architecture RISC, version 1.0.
1.1 Precision Architecture RISC, version 1.1.

The compiler determines the target architecture using the following precedence:

1. Command line specification of +DA.
2. Specification of +DA in the ADAOPTS environment variable.
3. The default as mentioned above.

+DSarchitecture
Use the instruction scheduler tuned to the architecture specified. architecture is required. If this option is not used, the compiler uses the instruction scheduler for the architecture on which the program is compiled. Defined values for architecture are:

1.0 Precision Architecture RISC, version 1.0.
1.1 Precision Architecture RISC, version 1.1, general scheduling for the series 700.
1.1a Scheduling for specific models of Precision Architecture RISC, version 1.1.

+O what
Selectively invoke optimizations. The what argument must be specified, and indicates which optimizations should be performed. Note that the option -O is equivalent to +O eilE.

The what argument can be a combination of the letters e, i, 0, 1, p, E, and P. Either e or p, but not both, can be specified. Similarly, either E or P, but not both, can be specified. Similarly, either 0 or 1, but not both, can be specified. All other combinations are permitted, but only one of each letter, at most, can be specified.

e   Same as p (below).

i   Permit procedures and functions not declared with pragma inline to be expanded in-line at the compiler's discretion. Only procedures and functions in the current source file are considered.

Procedures and functions declared with pragma inline are always considered candidates for inline expansion unless -I is specified; this optimization only causes additional procedures and functions to be considered.

0   The code generator performs no optimizations.

1   The code generator performs level 1 optimizations.

p   Optimizations are performed to remove unnecessary checks, optimize loops, and remove dead code.

E   Same as P (below).

P   Optimizations are performed on common subexpressions and register allocation.

+T   Suppress the generation of traceback information at compile time. In addition to suppressing traceback of the current compilation unit at run time, this also reduces the size of the object file in the ada library.

Unlike other Series 600/700/800 compilers, it is not possible to link-only using the ada(1) command. If separate linking is desired, use ld(1).

A successful bind produces a (non-executable) .o file. The .o file is normally deleted unless the option -c is specified.

AUTHOR

Ada was developed by HP and Alsys.

FILES

file.ad?   input file (Ada source file).
libraryname   user Ada library (created using

	ada.mklib(1)) in which compiled units are placed by a successful compilation and from which the binder extracts the units necessary to build a relocatable file for ld(1). Temporary files generated by the compiler are also created in this directory and are automatically deleted on successful completion. Users are strongly discouraged from placing any additional files in Ada library directories. User files in Ada libraries are subject to damage by, or may interfere with proper operation of ada and related tools.
file.o	binder-generated object file or user-specified object file relocated at link time.
a.out	linked executable output file.
file.cui	binder-generated debug/profiling information file.
$ADA_PATH/ada	Ada compilation driver program.
$ADA_PATH/adacomp	Ada compiler.
$ADA_PATH/adabind	Ada binder.
$ADA_PATH/ada_environ	Ada environment description file.
$ADA_PATH/adaargu	Ada argument formatter.
$ADA_PATH/alternate	Ada predefined library, sequential version.
$ADA_PATH/installation	Ada installation family.
$ADA_PATH/public	Ada public family.
$ADA_PATH/err_tpl	Ada compiler/binder error message files.
$ADA_PATH/predeflib	Ada predefined library, tasking version.
$ADA_PATH/libada.a	Ada run-time HP-UX library. Series 600/700/800 only.
$ADA_PATH/libada020.a	Ada run-time HP-UX library (MC68020). Series 300/400 only.
$ADA_PATH/libada881.a	Ada run-time HP-UX library (MC68881). Series 300/400 only.
/lib/crt0.o	C run-time startup.
/lib/libc.a	HP-UX C library.
/lib/libm.a	HP-UX math library

SEE ALSO

ada.cplib(1), ada.fmgr(1), ada.format(1), ada.funlock(1), ada.lmgr(1),
ada.lsfam(1), ada.lslib(1), ada.make(1), ada.mkfam(1), ada.mklib(1),
ada.mvfam(1), ada.mvlib(1), ada.probe(1), ada.protect(1), ada.rmfam(1),
ada.rmlib(1), ada.tune(1), ada.umgr(1), ada.unlock(1),

ada.xref(1),

Ada User's Guide for Ada/300,

Ada User's Guide — HP 9000 Series 600/700/800,

Ada Tools Manual for Ada/300,

Ada Tools Manual — HP 9000 Series 600/700/800,

Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A),

Reference Manual for the Ada Programming Language, Appendix F for Ada/300,

Reference Manual for the Ada Programming Language, Appendix F — HP 9000 Series 600/700/800.

LINKER OPTIONS

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

NAME

ld - link editor

SYNOPSIS

ld [-bdmnqrstvxzENQZ] [-a search] ... [-e epsym] [-h symbol] ... [-o outfile] [-u symbol] ... [-A name] [-B bind] [-L dir] ... [-R offset] [-V num] [-X num] [-lx | file] ...

DESCRIPTION

ld takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so it resolves references to external symbols, assigns final addresses to procedures and variables, revises code and data to reflect new addresses (a process called "relocation"), and updates symbolic debug information (when present in the file). By default, ld produces an executable file that can be run by the HP-UX loader exec(2). Alternatively, the linker can generate a relocatable file that is suitable for further processing by ld (see -r below). It can also generate a shared library (see -b below). The linker marks the output file non-executable if any unresolved external references remain. ld may or may not generate an output file if any other errors occur during its operation; see DEPENDENCIES. ld recognizes three kinds of input files: object files created by the compilers, assembler, or linker (also known as ''.o'' files), shared libraries created by the linker, and archives of object files (called archive libraries). An archive library contains an index of all the externally-visible symbols from its component object files. (The archiver command ar(1) creates and maintains this index.) ld uses this table to resolve references to external symbols.

ld processes files in the same order as they appear on the command line. It includes code and data from an archive library element if and only if that object module provides a definition for a currently unresolved reference within the user's program. It is common practice to list libraries following the names of all simple object files on the command line.

Code from shared libraries is never copied into an executable program, and data is copied only if referenced directly by the program. The dynamic loader /lib/dld.sl is invoked at startup time by /lib/crt0.o if a program uses shared libraries. The dynamic loader attaches each required library to the process and resolves all symbolic references between the program and its libraries. The text segment of a shared library is shared among all processes that use the library.

Environment Variables

Arguments can be passed to the linker through the LDOPTS environment variable as well as on the command line. The linker picks up the value of LDOPTS and places its contents before any arguments on the command line.

The LD_PXDB environment variable defines the full execution path for the debug preprocessor pxdb. The default value is /usr/bin/pxdb. ld invokes pxdb on its output file if that file is executable and contains debug information. To defer invocation of pxdb until the first debug session, set LD_PXDB to /dev/null.

Options

ld recognizes the following options:

-a search — Specify whether shared or archive libraries are searched with the -l option. The value of search should be one of archive, shared, or default. This option can appear more than once, interspersed among -l options, to control the searching for each library. The default is to use the shared version of a library if one is available, or the archive version if not. If either archive or shared is active, only the specified library type is accepted.

-b — Create a shared library rather than a normal executable file. Object files processed with this option should contain position independent code (PIC). See the discussion of PIC in cc(1), f77(1), pc(1), and as(1).

-d — Forces definition of ''common'' storage; i.e., assign addresses and sizes, for -r output.

-e epsym — Set the default entry point address for the output file to be that of the symbol epsym. (This option only applies to executable files.)

-h symbol — Prior to writing the symbol table to the output file, mark this name as ''local'' so that it is no longer externally visible. This ensures that this particular entry will not clash with a definition in another file during future processing by ld. (Of course, this only makes sense with the -r option.) More

than one *symbol* can be specified, but -h must precede each one.

-l*x*
Search a library lib*x*.a or lib*x*.sl, where *x* is one or more characters. The current state of the -a option determines whether the archive (.a) or shared (.sl) version of a library is searched. Because a library is searched when its name is encountered, the placement of a -l is significant. By default, libraries are located in /lib and /usr/lib. If the environment variable LPATH is present in the user's environment, it should contain a colon-separated list of directories to search. These directories are searched instead of the default directories, but -L options can still be used. If a program uses shared libraries, the dynamic loader /lib/dld.sl will attempt to load each library from the same directory in which it was found at link time.

-m
Produce a load map on the standard output.

-n
Generate an (executable) output file with code to be shared by all users. Compare with -N.

-o *outfile*
Produce an output object file by the name *outfile* (default name is a.out).

-q
Generate an (executable) output file that is demand-loadable. Compare with -Q.

-r
Retain relocation information in the output file for subsequent re-linking. The *ld* command does not report undefined symbols. The -r option is incompatible with -A and -b.

-s
Strip the output file of all symbol table, relocation, and debug support information. This might impair or prevent the use of a symbolic debugger on the resulting program. This option is incompatible with -r. (The *strip*(1) command also removes this information.)

-t
Print a trace (to standard output) of each input file as *ld* processes it.

-u symbol
: Enter symbol as an undefined symbol in the symbol table. The resulting unresolved reference is useful for linking a program solely from object files in a library. More than one symbol can be specified, but each must be preceded by -u.

-v
: Display verbose messages during linking. For each library module that is loaded, the linker indicates which symbol caused that module to be loaded.

-x
: Partially strip the output file; that is, leave out local symbols. The intention is to reduce the size of the output file without impairing the effectiveness of object file utilities. Note: use of -x might affect the use of a debugger.

-z
: Arrange for run-time dereferencing of null pointers to produce a SIGSEGV signal. (This is the complement of the -Z option.)

-A name
: This option specifies incremental loading. Linking is arranged so that the resulting object can be read into an already executing program. The argument name specifies a file whose symbol table provides the basis for defining additional symbols. Only newly linked material is entered into the text and data portions of a.out, but the new symbol table reflects all symbols defined before and after the incremental load. Also, the -R option can be used in conjunction with -A, which allows the newly linked segment to commence at the corresponding address. The default starting address is the old value of _end. The -A option is incompatible with -r and -b.

-B bind
: Select run-time binding behavior of a program using shared libraries. The value of bind must be one of immediate or deferred. The default is deferred, which tells the dynamic loader /lib/dld.sl to resolve procedure calls on first reference rather than at program start-up time.

-E — Mark all symbols defined by a program for export to shared libraries. By default, ld can mark only those symbols that are actually referenced by a shared library.

-L dir — Change the algorithm of searching for libx.a or libx.sl to look in dir before looking in default locations. More than one directory can be specified, but each must be preceded by -L. The -L option is effective only if it precedes the -l option on the command line.

-N — Generate an (executable) output file that cannot be shared. This option also causes the data to be placed immediately following the text, and makes the text writable.

-Q — Generate an (executable) output file that is not demand-loadable. (This is the complement of the -q option.)

-R offset — Set the origin (in hexadecimal) for the text (i.e., code) segment.

-V num — Use num as a decimal version stamp identifying the a.out file that is produced. This is not the same as the version information reported by the SCCS what (1) command, nor is it the same as the version information recorded for shared library use.

-X num — Define the initial size for the linker's global symbol table. This reduces link time for very large programs, especially those with a large number of external symbols.

-Z — Arrange to allow run-time dereferencing of null pointers> See the discussions of -Z and pointers in cc(1). (This is the complement of the -z option.)

Defaults

Unless otherwise directed, ld names its output a.out. The -o option overrides this. Executable output files can be shared. The default state of -a is to search shared libraries if available, archive libraries otherwise. The default bind behavior is deferred.

EXTERNAL INFLUENCES

Environment Variables

LANG determines the language in which messages are displayed (Series 700/800 only).

If LANG is not specified or is set to the empty string, a default of "C" (see _lang_(5)) is used instead of LANG.

If any internationalization variable contains an invalid setting, _ld_ behaves as if all internationalization variables are set to "C". See _environ_(5).

DIAGNOSTICS

_ld_ returns zero when the link is successful. A non-zero return code indicates that an error occurred.

EXAMPLES

The following command line links part of a C program for later processing by _ld_. It also specifies a version number of 2 for the output file. (Note the .o suffix for the output object file. This is an HP-UX convention for indicating a linkable object file.)

```
ld -V 2 -r file1.o file2.o -o prog.o
```

The next example links a simple FORTRAN program for use with the _cdb_(1) symbolic debugger. The output file name will be a.out since there is no -o option in the command line. (Note: the particular options shown here are for a Series 300/400 system.)

```
ld -e start /lib/frt0.o ftn.o -lI77 -lF77 -lm -lc
/usr/lib/end.o
```

This example creates a shared library.

```
ld -b -o libfunc.sl func1.o func2.o func3.o
```

Link a program with libfunc.sl but use the archive version of the C library:

```
ld /lib/crt0.o program.o -L . -lfunc -a archive
-lc
```

Link a Pascal program on a Series 300/400 system:

```
ld /lib/crt0.o main.o -lpc -lm -lc
```

WARNINGS

_ld_ recognizes several names as having special meanings. The symbol _end is reserved by the linker to refer to the first address beyond the end of the program's address space.

Similarly, the symbol _edata refers to the first address beyond the initialized data, and the symbol _etext refers to the first address beyond the program text. The linker treats a user definition of any of these symbols as an error. The symbols end, edata, and etext are also defined by the linker, but only if the program contains a reference to these symbols and does not define them (see end(3C) for details).

Through its options, the link editor gives users great flexibility. However, those who invoke the linker directly must assume some added responsibilities. Input options should ensure the following properties for programs:

- When the link editor is called through cc(1), a start-up routine is linked with the user's program. This routine calls exit(2) after execution of the main program. If users call ld directly, they must ensure that the program always calls exit() rather than falling through the end of the entry routine.
- When linking for use with the symbolic debugger cdb, the user must ensure that the program contains a routine called main. Also, the user must link in the file /usr/lib/end.o as the last file named on the command line.

There is no guarantee that the linker will pick up files from archive libraries and include them in the final program in the same relative order that they occur within the library.

DEPENDENCIES

Series 300/400

The default entry point is taken to be text location 0x0 (which is also the default origin of the program text) if shared libraries are not used. Otherwise, the entry point is taken to be the first text location after the extension header placed at the beginning of the text segment by ld for use by /lib/dld.sl. This corresponds to the first procedure in the first input file that the linker reads. If the C startup routine /lib/crt0.o is the first object file on the command

line, the label start denotes the entry point. Use the -e option to select a different entry point.

The version number specified with the -V option must be in the range 0 through 32767. Use of this option is not recommended because this field is used by several HP-UX commands that expect particular values here. Consult the C Programmer's Guide for more details on the version field.

The placement of -L options relative to -l is not significant.

The Series 300/400 linker does not support the following options: -m, -z, and -Z.

On Series 300/400 systems, the compilers place an underscore at the beginning of all external names. Thus, the symbol _end appears to the linker as __end.

ld does not generate an output file if any other errors occur during its operation.

Series 700/800

The linker searches for the symbol $START$ as the program entry point. This symbol is defined in the file /lib/crt0.o, which should be the first file loaded for all programs regardless of source language. Use the -e option to select a different entry point.

When invoking ld directly to link a C program whose main procedure is located in a library, the -u main option should be used to force the linker to load main from the library, since this symbol is not actually referenced until the _start routine is loaded from the C library. When using cc(1) to link the program, the compiler automatically passes this option to the linker. Because of this behavior, do not use cc to link a program containing a FORTRAN or Pascal main program; use f77 or pc instead.

Nonsharable, executable files generated with the -N option cannot be executed via exec(2). Typically, -N is used when rebuilding the kernel or when preparing an image for dynamic loading.

When the -A option is used to do an incremental link, the linker generates extra code where a procedure call crosses a quadrant boundary (a quadrant is one gigabyte, or one fourth of the addressing space). On Series 700/800 systems, text is normally in the first quadrant and data is in the second quadrant. When an object file is intended to be read into an already-executing program, both its code and data must be placed in the second quadrant, since the first quadrant is set to read-only. Procedure calls from one quadrant to the other require the extra code, called inter-space calling stubs. The linker generates an ''export'' stub for the entry point designated in the incremental link, and ''import'' stubs for each procedure in the basis program that is called by the new object file. The import stubs require the existence of a routine in the basis program called _sr4export, which is supplied in /lib/crt0.o. If a procedure in the basis program is

called indirectly by the new object file, the linker cannot detect the crossing of the quadrant boundary, and therefore will not generate the needed stub. A special version of $$dyncall placed in /lib/dyncall.o is provided for handling the inter-quadrant branch. This routine should be linked in when the -A option is in effect.

The Series 700/800 linker does not support the -V option.

The following options are specific to the Series 700/800 linker:

-y symbol — Indicate each file in which symbol appears. Many such options can be given to trace many symbols, but -y must precede each one.

-Cn — Set the maximum parameter-checking level to n. The default maximum is 3. See the language manuals for the meanings of the parameter-checking level.

-D offset — Set the origin (in hexadecimal) for the data space. The default value for offset is 0x40001000.

-G — Strip all unloadable data from the output file. This option is typically used to strip debug information.

-S — Generate an Initial Program Loader (IPL) auxiliary header for the output file, instead of the default HP-UX auxiliary header.

-T — Save the load data and relocation information in temporary files instead of memory during linking. This option reduces the virtual memory requirements of the linker. If the TMPDIR environment variable is set, the temporary files are created in the specified directory, rather than in /tmp.

ld treats both duplicate symbols and unresolved symbols in the same manner: an output file is generated and marked as non-executable if errors occur during its operation.

AUTHOR

ld was developed by AT&T, the University of California, Berkeley, and HP.

FILES

/lib/libx.a	archive libraries
/usr/lib/libx.a	archive libraries
/lib/libx.sl	shared libraries
/usr/lib/libx.sl	shared libraries
a.out	output file
/lib/dld.sl	dynamic loader

Series 300/400

/lib/crt0.o	run-time start-up for C and Pascal
/lib/mcrt0.o	run-time start-up for C and Pascal with profiling (see prof(1))
/lib/gcrt0.o	run-time start-up for C and Pascal with profiling (see gprof(1))
/lib/frt0.o	run-time start-up for FORTRAN
/lib/mfrt0.o	run-time start-up for FORTRAN with profiling (see prof(1))
/lib/gfrt0.o	run-time start-up for FORTRAN with profiling (see gprof(1))
/usr/lib/end.o	for use with cdb/fdb/pdb(1)

Series 700/800

/lib/crt0.o	run-time start-up
/lib/dyncall.o	used with -A-option links
/lib/mcrt0.o	run-time start-up with profiling (see prof(1))
/lib/milli.a	millicode library automatically searched by ld
/lib/gcrt0.o	run-time start-up with profiling (see gprof(1))
/usr/lib/xdbend.o	for use with xdb(1)
/usr/lib/nls/$LANG/ld.cat	message catalog
/tmp/ld*	temporary files

SEE ALSO

ar(1), cc(1), cdb(1), f77(1), gprof(1), nm(1), pc(1), prof(1), strip(1), exec(2), crt0(3), end(3C), a.out(4), ar(4), dld.sl(5).

Programming on HP-UX manual.

STANDARDS CONFORMANCE

ld: SVID2, XPG2

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 SHORT_SHORT_INTEGER is range -128 .. 127;

    type SHORT_INTEGER is range -32768 .. 32767;

    type INTEGER is range -2147483648 .. 2147483647;

    type FLOAT is digits 6 range -3.402823E+38 .. 3.402823E+38;

    type LONG_FLOAT is digits 15 range
        -1.797693134862315E+308 .. 1.797693134862315E+308;

    type DURATION is delta 2#0.00000000000001# range
       -86400.0 .. 86400.0;
    ...

end STANDARD;
```

HP 9000 Series 600, 700, and 800 Computers

Reference Manual for the Ada Programming Language, Appendix F

HP Part No. B2425-90001
Printed in U.S.A. 1991

First Edition
E1291

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY
3000 Hanover Street
Palo Alto, California 94304 U.S.A.

Copyright © 1991 by Hewlett-Packard Company

Print History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Software Version
First Edition	December 1991	B2425A.05.35

Preface

This manual describes the implementation-dependent characteristics of the HP Ada Development System on the HP 9000 Precision Architecture RISC (PA-RISC) machines. The PA-RISC family includes the HP 9000 Series 600, 700, and 800 systems. In this manual, the term "Ada" refers to the Ada Development System running on any Series 600, 700, or 800 computer system. This compiler has been validated using the Ada Compiler Validation Capability (ACVC) test suite from the Ada Joint Program Office.

This manual provides information on machine dependencies as stipulated in the *Reference Manual for the Ada Programming Language* (*Ada RM*). This manual describes the following:

- HP implementation-dependent pragmas and attributes.
- Specifications of the packages SYSTEM and STANDARD.
- Instructions on using type representation clauses to fully specify the layout of data objects in memory.
- Restrictions on unchecked type conversions.
- Implementation-dependent characteristics of input/output packages.
- Information about HP-UX signals and the Ada runtime environment.
- Instructions and examples on calling external subprograms written in Precision Architecture RISC Assembly Language, HP C, HP FORTRAN 77, and HP Pascal.

The following manuals provide additional information about the topics indicated (Hewlett-Packard part numbers are listed in parentheses):

Ada

- *Reference Manual for the Ada Programming Language*
  United States Department of Defense
  ANSI/MIL-STD-1815A-1983
  Order Number 008-000-00394-7
  U.S. Government Printing Office, Washington DC 20402
  (97055-90610)
- *Ada User's Guide*
  (B2425-90002)
- *Ada Tools Manual*
  (B2425-90005)

HP-UX Operating System

- *HP-UX Reference: HP 9000 Computers, 3 Volumes*
  (B2355-90004)
- *Programming on HP-UX*
  (B2355-90010)
- *Device I/O: User's Guide*
  (B1862-90002)
- *HP-UX Portability Guide*
  (B1864-90006)
- *Terminal Control: User's Guide*
  (B1862-90013)

Precision Architecture RISC Procedure Calling Convention

- *PA-RISC Architecture Procedure Calling Convention Reference Manual* (09740-90015)

Precision Architecture RISC Assembly Language

- *Assembly Language Reference Manual* (92432-90001)
- *ADB Tutorial* (92432-90005)

C Language

- *HP C/HP-UX Reference Manual* (92453-90024)
- *HP C Programmer's Guide* (92434-90002)

HP FORTRAN 77

- *HP FORTRAN 77/HP-UX Reference Manual* (92430-90005)
- *HP FORTRAN 77/HP-UX Programmer's Reference* (92430-90004)

HP Pascal

- *HP Pascal/HP-UX Reference Manual* (92431-90005)
- *HP Pascal/HP-UX Programmer's Guide* (92431-90006)
- *HP Pascal/HP-UX Quick Reference Guide* (92431-90007)

NFS® Systems

- *Using NFS Services* (B1013-90000)
- *Installing and Administering NFS Services* (B1013-90001)

NFS® is a trademark of SUN Microsystems, Inc.

Conventions

`lowercase nonbold`	In syntax, represents literals; items are to be entered exactly as shown.
lowercase boldface	In syntax, represents Ada language reserved words.
italics	Represents parameters which must be replaced by a user-supplied variable.
[ ]	Specifies that an element inside brackets is optional.
|	When several elements are separated vertically by bars in a syntax statement, the user must select one of these elements. For example: `A	B	C` User must select A or B or C.
underlining	In a syntax statement, when several elements are separated by bars, underling indicates the default value. For example, `ALL	CURRENT` (CURRENT underlined) `CURRENT` is the default value.
...	A horizontal ellipsis in a syntax statement indicates that a previous element can be repeated. For example: `[itemname...]` Within examples, ellipsis indicate when portions of the example are omitted.

Contents

1. F 1. Implementation Supported Pragmas

F 1.1 Interfacing the Ada Language with Other Languages . . . 1-2
F 1.1.1 Pragma INTERFACE 1-3
F 1.1.2 Pragma INTERFACE_NAME 1-4
F 1.1.3 Example of INTERFACE and INTERFACE_NAME . 1-5
F 1.1.4 Additional Information on INTERFACE and INTERFACE_NAME 1-6
F 1.1.5 Pragma EXPORT 1-8
F 1.1.6 Pragma EXTERNAL_NAME 1-12
F 1.1.7 Example of EXPORT and EXTERNAL_NAME . . . 1-13
F 1.1.8 Pragma EXPORT Applied to Ada Subprograms . . . 1-14
F 1.1.8.1 Calling an Ada Subprogram from Non-Ada Code . 1-14
F.1.1.8.2 Exceptions 1-17
F.1.1.8.3 Obtaining the Non-Ada Call Address in Ada at Runtime 1-18
F.1.1.8.4 Bind-Time Issues 1-19
F.1.1.8.5 HP-UX Signal Handlers in Ada 1-20
F.1.2 Using Text Processing Tools 1-21
F 1.2.1 Pragma INDENT 1-21
F 1.2.2 Pragma LIST 1-22
F 1.2.3 Pragma PAGE 1-22
F 1.3 Affecting the Layout of Array and Record Types 1-23
F 1.3.1 Pragma PACK 1-23
F 1.3.2 Pragma IMPROVE 1-23
F 1.4 Generating Code 1-24
F 1.4.1 Pragma ELABORATE 1-24
F 1.4.2 Pragma INLINE 1-25
F 1.4.3 Pragma SUPPRESS 1-26
F 1.5 Affecting Run Time Behavior 1-27
F 1.5.1 Pragma PRIORITY 1-27

F 1.5.2 Pragma SHARED 1-28
F 1.6 Pragmas Not Implemented 1-29

2. **F 2. Implementation-Dependent Attributes**
F 2.1 Limitation of the Attribute 'ADDRESS 2-2
F 2.2 Attribute SYSTEM.ADDRESS'IMPORT 2-3

3. **F 3. The SYSTEM and STANDARD Packages**
F 3.1 The Package SYSTEM 3-1
F 3.2 The Package STANDARD 3-10

4. **F 4. Type Representation**
F 4.1 Enumeration Types 4-2
F 4.1.1 Internal Codes of Enumeration Literals 4-2
F 4.1.2 Minimum Size of an Enumeration Type or Subtype . . 4-4
F 4.1.3 Size of an Enumeration Type 4-5
F 4.1.4 Alignment of an Enumeration Type 4-5
F 4.2 Integer Types 4-6
F 4.2.1 Predefined Integer Types 4-6
F 4.2.2 Internal Codes of Integer Values 4-6
F 4.2.3 Minimum Size of an Integer Type or Subtype 4-7
F 4.2.4 Size of an Integer Type 4-9
F 4.2.5 Alignment of an Integer Type 4-11
F 4.2.6 Performance of an Integer Type 4-11
F 4.3 Floating Point Types 4-12
F 4.3.1 Predefined Floating Point Types 4-12
F 4.3.2 Internal Codes of Floating Point Values 4-12
F 4.3.3 Minimum Size of a Floating Point Type or Subtype . 4-15
F 4.3.4 Size of a Floating Point Type 4-15
F 4.3.5 Alignment of a Floating Point Type 4-15
F 4.4 Fixed Point Types 4-16
F 4.4.1 Predefined Fixed Point Types 4-16
F 4.4.2 Internal Codes of Fixed Point Values 4-17
F 4.4.3 Small of a Fixed Point Type 4-17
F 4.4.4 Minimum Size of a Fixed Point Type or Subtype . . . 4-18
F 4.4.5 Size of a Fixed Point Type 4-20
F 4.4.6 Alignment of a Fixed Point Type 4-21
F 4.5 Access Types 4-22

F 4.5.1 Internal Codes of Access Values 4-22
F 4.5.2 Collection Size for Access Types 4-22
F 4.5.3 Minimum Size of an Access Type or Subtype 4-24
F 4.5.4 Size of an Access Type 4-24
F 4.5.5 Alignment of an Access Type 4-24
F 4.6 Task Types 4-25
F 4.6.1 Internal Codes of Task Values 4-25
F 4.6.2 Storage for a Task Activation 4-25
F 4.6.3 Minimum Size of a Task Stack 4-26
F 4.6.4 Limitation on Length Clause for Derived Task Types . 4-26
F 4.6.5 Minimum Size of a Task Type or Subtype 4-27
F 4.6.6 Size of a Task Type 4-27
F 4.6.7 Alignment of a Task Type 4-27
F 4.7 Array Types 4-28
F 4.7.1 Layout of an Array 4-28
F 4.7.2 Array Component Size and Pragma PACK 4-28
F 4.7.3 Array Gap Size and Pragma PACK 4-29
F 4.7.4 Size of an Array Type or Subtype 4-31
F 4.7.5 Alignment of an Array Type 4-32
F 4.8 Record Types 4-33
F 4.8.1 Layout of a Record 4-33
F 4.8.2 Bit Ordering in a Component Clause 4-35
F 4.8.3 Value used for SYSTEM.STORAGE_UNIT 4-36
F 4.8.4 Compiler-Chosen Record Layout 4-36
F 4.8.5 Change in Representation 4-37
F 4.8.6 Implicit Components 4-37
F 4.8.7 Indirect Components 4-39
F 4.8.8 Dynamic Components 4-40
F 4.8.9 Representation of the Offset of an Indirect Component 4-43
F 4.8.10 The Implicit Component RECORD_SIZE 4-44
F 4.8.11 The Implicit Component VARIANT_INDEX 4-44
F 4.8.12 The Implicit Component ARRAY_DESCRIPTOR . . 4-46
F 4.8.13 The Implicit Component RECORD_DESCRIPTOR . 4-46
F 4.8.14 Suppression of Implicit Components 4-47
F 4.8.15 Size of a Record Type or Subtype 4-48
F 4.8.16 Size of an Object of a Record Type 4-49
F 4.8.17 Alignment of a Record Subtype 4-49
F 4.9 Data Allocation 4-50

F 4.9.1 Direct Allocation versus Indirect Allocation 4-52
F 4.9.2 Object Deallocation 4-52
F 4.9.2.1 Compiler-Generated Objects 4-52
F 4.9.2.2 Programmer-Generated Objects 4-53
F 4.9.2.3 Program Termination 4-53
F 4.9.3 Dynamic Memory Management 4-54
F 4.9.3.1 Collections of Objects 4-54
F 4.9.3.2 Global Dynamic Objects 4-55
F 4.9.3.3 Local Objects 4-56
F 4.9.3.4 Temporary Objects 4-56
F 4.9.3.5 Reclaiming Heap Storage 4-57

5. F 5. Names for Predefined Library Units

6. F 6. Address Clauses
F 6.1 Objects 6-1
F 6.2 Subprograms 6-2
F 6.3 Constants 6-2
F 6.4 Packages 6-2
F 6.5 Tasks 6-2
F 6.6 Data Objects 6-3
F 6.7 Task Entries 6-3

7. F 7. Restrictions on Unchecked Type Conversions

8. F 8. Implementation-Dependent Input-Output
F 8.1 Ada I/O Packages for External Files 8-1
F 8.1.1 Implementation-Dependent Restrictions on I/O Packages 8-3
F 8.1.2 Correspondence between External Files and HP-UX . 8-3
F 8.1.3 Standard Implementation of External Files 8-7
F 8.1.3.1 SEQUENTIAL_IO Files 8-7
F 8.1.3.2 DIRECT_IO Files 8-8
F 8.1.3.3 TEXT_IO Files 8-10
F 8.1.4 Default Access Protection of External Files 8-11
F 8.1.5 System Level Sharing of External Files 8-11
F 8.1.6 I/O Involving Access Types 8-13
F 8.1.7 I/O Involving Local Area Networks 8-13

F 8.1.8 Potential Problems with I/O From Ada Tasks 8-14
F 8.1.9 I/O Involving Symbolic Links 8-15
F 8.1.10 Ada I/O System Dependencies 8-16
F 8.2 The FORM Parameter 8-18
F 8.2.1 An Overview of FORM Attributes 8-18
F 8.2.2 The Format of FORM Parameters 8-18
F 8.2.3 The FORM Parameter Attribute - File Protection . . 8-21
F 8.2.4 The FORM Parameter Attribute - File Buffering . . . 8-23
F 8.2.5 The FORM Parameter Attribute - File Sharing . . . 8-25
F 8.2.5.1 Interaction of File Sharing and File Buffering . . 8-26
F 8.2.6 The FORM Parameter Attribute - Appending to a File 8-27
F 8.2.7 The FORM Parameter Attribute - Blocking 8-28
F 8.2.7.1 Blocking 8-28
F 8.2.7.2 Non-Blocking 8-30
F 8.2.8 The FORM Parameter - FIFO Control 8-31
F 8.2.9 The FORM Parameter - Terminal Input 8-33
F 8.2.10 The FORM Parameter Attribute - File Structuring . 8-35
F 8.2.10.1 The Structure of TEXT_IO Files 8-35
F 8.2.10.2 The Structure of DIRECT_IO and SEQUENTIAL_IO Files 8-37

9. **F 9. The Ada Development System and HP-UX Signals**
F 9.1 HP-UX Signals Reserved by the Ada Runtime 9-2
F 9.2 Using HP-UX Signals in External Interfaced Subprograms 9-6
F 9.3 HP-UX Signals Used for Ada Exception Handling 9-7
F 9.4 HP-UX Signals Used for Ada Task Management 9-10
F 9.5 HP-UX Signals Used for Ada Delay Timing 9-11
F 9.6 HP-UX Signals Used for Ada Program Termination . . . 9-12
F 9.7 HP-UX Signals Used for Ada Interrupt Entries 9-14
F 9.8 Protecting Interfaced Code from Ada's Asynchronous Signals 9-15
F 9.9 Programming in Ada With HP-UX Signals 9-15

10. F 10. Limitations
F 10.1 Compiler Limitations 10-1
F 10.2 Ada Development Environment Limitations 10-5
F 10.3 Limitations Affecting User-Written Ada Applications . . 10-6
F 10.3.1 Restrictions Affecting Opening or Creating Files . . 10-6
F 10.3.1.1 Restrictions on Path and Component Sizes . . . 10-6
F 10.3.1.2 Additional Conditions that Raise NAME_ERROR 10-6
F 10.3.2 Restrictions on TEXT_IO.FORM 10-7
F 10.3.3 Restrictions on the Small of a Fixed Point Type . . . 10-7
F 10.3.4 Record Type Alignment Clause 10-7

11. F 11. Calling External Subprograms From Ada
F 11.1 General Considerations in Passing Ada Types 11-6
F 11.1.1 Scalar Types 11-6
F 11.1.1.1 Integer Types 11-7
F 11.1.1.2 Enumeration Types 11-7
F 11.1.1.3 Boolean Types 11-8
F 11.1.1.4 Character Types 11-9
F 11.1.1.5 Real Types 11-10
F 11.1.2 Access Types 11-11
F 11.1.3 Array Types 11-13
F 11.1.4 Record Types 11-15
F 11.1.5 Task Types 11-17
F 11.2 Calling Assembly Language Subprograms 11-18
F 11.2.1 Scalar Types and Assembly Language Subprograms . 11-19
F 11.2.1.1 Integer Types and Assembly Language Subprograms 11-19
F 11.2.1.2 Enumeration Types and Assembly Language . . 11-19
F 11.2.1.3 Boolean Types and Assembly Language Subprograms 11-19
F 11.2.1.4 Character Types and Assembly Language Subprograms 11-19
F 11.2.1.5 Real Types and Assembly Language Subprograms 11-19
F 11.2.2 Access Types and Assembly Language Subprograms . 11-20
F 11.2.3 Array Types and Assembly Language Subprograms . 11-20
F 11.2.4 Record Types and Assembly Language Subprograms . 11-20
F 11.3 Calling HP C Subprograms 11-21
F 11.3.1 Scalar Types and HP C Subprograms 11-22

F 11.3.1.1 Integer Types and HP C Subprograms 11-23
F 11.3.1.2 Enumeration Types and HP C Subprograms . . 11-23
F 11.3.1.3 Boolean Types and HP C Subprograms 11-24
F 11.3.1.4 Character Types and HP C Subprograms 11-24
F 11.3.1.5 Real Types and HP C Subprograms 11-25
F 11.3.2 Access Types and HP C Subprograms 11-26
F 11.3.3 Array Types and HP C Subprograms 11-26
F 11.3.4 Record Types and HP C Subprograms 11-31
F 11.4 Calling HP FORTRAN 77 Language Subprograms . . . 11-32
F 11.4.1 Scalar Types and HP FORTRAN 77 Subprograms . . 11-33
F 11.4.1.1 Integer Types and HP FORTRAN 77 Subprograms 11-33
F 11.4.1.2 Enumeration Types and HP FORTRAN 77 Subprograms 11-34
F 11.4.1.3 Boolean Types and HP FORTRAN 77 Subprograms 11-35
F 11.4.1.4 Character Types and HP FORTRAN 77 Subprograms 11-36
F 11.4.1.5 Real Types and HP FORTRAN 77 Subprograms 11-37
F 11.4.2 Access Types and HP FORTRAN 77 Subprograms . 11-38
F 11.4.3 Array Types and HP FORTRAN 77 Subprograms . . 11-39
F 11.4.4 String Types and HP FORTRAN 77 Subprograms . . 11-41
F 11.4.5 Record Types and HP FORTRAN 77 Subprograms . 11-45
F 11.4.6 Other FORTRAN Types 11-45
F 11.5 Calling HP Pascal Language Subprograms 11-46
F 11.5.1 Scalar Types and HP Pascal Subprograms 11-48
F 11.5.1.1 Integer Types and HP Pascal Subprograms . . . 11-48
F 11.5.1.2 Enumeration Types and HP Pascal Subprograms 11-50
F 11.5.1.3 Boolean Types and HP Pascal Subprograms . . 11-51
F 11.5.1.4 Character Types and HP Pascal Subprograms . . 11-51
F 11.5.1.5 Real Types and HP Pascal Subprograms 11-52
F 11.5.2 Access Types and HP Pascal Subprograms 11-52
F 11.5.3 Array Types and HP Pascal Subprograms 11-53
F 11.5.4 String Types and HP Pascal Subprograms 11-56
F 11.5.5 Record Types and HP Pascal Subprograms 11-60
F 11.6 Summary 11-61
F 11.7 Potential Problems Using Interfaced Subprograms . . . 11-64
F 11.7.1. Signals and Interfaced Subprograms 11-64

F 11.7.2 Files Opened by Ada and Interfaced Subprograms . . 11-67

12. **F 12. Interrupt Entries**
F 12.1 Introduction 12-1
F 12.2 Immediate Processing 12-2
F 12.3 Deferred Processing 12-3
F 12.4 Handling an Interrupt Entirely in the Immediate Processing Step 12-4
F 12.5 Initializing the Interrupt Entry Mechanism 12-5
F 12.6 Associating an Ada Handler with an HP-UX Signal . . . 12-7
F 12.6.1 Determining If Your Ada Handler Makes Ada Runtime Calls 12-9
F 12.7 Disassociating an Ada Handler from an HP-UX Signal . . 12-11
F 12.8 Determining How Many Handlers are Installed 12-11
F 12.9 When Ada Signal Handlers Will Not Be Called 12-11
F 12.10 Address Clauses for Entries 12-12
F 12.11 Example of Interrupt Entries 12-13
F 12.12 Specification of the package INTERRUPT_MANAGER 12-23
F 12.13 Ada Runtime Routine Descriptions 12-28

Index

Figures

4-1. Layout of an Array 4-28
4-2. Record layout with an Indirect Component 4-39
4-3. Example of a Data Layout 4-42
11-1. Passing Access Types to Interfaced Subprograms 11-12

Tables

1-1. Ada Pragmas 1-1
4-1. Methods to Control Layout and Size of Data Objects 4-1
4-2. Alignment and Pragma PACK 4-32
8-1. Standard Predefined I/O Packages 8-2
8-2. User Access Categories 8-21
8-3. File Access Rights 8-21
8-4. File Sharing Attribute Modes 8-25
8-5. Text File Terminators 8-36
8-6. Structuring Binary Files with the FORM Parameter 8-38
8-6. Structuring Binary Files with the FORM Parameter (Continued) 8-39
8-6. Structuring Binary Files with the FORM Parameter (Continued) 8-40
9-1. Ada Signals 9-2
11-1. Ada Types and Parameter Passing Modes 11-2
11-2. Ada versus HP C Integer Correspondence 11-23
11-3. Ada versus HP FORTRAN 77 Integer Correspondence 11-33
11-4 Ada versus HP Pascal Integer Correspondence 11-48
11-5. Ada versus HP Pascal Enumeration Correspondence 11-50
11-6. Modes for Passing Parameters to Interfaced Subprograms . . 11-61
11-7. Types Returned as External Function Subprogram Results . . 11-62
11-8. Parameter Passing in the Ada Implementation 11-63
12-1. Heap Management Routines 12-28
12-2. Collection Management (no STORAGE_SIZE representation clause) 12-28
12-3. Collection Management (collections with a STORAGE_SIZE representation clause) 12-29
12-4. Tasking Routines 12-30
12-4. Tasking Routines (Continued) 12-31
12-4. Tasking Routines (Continued) 12-32

12-5. Attributes Routines 12-33
12-6. Attributes for Tasks Routines 12-33
12-7. Support for Enumeration Representation Clauses Routines . . 12-34

1

F 1. Implementation Supported Pragmas

This section describes the predefined language pragmas and the Ada implementation-specific pragmas. Table 1-1 lists these pragmas.

Table 1-1. Ada Pragmas

Action	Pragma Name
Interface with subprograms written in other languages	INTERFACE INTERFACE_NAME
Support text processing tools	INDENT LIST PAGE
Determine the layout of array and record types in memory	PACK IMPROVE
Direct the compiler to generate different code than what is normally generated	ELABORATE INLINE SUPPRESS
Affect tasking programs	PRIORITY SHARED
Allows Ada code and data objects to be referenced by a non-Ada external subprogram.	EXPORT EXTERNAL_NAME

See section "F 1.6 Pragmas Not Implemented" for a list of predefined pragmas not implemented in Ada.

F 1.1 Interfacing the Ada Language with Other Languages

Your Ada programs can call subprograms written in other languages when you use the predefined pragmas INTERFACE and INTERFACE_NAME. Ada supports subprograms written in these languages:

- PA-RISC Assembly Language.
- HP C.
- HP Pascal.
- HP FORTRAN 77 for HP 9000 Series 600, 700, and 800 computers.

Compiler products from vendors other than Hewlett-Packard may not conform to the parameter passing conventions given below. See section "F 11. Calling External Subprograms from Ada" for detailed information, instructions, and examples for interfacing your Ada programs with the above languages.

In addition, data objects declared in other languages can be made accessible to Ada by using the 'IMPORT attribute of the SYSTEM.ADDRESS type (see "F 2.2 Attribute SYSTEM.ADDRESS'IMPORT" in Chapter 2). Data objects declared in a global Ada scope can be referenced by a non-Ada external subprogram when you use the predefined pragma EXPORT. Alternative names for a global Ada data object can be defined when you use the pragma EXTERNAL_NAME.

F 1.1.1 Pragma INTERFACE

The pragma INTERFACE (*Ada RM*, section 13.9) informs the compiler that a non-Ada external subprogram will be supplied when the Ada program is linked. Pragma INTERFACE specifies the programming language used in the external subprogram and the name of the Ada interfaced subprogram. The corresponding parameter calling convention to be used in the interface is implicitly defined in the language specification.

Syntax

pragma INTERFACE (*Language_name*, *Ada_subprogram_name*);

Parameter	Description
Language_name	is one of ASSEMBLER, C, PASCAL, or FORTRAN.
Ada_subprogram_name	is the name used within the Ada program when referring to the interfaced external subprogram.

It is possible to supply a pragma INTERFACE to a library-level subprogram.

1

F 1.1.2 Pragma INTERFACE_NAME

Ada provides the implementation-defined pragma INTERFACE_NAME to associate an alternative name with a non-Ada external subprogram that has been specified to the Ada program by the pragma INTERFACE.

Syntax

```
pragma INTERFACE_NAME (Ada_subprogram_name,
                       "External_subprogram_name");
```

Parameter	Description
Ada_subprogram_name	is the name when referring to the interfaced external subprogram.
External_subprogram_name	is the external name used outside the Ada program.

You must use pragma INTERFACE_NAME whenever the interfaced subprogram name contains characters not acceptable within Ada identifiers or when the interfaced subprogram name contains uppercase letter(s). You can also use a pragma INTERFACE_NAME if you want your Ada subprogram name to be different than the external subprogram name.

If a pragma INTERFACE_NAME is not supplied, the external subprogram name is the name of the Ada subprogram specified in the pragma INTERFACE, with all alphabetic characters shifted to lowercase letters.

The compiler truncates the name to 255 characters if necessary.

Pragma INTERFACE_NAME is allowed at the same places in an Ada program as pragma INTERFACE (see *Ada RM*, section 13.9.) Pragma INTERFACE_NAME must follow the corresponding pragma INTERFACE. If the pragma INTERFACE appears in a declarative part, the pragma INTERFACE_NAME must appear within the same declarative part. although it need *not* immediately follow the pragma INTERFACE. If the pragma INTERFACE appears outside of any program unit, as it does when being applied to a library-level subprogram, the pragma INTERFACE_NAME must appear after the pragma INTERFACE and before any subsequent compilation unit.

F 1.1.3 Example of INTERFACE and INTERFACE_NAME

The following example illustrates the INTERFACE and INTERFACE_NAME pragmas.

```
package SAMPLE_LIB is

  function SAMPLE_DEVICE   (X : INTEGER) return INTEGER;
  function PROCESS_SAMPLE  (X : INTEGER) return INTEGER;

private

  pragma INTERFACE (ASSEMBLER, SAMPLE_DEVICE );
  pragma INTERFACE (C, PROCESS_SAMPLE );

  pragma INTERFACE_NAME (SAMPLE_DEVICE, "Dev10" );
  pragma INTERFACE_NAME (PROCESS_SAMPLE, "DoSample" );

end SAMPLE_LIB;
```

This example defines two Ada subprograms that are known in Ada code as SAMPLE_DEVICE and PROCESS_SAMPLE. When a call to SAMPLE_DEVICE is executed, the program generates a call to the externally supplied assembly function Dev10. Likewise, when a call to PROCESS_SAMPLE is executed, the program will generate a call to the externally supplied HP C function DoSample.

By using the pragma INTERFACE_NAME, the names for the external subprograms to associate with the Ada subprogram are explicitly identified. If pragma INTERFACE_NAME had not been used, the external names referenced would be sample_device and process_sample.

F 1.1.4 Additional Information on INTERFACE and INTERFACE_NAME

Either an object file or an object library that defines the external subprograms must be provided as a command line parameter to the Ada binder. The command line parameter must be provided to the linker `ld(1)` if you call the linker separately. If you do not provide an object file that contains the definition for the external subprogram, the HP-UX linker, `ld(1)`, will issue an error message.

To avoid conflicts with the Ada Runtime System, the names of interfaced external routines should not begin with the letters "`alsy`" or "`A_`" because the Ada runtime system prefixes its internal routines with these prefixes.

When you want to call an HP-UX system call from Ada code, you should use a pragma `INTERFACE` with `C` as the language name. You might need to use a pragma `INTERFACE_NAME` to explicitly supply the external name. This external name must be the same as the name of the system call that you want to call. (See section 2 of the *HP-UX Reference* for details.) In this case it is not necessary to provide the `C` object file to the binder, because it will be found automatically when the linker searches the system library.

When you want to call an HP-UX library function from Ada code, you should use a pragma `INTERFACE` with `C` as the language name. You should use pragma `INTERFACE_NAME` to explicitly supply the external name. This external name must be exactly the same as the name of the library function. (See section 3 of the *HP-UX Reference* for details.) If your library function is located in either the Standard C Library or the Math Library, it is not necessary to provide the object library to the binder because the binder always requests that the linker search these two libraries. If your library function is located in any of the other standard libraries, you must provide the appropriate `-l`*x* option to the binder. The binder will pass this information onto the linker as a request to search the specified library.

Caution All array and record type parameters are passed by reference from Ada code to non-Ada interfaced code. In particular, arrays and records occupying 64 bits or less of storage are passed by reference and are not passed by copy, as is the standard PA-RISC calling convention. Therefore, non-Ada code expecting to receive such array or record parameters must expect to receive them by reference, not by copy. C should declare such parameters to be appropriate pointer types; Pascal should declare such parameters to be VAR parameters; FORTRAN always expects explicit parameters by reference. Note that array and record type parameters occupying more than 64 bits of storage are passed by reference, both by Ada and by the standard PA-RISC calling convention, and require no special precautions.

See section "F 11. Calling External Subprograms From Ada" for additional information on using pragma INTERFACE and pragma INTERFACE_NAME.

1

F 1.1.5 Pragma EXPORT

The pragma EXPORT allows for data objects and subprograms declared in a global Ada scope to be referenced by a non-Ada external subprogram. Pragma EXPORT specifies the programming language and the name of the Ada data object or subprogram. The default name for the externally visible symbol is the name of the Ada object in all lowercase letters. The pragma EXTERNAL_NAME (described in section "F 1.1.6 Pragma EXTERNAL_NAME") can be used to change this default.

Syntax

pragma EXPORT (*Language_name*, *Ada_name*);

Parameter	Description
Language_name	is one of ASSEMBLER, C, PASCAL, or FORTRAN.
Ada_name	is the simple name of the Ada data object or Ada subprogram that is to be made visible to a non-Ada subprogram.

The pragma EXPORT must occur in a declarative part and applies only to data objects or subprograms declared in the same declarative part; that is, generic instantiated objects or renamed objects are excluded. In addition, if an Ada subprogram name is specified to pragma EXPORT, there must be a unique Ada subprogram with that name (that is, the Ada subprogram name must not be overloaded).

The pragma EXPORT can only be used for data objects with direct allocation mode (objects are allocated with indirect allocation mode if they are dynamic or have a significant size; see section "F 4.9 Data Allocation", for details) and for subprograms that are declared in a library package or in a package that is ultimately nested in a library package. The pragma cannot be used for a data object or subprogram that is declared within another subprogram, nor can it be used for a library level subprogram. When applied to an Ada subprogram, the pragma EXPORT is additionally restricted to only being specified in a library package specification.

The pragma EXPORT cannot be used for a subprogram that:

- has an IN OUT or OUT parameter
- has an unconstrained array parameter
- has a task type parameter
- is a function with an array result type
- is a function with a record result type
- is a function with a task type result type
- has a pragma INTERFACE also specified
- has a pragma INLINE also specified

When using pragma EXPORT with one or more Ada subprograms, the main program unit must be an Ada procedure. This Ada procedure can call routines in other languages that themselves call Ada, but the main procedure itself must be in Ada.

Caution The name of any exported subprogram (either the default name or the name specified by pragma EXTERNAL_NAME) is a linker symbol visible to all components of the application. If this name happens to be the same as an HP-UX system call, it causes program failure in unexpected situations. For example, if the name of your exported routine is close, it intercepts all calls to the system routine close(2). This causes failure of the Ada I/O system.

For this reason, make sure that the name you choose is unique, and not either a system call or in use by some other part of your application.

Following is an example of pragma EXPORT applied to an Ada subprogram and pragma INTERFACE applied to a C subprogram. It consists of two files, callback.c and callback.ada, and compilation directions.

The callback.c file is as follows:

```
extern void hi_there ();

/* procedure GO_TO_C gets called from Ada MAIN, adds 5 to
   argument, and calls Ada routine HI_THERE */
void go_to_c (c_arg)
    int c_arg;
{
   hi_there (c_arg + 5);
}
```

The callback.ada file is as follows:

```
with TEXT_IO;
package FROM_C is

   -- Declaration of Ada routine HI_THERE, to be called
   -- from C.  This must be in a library package.
   procedure HI_THERE (ADA_ARG: in INTEGER);

   pragma EXPORT (C, HI_THERE);

end FROM_C;

package body FROM_C is

   procedure HI_THERE (ADA_ARG: in INTEGER) is
      -- This procedure called from C.  It will write a
      -- message including the value passed into ADA_ARG.
   begin
      TEXT_IO.PUT_LINE ("Now in Ada, called from C!");
      TEXT_IO.PUT_LINE
           ("integer passed was" & INTEGER'IMAGE (ADA_ARG));
   end HI_THERE;

end FROM_C;
```

```
with FROM_C; -- **** WITH IS NECESSARY SO FROM_C GETS
             -- **** INCLUDED IN PROGRAM
procedure MAIN is

   -- This is an Ada main procedure.  It will call a C routine
   -- called GO_TO_C, passing the value 5 to that routine.

   -- GO_TO_C will call the Ada routine HI_THERE to demonstrate
   -- callbacks.

   -- The C routine that will call Ada:
   procedure GO_TO_C (C_ARG: in INTEGER);
   pragma INTERFACE (C, GO_TO_C);

begin

   -- Call C.  C will then call Ada.
   GO_TO_C (5);

end MAIN;
```

To compile and run the example, invoke the C compiler to compile callback.c and produce callback.o:

```
cc -c callback.c
```

Now invoke the Ada compiler to compile callback.ada; bind and link it with the file callback.o.

```
ada callback.ada ada_library_name -M main callback.o
```

Now run the result:

```
a.out
Now in Ada, called from C!
integer passed was 10
```

1

F 1.1.6 Pragma EXTERNAL_NAME

The pragma EXTERNAL_NAME is used to supply an alternate externally visible name for a global Ada data object or Ada subprogram that has been exported using a pragma EXPORT. The pragma EXTERNAL_NAME can be used anywhere in an Ada program where the pragma EXPORT is allowed. The pragma EXTERNAL_NAME must occur after the corresponding pragma EXPORT and within the same library package as the corresponding pragma EXPORT.

Syntax

pragma EXTERNAL_NAME (*Ada_name*, "*External_name*");

Parameter	Description
Ada_name	is the simple name of the Ada data object or Ada subprogram that is to be made visible to a non-Ada subprogram.
External_name	is the externally visible name that is to be accessed in the non-Ada subprogram.

You must use the pragma EXTERNAL_NAME whenever the externally visible name contains characters not acceptable within Ada identifiers or when the externally visible name contains an uppercase letter or letters. You can also use the pragma EXTERNAL_NAME if you want your Ada data object name or Ada subprogram name to be different than the externally visible name.

If a pragma EXTERNAL_NAME is not supplied, the externally visible name is the name of the Ada data object or subprogram specified in the pragma EXPORT, with all alphabetic characters shifted to lowercase letters.

The compiler truncates the name to 255 characters if necessary.

F 1.1.7 Example of EXPORT and EXTERNAL_NAME

The following example illustrates the EXPORT and EXTERNAL_NAME pragmas.

```
package ADA_GLOBALS is

   MY_INT  : INTEGER;
   MY_CHAR : CHARACTER;

   procedure MY_PROC (X: INTEGER);
   function  MY_IFUN (A: CHARACTER; B: SHORT_INTEGER)
                      return INTEGER;
   function  MY_FUNC (Z: BOOLEAN) return LONG_FLOAT;

private

   pragma EXPORT (ASSEMBLER, MY_INT);
   pragma EXPORT (C, MY_CHAR);
   pragma EXPORT (C, MY_PROC);
   pragma EXPORT (C, MY_IFUN);
   pragma EXPORT (C, MY_FUNC);

   pragma EXTERNAL_NAME (MY_INT,  "Int_from_Ada");
   pragma EXTERNAL_NAME (MY_CHAR, "Char_from_Ada");
   pragma EXTERNAL_NAME (MY_PROC, "Proc_from_Ada");
   pragma EXTERNAL_NAME (MY_IFUN, "Ifun_from_Ada");
   pragma EXTERNAL_NAME (MY_FUNC, "Func_from_Ada");

end ADA_GLOBALS;
```

This example defines two Ada data objects that are known in Ada code as MY_INT and MY_CHAR and three Ada subprograms that are known in Ada code as MY_PROC, MY_IFUN, and MY_FUNC. The externally visible symbols for the data objects are, respectively, Int_from_Ada and Char_from_Ada and for the subprograms, Proc_from_Ada, Ifun_from_Ada, and Func_from_Ada.

By using the pragma EXTERNAL_NAME, the names of the external symbols are explicitly identified. If pragma EXTERNAL_NAME had not been used, the external names would be my_int, my_char, my_proc, my_ifun, and my_func.

F 1.1.8 Pragma EXPORT Applied to Ada Subprograms

In this section, the term "exported Ada subprogram" refers to an Ada subprogram to which the pragma EXPORT has been applied. An exported Ada subprogram can be called from Ada code as usual; the subprogram will not behave any differently just because the pragma EXPORT has been specified.

F 1.1.8.1 Calling an Ada Subprogram from Non-Ada Code

An exported Ada subprogram must not be called from non-Ada code before the body of the exported Ada subprogram has been elaborated. If non-Ada code calls an exported Ada subprogram prior to the elaboration of the Ada subprogram body, the results are unpredictable and could lead to program failure.

When calling an exported Ada subprogram from non-Ada code, the parameters passed to Ada from non-Ada code must be compatible with the data types and sizes of the parameters that the exported Ada subprogram is expecting.

Exported Ada subprograms can only have parameters of mode IN. All record and array parameters must be passed by non-Ada code to Ada by reference.

If an exported Ada subprogram has default parameter values, these values are ignored when the exported Ada subprogram is called from non-Ada code. Non-Ada code must always pass all parameters explicitly to Ada code.

With respect to parameter type compatibility, the action of calling an exported Ada subprogram from non-Ada code is very similar to calling non-Ada code (via pragma INTERFACE) from Ada code. Section "F 11. Calling External Subprograms From Ada", describes the correspondences, or lack of correspondences, between Ada types and non-Ada types and should be consulted for information on which non-Ada objects are compatible with which Ada objects.

Caution All array and record type parameters are expected to be passed by reference from non-Ada code to exported Ada code. In particular, arrays and records occupying 64 bits or less of storage must be passed by reference and must not be passed by copy, as is the standard PA-RISC calling convention. Therefore, non-Ada code expecting to pass such array or record type parameters to Ada must pass such parameters by reference not by copy. C must pass the addresses of such parameters cast to appropriate pointer types; Pascal must declare such parameters to be `VAR` parameters in the `EXTERNAL` declaration; FORTRAN always passes explicit parameters by reference. Note that array and record type parameters occupying more that 64 bits of storage are passed by reference, both by Ada and by the standard PS-RISC calling convention, and require no special precautions.

The next sections describe general conditions and constraints that apply to Ada being called from Assembly, C, FORTRAN, and Pascal.

Assembler

Ada expects any parameter passed, except LONG_FLOAT by value, to be passed as a 4 byte quantity, sign extended as necessary, in the location (register or memory) appropriate for its parameter type and position in the parameter list (LONG_FLOAT by value is expected to be passed as a 8 byte quantity).

C

Ada expects any parameter passed, except LONG_FLOAT by value, to be passed as a 4 byte quantity, sign extended as necessary, in the location (register or memory) appropriate for its parameter type and position in the parameter list (LONG_FLOAT by value is expected to be passed as a 8 byte quantity).

If an exported Ada subprogram is called from C, and C is operating in non-ANSI mode, or it is operating in ANSI mode, but lacks a function prototype for the called function. C will convert all `float` values to `double` when passing them as parameters. Therefore, passing parameters from C to Ada when Ada is expecting a parameter of type `FLOAT` requires that C be operating in ANSI mode with a function prototype for the Ada subprogram.

FORTRAN

The exported Ada subprogram parameters that correspond to parameters passed explicitly from FORTRAN must all be of an access type or of type SYSTEM.ADDRESS as FORTRAN passes such parameters by reference. The exported Ada subprogram parameters that correspond to parameters passed implicitly from FORTRAN (such as the length of a FORTRAN string parameter; see "F 11.4.4 String Types and HP FORTRAN 77 Subprograms" in Chapter 11) must be of an appropriate scalar type (not an access type nor SYSTEM.ADDRESS). This type is usually INTEGER.

Pascal

Ada expects any parameter passed, except LONG_FLOAT by value, to be passed as a 4 byte quantity, sign extended as necessary, in the location (register or memory) appropriate for its parameter type and position in the parameter list (LONG_FLOAT by value is expected to be passed as a 8 byte quantity). The exported Ada subprogram parameters that correspond to Pascal VAR parameters must all be of the appropriate access type or of type `SYSTEM.ADDRESS` as Pascal passes such parameters by reference. Non-scalar parameters can only be passed to Ada using Pascal VAR parameters, they may not be passed using Pascal value parameters. The exported Ada subprogram parameters that correspond to Pascal value parameters must all be of an appropriate scalar, access, or `SYSTEM.ADDRESS` type as Pascal passes such parameters by value.

The following example shows C calling the exported Ada subprograms MY_PROC, MY_IFUN, and MY_FUNC which are declared in the ADA_GLOBALS example given in "F 1.1.7 Example of EXPORT and EXTERNAL_NAME":

```
extern void Proc_from_Ada ();
extern int  Ifun_from_Ada ();
extern void Func_from_Ada ();

call_ada (i, flag)
short i; int flag;
{
   double dres;
   int    ires;
   short  temp;

   Proc_from_Ada (i * 28);

   ires = Ifun_from_Ada ('!', i);

   Proc_from_Ada (ires);

   /*
    * Note the conversion of 'flag' to the appropriate Ada Boolean
    * value.
    */

   dres = Func_from_Ada (flag ? 1 : 0);
}
```

F.1.1.8.2 Exceptions

An exported Ada subprogram must *not* allow an exception to propagate out of itself if it was called by a non-Ada caller. If an exception is propagated back to a non-Ada caller, the behavior of the Ada runtime is unpredictable and may result in program failure. If the exported Ada subprogram was called by Ada code and the Ada subprogram is aware of that fact, it can safely propagate exceptions out of itself.

F.1.1.8.3 Obtaining the Non-Ada Call Address in Ada at Runtime

When pragma EXPORT (and optionally pragma EXTERNAL_NAME) is applied to an Ada subprogram, an externally visible name is created such that the Ada subprogram can be called directly by that externally visible name from non-Ada code. However, the Ada code may need to obtain the address of an exported Ada subprogram at run time. For example, it might pass such an address as a parameter to non-Ada code so that the non-Ada code can use the address to call an Ada subprogram (a "callback" situation). The function EXPORTED_SUBPROGRAM is supplied in the package SYSTEM to obtain the "non-Ada call address" of an exported Ada subprogram. The function SYSTEM.EXPORTED_SUBPROGRAM is passed a single parameter, the 'ADDRESS value of an exported Ada subprogram, and it returns the address which non-Ada code must call to invoke that Ada subprogram. See section "F 3.1 The Package SYSTEM", for more information.

Caution The value of 'ADDRESS of an exported Ada subprogram is *not* the address that non-Ada code should call to invoke the exported Ada subprogram. The address that non-Ada code must call to invoke an Ada subprogram is obtained as the result of passing the 'ADDRESS value of the exported Ada subprogram to SYSTEM.EXPORTED_SUBPROGRAM; the function result value is the address that the non-Ada code must call. If non-Ada code attempts to call an exported Ada subprogram using the 'ADDRESS value for that subprogram, the result is unpredictable and will most likely result in program failure.

F.1.1.8.4 Bind-Time Issues

To ensure that the desired exported Ada subprograms are present in the executable program produced by the binder, the Ada library level packages that contain those exported subprograms must be "withed" into at least one Ada program unit that will be present in the executable program. No special action is needed if the Ada program already uses any of the facilities from a library level package that contains exported Ada subprograms because that package will, of necessity, already be "withed" somewhere in the program. Only if the Ada program does not use any of the facilities from a library level package that contains exported Ada subprograms (for example, if that package only contains exported Ada subprograms that are called directly from non-Ada code), will it be necessary to take extra action to ensure the presence of the exported Ada subprograms in the executable program. If no obvious place to "with" such a package exists, the package can always be "withed" into the main program procedure.

Note The Ada binder supports a mechanism to eliminate uncalled subprograms. Within Ada compilation units, subprograms that are never called *(or have their 'ADDRESS taken)* can be eliminated from the executable program produced by the binder. Because exported Ada subprograms legitimately might never be called by Ada code (for example, if they are only called by non-Ada code), they are automatically protected from uncalled subprogram elimination. Therefore, if your program "withed" a library level package that declares one or more exported Ada subprograms, those subprograms will always be present in the executable file produced by the binder (even if they are also not called from non-Ada code). In addition, if any of the exported Ada subprograms call other Ada subprograms, those other Ada subprograms (and in turn, repeatedly, any Ada subprograms they call) will also be present in the executable program. Therefore, care must be taken with respect to the placement of exported Ada subprograms in packages and the "withing" of those packages into your program to avoid including in your executable file the code for Ada subprograms that your program does not call (either from Ada or from non-Ada code).

F.1.1.8.5 HP-UX Signal Handlers in Ada

Although possible, it is not recommended that you apply the pragma `EXPORT` to an Ada subprogram so that the Ada subprogram can be specified as an HP-UX signal handler. If you do so, the Ada subprogram that is to act as a signal handler must be compiled with checks off (using the `-R` option) and it must not call any Ada runtime system routines. The Ada subprogram must not call HP-UX routines or other non-Ada code, either via pragma `INTERFACE` or via a binding. If the Ada subprogram calls another Ada subprogram, the called subprogram must follow the same constraints.

Using the interrupt entry mechanism to provide Ada subprograms as a signal handlers is preferred. This is described in section "F 12. Interrupt Entries".

F 1.2 Using Text Processing Tools

The pragma INDENT is a formatting command that affects the HP supplied formatter, ada.format(1). This pragma does not affect the compilation listing output of the compiler. The pragmas LIST and PAGE are formatting commands that affect the compilation listing output of the compiler.

F 1.2.1 Pragma INDENT

Ada provides the implementation-defined pragma INDENT to assist in formatting Ada source code. You can place these pragmas in the source code to control the actions of ada.format(1).

Syntax

```
pragma INDENT ( ON | OFF );
```

Parameter	Description
OFF	ada.format does not modify the source lines after the pragma.
ON	ada.format resumes its action after the pragma.

The default for pragma INDENT is ON.

F 1.2.2 Pragma LIST

The pragma LIST affects only the compilation listing output of the compiler. It specifies that the listing of the compilation is to be continued or suspended until a LIST pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing. The compilation listing feature of the compiler is enabled by issuing one of the compiler options -L or -B to the ada(1) command.

Syntax

```
pragma LIST ( ON | OFF );
```

Parameter	Description
OFF	The listing of the compilation is suspended after the pragma.
ON	The listing of the compilation is resumed and the pragma is listed.

The default for pragma LIST is ON.

F 1.2.3 Pragma PAGE

The pragma PAGE affects the compilation listing output of the compiler. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).

Syntax

```
pragma PAGE;
```

F 1.3 Affecting the Layout of Array and Record Types

The pragmas PACK and IMPROVE affect the layout of array and record types in memory.

F 1.3.1 Pragma PACK

The pragma PACK takes the simple name of an array type as its only argument. The allowed positions for this pragma and the restrictions on the named type are governed by the same rules as for a representation clause. The pragma specifies that storage minimization should be the main criterion when selecting the representation of the given type.

Syntax

```
pragma PACK (array_type_name);
```

The pragma PACK is not implemented for record types on Ada. You can use a record representation clause to minimize the storage requirements for a record type.

The pragma PACK is discussed further in section "F 4.7 Array Types."

F 1.3.2 Pragma IMPROVE

The pragma IMPROVE, an implementation-defined pragma, suppresses implicit components in a record type.

Syntax

```
pragma IMPROVE ( TIME | SPACE , [ON =>] record_type_name);
```

The default for pragma IMPROVE is TIME. This pragma is discussed further in section "F 4.8 Record Types."

F 1.4 Generating Code

The pragmas ELABORATE, INLINE, and SUPPRESS direct the compiler to generate different code than would have been normally generated. These pragmas can change the run time behavior of an Ada program unit.

F 1.4.1 Pragma ELABORATE

The pragma ELABORATE is used when a dependency upon elaboration order exists. Normally the Ada compiler is given the freedom to elaborate library units in any order. This pragma specifies that the bodies for each of the library units named in the pragma must be elaborated before the current compilation unit. If the current compilation unit is a subunit, the bodies of the named library units must be elaborated before the body of the parent of the current subunit.

Syntax

pragma ELABORATE (*library_unit_name* [, *library_unit_name*] ...);

This pragma takes as its arguments one or more simple names, each of which denotes a library unit. This pragma is only allowed immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument must be the simple name of a library unit that was identified by the context clause. (See the *Ada RM*, section 10.5, for additional information on elaboration of library units .)

F 1.4.2 Pragma INLINE

The pragma `INLINE` specifies that the subprogram bodies should be expanded inline at each call whenever possible; in the case of a generic subprogram, the pragma applies to calls of its instantiations. If the subprogram name is overloaded, the pragma applies to every overloaded subprogram. Note that pragma `INLINE` has no effect on function calls appearing inside package specifications.

Syntax

pragma `INLINE` `(`*subprogram_name* `[,`*subprogram_name*`] ... );`

This pragma takes as its arguments one or more names, each of which is either the name of a subprogram or the name of a generic subprogram. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. See the *Ada RM*, section 6.3.2, for additional information on inline expansion of subprograms.

This pragma can be suppressed at compile time by issuing the compiler option `-I` to the `ada(1)` command.

F 1.4.3 Pragma SUPPRESS

The pragma SUPPRESS allows the compiler to omit the given check from the place of the pragma to the end of the declarative region associated with the innermost enclosing block statement or program unit. For a pragma given in a package specification, the permission extends to the end of the scope of the named entity.

Syntax

pragma SUPPRESS (*check_identifier* [, [ON =>] *name*]);

The pragma SUPPRESS takes as arguments the identifier of a check and optionally the name of either an object, a type or subtype, a subprogram, a task unit, or a generic unit. This pragma is only allowed at the place of a declarative item in a declarative part or a package specification.

If the pragma includes a name, the permission to omit the given check is further restricted: it is given only for operations on the named object or on all objects of the base type of a named type or subtype; for calls of a named subprogram; for activations of tasks of the named task type; or for instantiations of the given generic unit. (See the *Ada RM*, section 11.7, for additional information on suppressing run time checks.)

The compiler can be directed to suppress all run time checks by issuing the compiler option -R to the ada(1) command. The compiler can also be directed to suppress all run time checks except for stack checks by issuing the compiler option -C to the ada(1) command.

F 1.5 Affecting Run Time Behavior

The pragmas PRIORITY and SHARED affect the run time behavior of a tasking program.

F 1.5.1 Pragma PRIORITY

The pragma PRIORITY specifies the priority to be used for the task or tasks of the task type. When the pragma is applied within the outermost declarative part of the main subprogram, it specifies the priority to be used for the environment task, which is the task that encloses the main subprogram. If a pragma PRIORITY is applied to a subprogram that is not the main subprogram, it is ignored.

Syntax

pragma PRIORITY (*static_expression*);

The pragma PRIORITY takes as its argument a static expression of the predefined integer subtype PRIORITY. For Ada, the range of the subtype PRIORITY is 1 to 16. Note that during an entry call invoked by an interrupt handler, the priority of a task is temporarily raised to a value higher than PRIORITY'LAST. The priority value is specified when the interrupt handler is installed; see section "F 12.6 Associating an Ada Handler with an HP-UX Signal", for details.

The PRIORITY pragma is only allowed within the specification of a task unit or within the outermost declarative part of the main subprogram.

These task priorities are only relative to other Ada tasks that are concurrently executing with the environment task. This pragma does *not* change the priority of an Ada task or the Ada environment task relative to other HP-UX processes. All the Ada tasks execute within a single HP-UX process. This HP-UX process executes together with other HP-UX processes and is scheduled by the HP-UX kernal. To change the priority of an HP-UX process, see the command nice(1). See the *Ada RM*, section 9.8, for additional information on task priorities.

1

F 1.5.2 Pragma SHARED

The pragma SHARED specifies that every read or update of the variable is a synchronization point for that variable. The type for the variable object is limited to scalar or access types because each read or update must be implemented as an indivisible operation.

The effect of pragma SHARED on a variable object is to suppress the promotion of this object to a register by the compiler. The compiler suppresses this optimization and ensures that any reference to the variable always retrieves the value stored by the most recent update operation.

Syntax

pragma SHARED (*variable_simple_name*);

The pragma SHARED takes as its argument a simple name of a variable. This pragma is only allowed for a variable declared by an object declaration and whose type is a scalar or access type; the variable declaration and the pragma must both occur (in this order) within the same declarative part or package specification.

See the *Ada RM*, section 9.11, for additional information on shared variables.

F 1.6 Pragmas Not Implemented

The following predefined language pragmas are not implemented and will issue a warning at compile time:

pragma CONTROLLED (*access_type_simple_name*);

pragma MEMORY_SIZE (*numeric_literal*);

pragma OPTIMIZE (TIME | SPACE);

pragma STORAGE_UNIT (*numeric_literal*);

pragma SYSTEM_NAME (*enumeration_literal*);

See the *Ada RM*, Appendix B, for additional information on these predefined language pragmas.

2

F 2. Implementation-Dependent Attributes

In addition to the representation attributes discussed in the *Ada RM*, section 13.7.2, there are five implementation-defined representation attributes:

```
'OFFSET
'RECORD_SIZE
'VARIANT_INDEX
'ARRAY_DESCRIPTOR
'RECORD_DESCRIPTOR
```

These implementation-defined attributes are only used to refer to implicit components of record types inside a record representation clause. Using these attributes outside of a record representation clause will cause a *compiler error* message. For additional information, see section "F 4.8 Record Types".

2

F 2.1 Limitation of the Attribute 'ADDRESS

The attribute 'ADDRESS is implemented for all entities that have meaningful addresses. The compiler will issue the following warning message when the prefix for the attribute 'ADDRESS refers to an object that has a meaningless address:

```
The prefix of the 'ADDRESS attribute denotes a program unit that
has no meaningful address: the result of such an evaluation is
SYSTEM.NULL_ADDRESS.
```

The following entities do not have meaningful addresses and will cause the above compilation warning if used as a prefix to 'ADDRESS:

- A constant that is implemented as an immediate value (that is, a constant that does not have any space allocated for it).
- A package identifier that is not a library unit or a subunit.
- A function that renames an enumeration literal.

Additionally, the attribute 'ADDRESS, when applied to a task or task type, returns different values depending on the elaboration status of the task body. In particular, the value returned by the attribute 'ADDRESS changes after the elaboration of the task body. Therefore, the attribute *task*'ADDRESS or *task_type*'ADDRESS should be used only after the body of the task is elaborated.

F 2.2 Attribute SYSTEM.ADDRESS'IMPORT

This implementation of Ada defines an additional attribute for the type SYSTEM.ADDRESS. The attribute 'IMPORT can be applied to the type SYSTEM.ADDRESS. This attribute is a function with two parameters; the parameters are described in the table below.

Syntax

```
SYSTEM.ADDRESS'IMPORT ("Language_name", "external_symbol_name");
```

Parameter	Description
Language_name	Specifies the language. This parameter is a static Ada string constant that must be either **C**, **ASSEMBLER**, **PASCAL**, or **FORTRAN**. The characters used in the language specification can be uppercase or lowercase letters.
external_symbol_name	Specifies the name of an external data object. This parameter is a static Ada string constant.

The result is a value of the type SYSTEM.ADDRESS that can be used to denote this object in an address clause (see section "F 6. Address Clauses" for details.)

2

The following example shows the use of SYSTEM.ADDRESS'IMPORT in Ada address clauses to provide Ada access to global data objects declared in C.

C code:

```
/* This C code declares a variety of globally visible data
objects that can be accessed from an Ada program that uses
SYSTEM.ADDRESS'IMPORT.
*/

extern int errno;

struct {
   short f1;
   short f2;
} ada_info = {-123, -456};

static int ada_data[10] = {12, 5, 55, 7, 31, 45, 4, 2, 88, 0};

int *ada_data_ptr = {ada_data};

struct {
   int   *a1;
   short  a2;
} ada_info_with_ptr = {ada_data, -789};
```

The sample Ada program follows.

Ada code:

```
with SYSTEM, UNCHECKED_CONVERSION;
package IMPORT_EXAMPLE is

   -- Import a simple C scalar variable, in this case errno.

   ERRNO: INTEGER;
   for ERRNO use at SYSTEM.ADDRESS'IMPORT ("C", "errno");

   -- Import a C struct which contains no pointer values as
   -- an Ada record (see below for importing records which
   -- contain pointer fields).  Note that a representation
   -- specification is used to guarantee that Ada allocates
   -- the record fields the same way C allocates the struct
   -- fields.

   type ADA_INFO is
      record
         f1: SHORT_INTEGER;
         f2: SHORT_INTEGER;
      end record;

   for ADA_INFO use
      record
         f1 at 0 range 0..15;
         f2 at 2 range 0..15;
      end record;

   AI : ADA_INFO;
   for AI use at SYSTEM.ADDRESS'IMPORT ("C", "ada_info");
```

```
-- Import a C pointer object as an Ada access type.  If
-- such a pointer was imported directly as an Ada access
-- type value (in a manner similar to ERRNO above), the
-- elaboration code for this declaration section would
-- initialize the Ada access type value to null, modifying
-- the C pointer object.  The technique of renaming the
-- dereferenced result of an unchecked conversion prevents
-- Ada from initializing the Ada access type value and
-- leaves the value that C initialized the pointer object
-- with, intact.  Of course if you do want Ada to initialize
-- a non-Ada pointer object to null, import it directly
-- similarly to the manner in which ERRNO is imported.

type ADA_DATA                 is array (1..10) of INTEGER;
type ADA_DATA_ACCESS          is access ADA_DATA;
type ADA_DATA_ACCESS_ACCESS is access ADA_DATA_ACCESS;

function ADDRESS_TO_ADA_DATA_ACCESS_ACCESS is new
         UNCHECKED_CONVERSION
            (SYSTEM.ADDRESS, ADA_DATA_ACCESS_ACCESS);

ADA : ADA_DATA_ACCESS renames
                 ADDRESS_TO_ADA_DATA_ACCESS_ACCESS
                    (SYSTEM.ADDRESS'IMPORT
                       ("C", "ada_data_ptr")).all;
```

```
-- Import a C struct which contains a pointer value as an
-- Ada record.  Note that a representation specification is
-- used to guarantee that Ada allocates the record fields
-- the same way C allocates the struct fields.  If such a
-- record was imported directly as an Ada record type (in a
-- manner similar to ADA_INFO above), the elaboration code
-- for this declaration section would initialize the Ada
-- access type field of the record to null, modifying the
-- C record object.  The technique of renaming the result of
-- an unchecked conversion prevents Ada from initializing
-- the Ada access type field and leaves the value that C
-- initialized the record object with, intact.  A similar
-- set of declarations can be used to import any arbitrary
-- record as protection against initializing any access value
-- fields it may have (which could be nested at any depth
-- in the record such that they might be accidentally be
-- overlooked).  Of course if you do want Ada to initialize
-- pointer fields or elements of a non-Ada record or array
-- to null, import the record or array directly, similarly
-- to the manner in which ADA_INFO is imported.

type ADA_INFO_WITH_PTR is
   record
      a1: ADA_DATA_ACCESS;
      a2: SHORT_INTEGER;
   end record;
```

```
   for ADA_INFO_WITH_PTR use
      record
         a1 at 0 range 0..31;
         a2 at 4 range 0..15;
      end record;

   type ADA_INFO_WITH_PTR_ACCESS is access ADA_INFO_WITH_PTR;

   function ADDRESS_TO_ADA_INFO_WITH_PTR_ACCESS is new
            UNCHECKED_CONVERSION
               (SYSTEM.ADDRESS, ADA_INFO_WITH_PTR_ACCESS);

   AIWP : ADA_INFO_WITH_PTR renames
             ADDRESS_TO_ADA_INFO_WITH_PTR_ACCESS
                (SYSTEM.ADDRESS'IMPORT
                   ("C", "ada_info_with_ptr")).all;

end IMPORT_EXAMPLE;

with IMPORT_EXAMPLE;
procedure USE_IMPORT (N: INTEGER) is

begin -- USE_IMPORT

   if N < 1 or else N > 10 then
      -- Change errno to a value from the imported record.

      IMPORT_EXAMPLE.ERRNO := INTEGER (IMPORT_EXAMPLE.AI.F2);
```

```
         -- Change what the C pointer points at (note that this
         -- does not change the contents of the ada_data array in
         -- C, it changes the array that the C ada_data_ptr
         -- points at).

         IMPORT_EXAMPLE.ADA   := new IMPORT_EXAMPLE.ADA_DATA;

         -- Fill in the new array

         for I in 1..10 loop
            IMPORT_EXAMPLE.ADA (I) := I;
         end loop;
      else
         -- Change errno to a value from the imported pointed
         -- to array.

         IMPORT_EXAMPLE.ERRNO := IMPORT_EXAMPLE.ADA (N);

         -- Change that element of the currently pointed to array.

         IMPORT_EXAMPLE.ADA (N) := INTEGER(IMPORT_EXAMPLE.AIWP.A2);
      end if;

   end USE_IMPORT;
```

3

F 3. The SYSTEM and STANDARD Packages

This section contains a complete listing of the two predefined library packages: SYSTEM and STANDARD. These packages both contain implementation-dependent specifications.

F 3.1 The Package SYSTEM

The specification of the predefined library package SYSTEM follows:

```
package SYSTEM is

   type NAME is (HP9000_PA_RISC);

   SYSTEM_NAME  : constant NAME := HP9000_PA_RISC;

   STORAGE_UNIT : constant := 8;

   MEMORY_SIZE  : constant := 2**31-1;

   MIN_INT      : constant := - (2**31);

   MAX_INT      : constant := 2**31 - 1;

   MAX_DIGITS   : constant := 15;

   MAX_MANTISSA : constant := 31;
```

```
FINE_DELTA    : constant := 2#1.0#E-31;

TICK          : constant := 0.010;  -- 10 milliseconds

subtype PRIORITY is INTEGER range 1 .. 16;

type ADDRESS is private;
NULL_ADDRESS : constant ADDRESS; -- set to NULL

-- -----------------------------------------------------------
-- The functions TO_INTEGER and TO_ADDRESS are provided for
-- backwards compatibility with Ada/800 04.35
-- -----------------------------------------------------------

function TO_INTEGER (LEFT : ADDRESS) return INTEGER;
-- -----------------------------------------------------------
-- Converts an ADDRESS to and INTEGER.
-- -----------------------------------------------------------

function TO_ADDRESS (LEFT : INTEGER) return ADDRESS;
-- -----------------------------------------------------------
-- Converts and INTEGER to an ADDRESS.
-- -----------------------------------------------------------

function VALUE (LEFT : in STRING) return ADDRESS;
-- -----------------------------------------------------------
-- Converts a string to an address.  The string can represent
-- either an unsigned address (i.e.  "16#XXXXXXXX#" where
-- XXXXXXXX is in the range 0..FFFFFFFF) or a signed address
-- (i.e.  "-16#XXXXXXXX#" where XXXXXXXX is in the
-- range 0..7FFFFFFF).  Leading blanks are ignored.  The
-- exception CONSTRAINT_ERROR is raised if the string has
-- not the proper syntax.
-- -----------------------------------------------------------
```

```
ADDRESS_WIDTH : constant := 3 + 8 + 1;
subtype ADDRESS_STRING is STRING(1..ADDRESS_WIDTH);

function IMAGE (LEFT : in ADDRESS) return ADDRESS_STRING;
-- ---------------------------------------------------------
-- Converts an address to a string.  The returned string has
-- the unsigned representation described for the VALUE
-- function.
-- ---------------------------------------------------------

type OFFSET is range -2**31 .. 2**31-1;
-- ---------------------------------------------------------
-- This type is used to measure a number of storage units
-- (bytes). The type is an Ada integer type.
-- ---------------------------------------------------------

function SAME_SPACE_ID
  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
-- ---------------------------------------------------------
-- This function returns true if the two addresses have the
-- same space id.
-- ---------------------------------------------------------

ADDRESS_ERROR : exception;
-- ---------------------------------------------------------
 -- This exception is raised by "<", "<=", ">", ">=", and "-"
-- (with two ADDRESS operands) if the two addresses do not
-- have the same space id.  The exception CONSTRAINT_ERROR
-- can be raised by "+" and "-" if the result of ADDRESS does
-- not have the same space id as the ADDRESS operand.
-- ---------------------------------------------------------
```

```
function "+" (LEFT : in ADDRESS; RIGHT : in OFFSET)
             return ADDRESS;
function "+" (LEFT : in OFFSET;  RIGHT : in ADDRESS)
             return ADDRESS;
function "-" (LEFT : in ADDRESS; RIGHT : in OFFSET)
             return ADDRESS;
-- --------------------------------------------------------
-- These routines provide support for address computations.
-- The meaning of the "+" and "-" operators is architecture
-- dependent. For the HP 9000/PA-RISC consider the ADDRESS
-- parameter to be the address of the first byte, of an array
-- of contiguous bytes, that grows from lower toward higher
-- (in an unsigned sense) memory addresses.
--
-- The "+" function returns the address of the byte at offset
-- OFFSET in the ADDRESS array.  In C syntax it returns:
--    &(((char *) ADDRESS)[OFFSET])
--
-- The "-" function returns the address of the byte at offset
-- -OFFSET in the ADDRESS array.  In C syntax it returns:
--    &(((char *) ADDRESS)[-OFFSET])
-- --------------------------------------------------------

function "-" (LEFT : in ADDRESS; RIGHT : in ADDRESS)
             return OFFSET;
-- --------------------------------------------------------
-- Returns the distance between the given addresses.  The
-- result is signed.
-- --------------------------------------------------------
```

```
function "<=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
function "<"  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
function ">=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
function ">"  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
-- ------------------------------------------------------------
-- Perform a comparison on addresses.  The comparison
-- is unsigned.
-- ------------------------------------------------------------

function "mod" (LEFT : in ADDRESS; RIGHT : in POSITIVE)
              return NATURAL;
-- ------------------------------------------------------------
-- Returns the offset of LEFT relative to the memory block
-- immediately below it starting at a multiple of RIGHT
-- storage units.
-- ------------------------------------------------------------

type ROUND_DIRECTION is (DOWN, UP);

function ROUND (VALUE     : in ADDRESS;
          DIRECTION : in ROUND_DIRECTION;
          MODULUS   : in POSITIVE) return ADDRESS;
-- ------------------------------------------------------------
-- Returns the given address rounded to a specific value.
-- ------------------------------------------------------------
```

```
-- ----------------------------------------------------------
-- The functions ALIGN, ALIGN and IS_ALIGNED are provided for
-- backwards compatibility with Ada/800 04.35
-- ----------------------------------------------------------

function ALIGN (LEFT : ADDRESS) return ADDRESS;
-- ----------------------------------------------------------
-- Align the given address up to four bytes boundary
-- (equivalent to SYSTEM.ROUND (LEFT, SYSTEM.UP, 4);)
-- ----------------------------------------------------------

function ALIGN (LEFT : ADDRESS; ALIGNMENT : INTEGER)
        return ADDRESS;
-- ----------------------------------------------------------
-- Align the given address up to the alignment boundary if the
-- alignment is positive (equivalent to
-- SYSTEM.ROUND (LEFT, SYSTEM.UP, POSITIVE (ALIGNMENT));).
-- Align the given address down to the alignment boundary if
-- the alignment is negative (equivalent to
-- SYSTEM.ROUND (LEFT, SYSTEM.DOWN, POSITIVE (-ALIGNMENT));).
-- ----------------------------------------------------------

function IS_ALIGNED (LEFT : ADDRESS; ALIGNMENT : POSITIVE)
        return BOOLEAN;
-- ----------------------------------------------------------
-- Returns TRUE if the LEFT is aligned at the ALIGNMENT
-- boundary (equivalent to
-- SYSTEM."mod" (LEFT, ALIGNMENT) = 0).
-- ----------------------------------------------------------
```

```
generic
  type TARGET is private;
function FETCH_FROM_ADDRESS (A : in ADDRESS)
      return TARGET;
-- -------------------------------------------------------------
-- Return the bit pattern stored at address A, as a value of
-- the specified TARGET type.
--
-- WARNING: These routines may give unexpected results if used
-- with unconstrained types.
-- -------------------------------------------------------------

generic
  type TARGET is private;
procedure ASSIGN_TO_ADDRESS
    (A : in ADDRESS; T : in TARGET);
-- -------------------------------------------------------------
-- Store the bit pattern representing the value of the
-- specified TARGET object, into address A.
--
-- WARNING: These routines may give unexpected results if used
-- with unconstrained types.
-- -------------------------------------------------------------

type OBJECT_LENGTH is range 0 .. 2**31 -1;
-- -------------------------------------------------------------
-- This type is used to designate the size of an object in
-- storage units.
-- -------------------------------------------------------------
```

```
procedure MOVE (TO     : in ADDRESS;
                FROM   : in ADDRESS;
                LENGTH : in OBJECT_LENGTH);
-- -----------------------------------------------------------
-- Copies LENGTH storage units starting at the address FROM
-- to the address TO.  The source and destination may overlap.
-- Use of this procedure in optimized code may lead to
-- unexpected results.
-- -----------------------------------------------------------

-- Exported subprogram types, exceptions, and functions.

type EXPORTED_SUBPROGRAM_ADDRESS is new INTEGER;

NOT_EXPORTED_SUBPROGRAM : exception;

function EXPORTED_SUBPROGRAM (ADA_ADDRESS : in ADDRESS)
               return EXPORTED_SUBPROGRAM_ADDRESS;
```

3

```
-- -----------------------------------------------------------
-- When the parameter ADA_ADDRESS is passed the 'ADDRESS of
-- an Ada subprogram which has been exported via pragma EXPORT,
-- the function returns as a result, the "address" which must
-- be called by non-Ada code to invoke that Ada subprogram.
-- Note that the address provided by 'ADDRESS must NOT be
-- called directly by non-Ada code to invoke an exported Ada
-- subprogram (the result of doing so is unpredictable and will
-- most probably result in program failure). The result of
-- this function must always be used as the address to be
-- called by non-Ada code.  If ADA_ADDRESS is not the
-- 'ADDRESS of an exported Ada subprogram, the function will
-- raise the exception NOT_EXPORTED_SUBPROGRAM.
-- -----------------------------------------------------------

private

  -- private part of package SYSTEM

end SYSTEM;
```

3

F 3.2 The Package STANDARD

The specification of the predefined library package STANDARD follows:

```
package STANDARD is

  -- The operators that are predefined for the types declared
  -- in this package are given in comments since they are
  -- implicitly declared.  Italics are used for pseudo-names
  -- of anonymous types (such as universal_real,
  -- universal_integer, and universal_fixed) and for undefined
  -- information (such as any_fixed_point_type).


  -- Predefined type BOOLEAN
  type BOOLEAN is (FALSE, TRUE);

  -- The predefined relational operators for this type are
  -- as follows (these are implicitly declared):
  -- function "="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "/=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "<"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "<=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function ">"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function ">=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

  -- The predefined logical operands and the predefined
  -- logical negation operator are as follows (these are
  -- implicitly declared):

  -- function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "or"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

  -- function "not" (RIGHT : BOOLEAN) return BOOLEAN;
```

```
-- Predefined universal types

-- type universal_integer is predefined;

-- The predefined operators for the type universal_integer
-- are as follows (these are implicitly declared):
-- function "=" (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;
-- function "/=" (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;
-- function "<"  (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;
-- function "<=" (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;
-- function ">"  (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;
-- function ">=" (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;

-- function "+"   (RIGHT : universal_integer)
-- return universal_integer;
-- function "-"   (RIGHT : universal_integer)
-- return universal_integer;
-- function "abs" (RIGHT : universal_integer)
-- return universal_integer;

-- function "+"   (LEFT, RIGHT : universal_integer)
-- return universal_integer;
-- function "-"   (LEFT, RIGHT : universal_integer)
-- return universal_integer;
-- function "*"   (LEFT, RIGHT : universal_integer)
-- return universal_integer;
-- function "/"   (LEFT, RIGHT : universal_integer)
-- return universal_integer;
```

```
-- function "rem" (LEFT, RIGHT : universal_integer)
-- return universal_integer;
-- function "mod" (LEFT, RIGHT : universal_integer)
-- return universal_integer;
-- function "**"  (LEFT  : universal_integer;
--                 RIGHT : INTEGER) return universal_integer;

-- type universal_real is predefined;

-- The predefined operators for the type universal_real
-- are as follows (these are implicitly declared):
-- function "="  (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;
-- function "/=" (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;
-- function "<"  (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;
-- function "<=" (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;
-- function ">"  (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;
-- function ">=" (LEFT, RIGHT : universal_integer)
-- return BOOLEAN;

-- function "+"   (RIGHT : universal_integer)
-- return universal_integer;
-- function "-"   (RIGHT : universal_integer)
-- return universal_integer;
-- function "abs" (RIGHT : universal_integer)
-- return universal_integer;
-- function "+"   (LEFT, RIGHT : universal_integer)
-- return universal_integer;
```

```
-- function "-"    (LEFT, RIGHT : universal_integer)
-- return universal_integer;
-- function "*"    (LEFT, RIGHT : universal_integer)
-- return universal_integer;
-- function "/"    (LEFT, RIGHT : universal_integer)
-- return universal_integer;

-- function "**"   (LEFT : universal_real;
--                   RIGHT : INTEGER) return universal_real;
-- In addition, the following operators are
-- predefined for universal types:

-- function "*"    (LEFT  : universal_integer;
--                   RIGHT : universal_real)
--             return universal_real
-- function "*"    (LEFT  : universal_real;
--                   RIGHT : universal_integer)
--             return universal_ real;
-- function "/"    (LEFT  : universal_real;
--                   RIGHT : universal_integer)
--             return universal_real;

-- type universal_fixed is predefined;

-- The only operators declared for this type are:
-- function "*"    (LEFT  : any_fixed_point_type;
--                   RIGHT : any_fixed_point_type)
-- return universal_fixed;
-- function "/"    (LEFT  : any_fixed_point_type;
--                   RIGHT : any_fixed_point_type)
-- return universal_fixed;
```

```
-- Predefined and additional integer types

type SHORT_SHORT_INTEGER is range -128 .. 127; -- 8 bits long
-- This is equivalent to -(2**7) .. (2**7)-1
-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : SHORT_SHORT_INTEGER)
-- return BOOLEAN;
-- function "/=" (LEFT, RIGHT : SHORT_SHORT_INTEGER)
-- return BOOLEAN;
-- function "<"  (LEFT, RIGHT : SHORT_SHORT_INTEGER)
-- return BOOLEAN;
-- function "<=" (LEFT, RIGHT : SHORT_SHORT_INTEGER)
-- return BOOLEAN;
-- function ">"  (LEFT, RIGHT : SHORT_SHORT_INTEGER)
-- return BOOLEAN;
-- function ">=" (LEFT, RIGHT : SHORT_SHORT_INTEGER)
-- return BOOLEAN;

-- function "+"   (RIGHT : SHORT_SHORT_INTEGER)
-- return SHORT_SHORT_INTEGER;
-- function "-"   (RIGHT : SHORT_SHORT_INTEGER)
-- return SHORT_SHORT_INTEGER;
-- function "abs" (RIGHT : SHORT_SHORT_INTEGER)
-- return SHORT_SHORT_INTEGER;
-- function "+"   (LEFT,RIGHT: SHORT_SHORT_INTEGER)
-- return SHORT_SHORT_INTEGER;
-- function "-"   (LEFT,RIGHT: SHORT_SHORT_INTEGER)
-- return SHORT_SHORT_INTEGER;
-- function "*"   (LEFT,RIGHT: SHORT_SHORT_INTEGER)
-- return SHORT_SHORT_INTEGER;
-- function "/"   (LEFT,RIGHT: SHORT_SHORT_INTEGER)
-- return SHORT_SHORT_INTEGER;
```

```
-- function "rem" (LEFT,RIGHT: SHORT_SHORT_INTEGER)
-- return SHORT_SHORT_INTEGER;
-- function "mod" (LEFT,RIGHT: SHORT_SHORT_INTEGER)
-- return SHORT_SHORT_INTEGER;

-- function "**"  (LEFT  : SHORT_SHORT_INTEGER;
--                 RIGHT : INTEGER) return SHORT_SHORT_INTEGER;

type SHORT_INTEGER is range -32_768 ..  32_767; --16 bits long

-- This is equivalent to -(2**15) .. (2**15)-1
-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;

-- function "+"  (RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "-"  (RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "abs"(RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "+"   (LEFT, RIGHT : SHORT_INTEGER)
-- return SHORT_INTEGER;
-- function "-"   (LEFT, RIGHT : SHORT_INTEGER)
-- return SHORT_INTEGER;
-- function "*"   (LEFT, RIGHT : SHORT_INTEGER)
-- return SHORT_INTEGER;
-- function "/"   (LEFT, RIGHT : SHORT_INTEGER)
-- return SHORT_INTEGER;
```

```
-- function "rem" (LEFT, RIGHT : SHORT_INTEGER)
-- return SHORT_INTEGER;
-- function "mod" (LEFT, RIGHT : SHORT_INTEGER)
-- return SHORT_INTEGER;

-- function "**"  (LEFT  : SHORT_INTEGER;
--                  RIGHT : INTEGER) return SHORT_INTEGER;

type INTEGER is range -2_147_483_648 .. 2_147_483_647;
-- type INTEGER is 32 bits long

-- This is equivalent to -(2**31) .. (2**31)-1
-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : INTEGER) return BOOLEAN;

-- function "+"   (RIGHT : INTEGER) return INTEGER;
-- function "-"   (RIGHT : INTEGER) return INTEGER;
-- function "abs" (RIGHT : INTEGER) return INTEGER;
-- function "+"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "-"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "*"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "/"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;

-- function "**"  (LEFT : INTEGER; RIGHT : INTEGER)
-- return INTEGER;
```

```
-- Predefined INTEGER subtypes
subtype NATURAL is INTEGER range 0 .. INTEGER'LAST;
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;

-- Predefined and additional floating point types

type FLOAT is digits 6 range -- 32 bits long
-2#1.111_1111_1111_1111_1111_1111#E+127 ..
 2#1.111_1111_1111_1111_1111_1111#E+127;
--
-- This is equivalent to -(2.0 - 2.0**(-23)) * 2.0**127 ..
--                         +(2.0 - 2.0**(-23)) * 2.0**127
--
--  This is approximately equal to the decimal range:
--  -3.402823E+38 .. +3.402823E+38

-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "+"   (RIGHT : FLOAT) return FLOAT;
-- function "-"   (RIGHT : FLOAT) return FLOAT;
-- function "abs" (RIGHT : FLOAT) return FLOAT;

-- function "+"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "-"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "*"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "/"   (LEFT, RIGHT : FLOAT) return FLOAT;

-- function "**"  (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;
```

```
type LONG_FLOAT is digits 15 range -- 64 bits long

-- Note: the following ranges are intentionally split
--       over two lines.
-- In actual Ada programs, the values must be on one line.
-2#1.1111_1111_1111_1111_1111_1111_1111_
 1111_1111_1111_1111_1111_1111#E+1023
 ..
 2#1.1111_1111_1111_1111_1111_1111_1111_
 1111_1111_1111_1111_1111_1111#E+1023;
--
-- This is equivalent to -(2.0 - 2.0**(-52)) * 2.0**1023 ..
--       +(2.0 - 2.0**(-52)) * 2.0**1023 ..
-- This is approximately equal to the decimal range:
-- -1.797693134862315E+308 .. +1.797693134862315E+308

-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "+"   (RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "-"   (RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "abs" (RIGHT : LONG_FLOAT) return LONG_FLOAT;

-- function "+"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "-"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "*"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "/"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;

-- function "**"  (LEFT : LONG_FLOAT; RIGHT : INTEGER)
         return LONG_FLOAT
```

```
--This implementation does not provide any other
--floating point types

-- Predefined type DURATION
type DURATION is delta 2#0.000_000_000_000_01#
range -86_400.0 .. 86_400.0;
--
-- DURATION'SMALL derived from this delta is 2.0**(-14),
-- which is the maximum precision that an object of type
-- DURATION can have and still be representable in this
-- implementation. This has an approximate decimal equivalent
-- of 0.000061 (61 microseconds).  The predefined operators
-- for the type DURATION are the same as for any
-- fixed point type.

-- This implementation provides many anonymous predefined
-- fixed point types.  They consist of fixed point types
-- whose "small" value is a power of 2.0 and whose mantissa
-- can be expressed using 31 or less binary digits.

-- Predefined type CHARACTER

-- The following lists characters for the standard ASCII
-- character set.  Character literals corresponding to
-- control characters are not identifiers; they are
-- indicated in italics in this section.

type CHARACTER is
 ( nul, soh, stx, etx,  eot, enq, ack, bel,
   bs,  ht,  lf,  vt,   ff,  cr,  so,  si,
   dle, dc1, dc2, dc3,  dc4, nak, syn, etb,
   can, em,  sub, esc,  fs,  gs,  rs,  us,
```

3

```
' ', '!', '"', '#', '$', '%', '&', ''',
'(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', ':', ';', '<', '=', '>', '?',

'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',

'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', del);

--The predefined operators for the type CHARACTER are
--the same as for any enumeration type.

-- Predefined type STRING (RM 3.6.3)
type STRING is array (POSITIVE range <>) of CHARACTER;

-- The predefined operators for this type are as follows:
-- function "="  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : STRING) return BOOLEAN;
```

```
-- Predefined catenation operators
-- function "&" (LEFT : STRING;    RIGHT : STRING)
return STRING;
-- function "&" (LEFT : CHARACTER; RIGHT : STRING)
return STRING;
-- function "&" (LEFT : STRING;    RIGHT : CHARACTER)
return STRING;
-- function "&" (LEFT : CHARACTER; RIGHT : CHARACTER)
return STRING;



-- Predefined exceptions
CONSTRAINT_ERROR : exception;
NUMERIC_ERROR    : exception;
PROGRAM_ERROR    : exception;
STORAGE_ERROR    : exception;
TASKING_ERROR    : exception;

-- Predefined package ASCII
package ASCII is

  -- Control characters
  NUL  : constant CHARACTER := nul;
  SOH  : constant CHARACTER := soh;
  STX  : constant CHARACTER := stx;
  ETX  : constant CHARACTER := etx;
  EOT  : constant CHARACTER := eot;
  ENQ  : constant CHARACTER := enq;
  ACK  : constant CHARACTER := ack;
  BEL  : constant CHARACTER := bel;
```

3

```
BS    : constant CHARACTER := bs;
HT    : constant CHARACTER := ht;
LF    : constant CHARACTER := lf;
VT    : constant CHARACTER := vt;
FF    : constant CHARACTER := ff;
CR    : constant CHARACTER := cr;
SO    : constant CHARACTER := so;
SI    : constant CHARACTER := si;
DLE   : constant CHARACTER := dle;
DC1   : constant CHARACTER := dc1;
DC2   : constant CHARACTER := dc2;
DC3   : constant CHARACTER := dc3;
DC4   : constant CHARACTER := dc4;
NAK   : constant CHARACTER := nak;
SYN   : constant CHARACTER := syn;
ETB   : constant CHARACTER := etb;
CAN   : constant CHARACTER := can;
EM    : constant CHARACTER := em;
SUB   : constant CHARACTER := sub;
ESC   : constant CHARACTER := esc;
FS    : constant CHARACTER := fs;
GS    : constant CHARACTER := gs;
RS    : constant CHARACTER := rs;
US    : constant CHARACTER := us;
DEL   : constant CHARACTER := del;

-- other characters
EXCLAM        : constant CHARACTER := '!';
QUOTATION     : constant CHARACTER := '"';
SHARP         : constant CHARACTER := '#';
DOLLAR        : constant CHARACTER := '$';
PERCENT       : constant CHARACTER := '%';
AMPERSAND     : constant CHARACTER := '&';
COLON         : constant CHARACTER := ':';
```

```
SEMICOLON     : constant CHARACTER := ';';
QUERY         : constant CHARACTER := '?';
AT_SIGN       : constant CHARACTER := '@';
L_BRACKET     : constant CHARACTER := '[';
BACK_SLASH    : constant CHARACTER := '\';
R_BRACKET     : constant CHARACTER := ']';
CIRCUMFLEX    : constant CHARACTER := '^';
UNDERLINE     : constant CHARACTER := '_';
GRAVE         : constant CHARACTER := '`';
L_BRACE       : constant CHARACTER := '{';
BAR           : constant CHARACTER := '|';
R_BRACE       : constant CHARACTER := '}';
TILDE         : constant CHARACTER := '~';

-- Lower case letters
LC_A : constant CHARACTER := 'a';
LC_B : constant CHARACTER := 'b';
LC_C : constant CHARACTER := 'c';
LC_D : constant CHARACTER := 'd';
LC_E : constant CHARACTER := 'e';
LC_F : constant CHARACTER := 'f';
LC_G : constant CHARACTER := 'g';
LC_H : constant CHARACTER := 'h';
LC_I : constant CHARACTER := 'i';
LC_J : constant CHARACTER := 'j';
LC_K : constant CHARACTER := 'k';
LC_L : constant CHARACTER := 'l';
LC_M : constant CHARACTER := 'm';
LC_N : constant CHARACTER := 'n';
LC_O : constant CHARACTER := 'o';
LC_P : constant CHARACTER := 'p';
LC_Q : constant CHARACTER := 'q';
LC_R : constant CHARACTER := 'r';
```

```
      LC_S : constant CHARACTER := 's';
      LC_T : constant CHARACTER := 't';
      LC_U : constant CHARACTER := 'u';
      LC_V : constant CHARACTER := 'v';
      LC_W : constant CHARACTER := 'w';
      LC_X : constant CHARACTER := 'x';
      LC_Y : constant CHARACTER := 'y';
      LC_Z : constant CHARACTER := 'z';

   end ASCII;

end STANDARD;
```

4

F 4. Type Representation

This chapter explains how data objects are represented and allocated by the HP Ada compiler for the HP 9000 Series 600, 700, and 800 Computer Systems and how to control this using representation clauses.

The representation of a data object is closely connected with its type. Therefore, this section successively describes the representation of enumeration, integer, floating point, fixed point, access, task, array, and record types. For each class of type, the representation of the corresponding data object is described. Except for array and record types, the description for each class of type is independent of the others. Because array and record types are composite types, it is necessary to understand the representation of their components.

Ada provides several methods to control the layout and size of data objects; these methods are listed in Table 4-1.

Table 4-1. Methods to Control Layout and Size of Data Objects

Method	Type Used On
pragma PACK	array
pragma IMPROVE	record
enumeration representation clause	enumeration
record representation clause	record
size specification clause	any type

F 4.1 Enumeration Types

Syntax (Enumeration representation clause)

for *enumeration-type-name* **use** *aggregate*;

The aggregate used to specify this mapping is written as a one-dimensional aggregate, for which the index subtype is the enumeration type and the component type is *universal_integer*. An **others** choice is *not* permitted in this aggregate.

F 4.1.1 Internal Codes of Enumeration Literals

When no enumeration representation clause applies to an enumeration type, the internal code associated with an enumeration literal is the position number of the enumeration literal. Thus, for an enumeration type with *n* elements, the internal codes are the integers 0, 1, 2, .. , *n*-1.

An enumeration representation clause can be provided to specify the value of each internal code as described in the *Ada RM*, section 13.3. The values used to specify the internal codes must be in the range -(2**31) to (2**31)-1.

The following example illustrates the use of an enumeration representation clause.

Example

```
type COLOR is (RED, ORANGE, YELLOW, GREEN, AQUA, BLUE, VIOLET);

for COLOR use
       (RED    => 10,
        ORANGE => 20,
        YELLOW => 40,
        GREEN  => 80,
        AQUA   => 160,
        BLUE   => 320,
        VIOLET => 640);
```

In the above example, the internal representation for GREEN will be the integer 80. The attributes 'PRED and 'SUCC will still return YELLOW and AQUA, respectively. Also, section 13.3(6) in the *Ada RM* states that the 'POS attribute will still return the positional value of the enumeration literal. In the case of GREEN, the value that 'POS returns will be 3 and not 80. The only way to examine the internal representation of the enumeration literal is to write the value to a file or use UNCHECKED_CONVERSION to examine the value in memory.

F 4.1.2 Minimum Size of an Enumeration Type or Subtype

The minimum size of an enumeration subtype is the minimum number of bits necessary for representing the internal codes in normal binary form.

A static subtype of a null range has a minimum size of one. Otherwise, define m and M to be the smallest and largest values for the internal codes values of the subtype. The minimum size L is determined as follows:

Value of m	Calculation of L - smallest positive integer such that:	Representation
$m \geq 0$	$M \leq 2^L - 1$	Unsigned
$m < 0$	$-2^{L-1} \leq m$ and $M \leq 2^{L-1} - 1$	Signed two's complement

Example

```
type COLOR is (RED, ORANGE, YELLOW, GREEN, AQUA, BLUE, VIOLET);
-- The minimum size of COLOR is 3 bits.

subtype SUNNY_COLOR is COLOR range ORANGE .. YELLOW;
-- The minimum size of COLOR is 2 bits.
-- because the internal code for YELLOW is 2
-- and (2**1)-1 <= 2 <= (2**2)-1

type TEMPERATURE is (FREEZING, COLD, MILD, WARM, HOT);
for TEMPERATURE use
          (FREEZING => -10,
               COLD =>   0,
               MILD =>  10,
               WARM =>  20,
                HOT =>  30);
    -- The minimum size of TEMPERATURE is 6 bits
    -- because with six bits we can represent signed
    -- integers between -32 and 31.
```

F 4.1.3 Size of an Enumeration Type

When no size specification is applied to an enumeration type, the objects of that type are represented as signed machine integers. The HP 9000 Series 600, 700, and 800 Computer Systems provides 8-, 16-, and 32-bit integers, and the compiler automatically selects the smallest signed machine integer that can hold all of the internal codes of the enumeration type. Thus, the default size for enumeration types with 128 or less elements is 8 bits, the default size for enumeration types with 129 to 32768 elements is 16 bits. Because this implementation does not support enumeration types with more than 32768 elements, a size specification or enumeration representation clause must be used for enumeration types that use a 32-bit representation.

When a size specification is applied to an enumeration type, this enumeration type and all of its subtypes have the size specified by the length clause. The size specification must specify a value greater than or equal to the minimum size of the type. Note that if the size specification specifies the minimum size and none of the internal codes are negative integers, the internal representation will be that of an unsigned type. Thus, when using a size specification of eight bits, you can have up to 256 elements in the enumeration type.

If the enumeration type is used as a component type in an array or record definition that is further constrained by a pragma PACK or a record representation clause, the size of this component will be determined by the pragma PACK or the record representation clause. This allows the array or record type to temporarily override any size specification that may have applied to the enumeration type.

The Ada compiler provides a complete implementation of size specifications. Nevertheless, because enumeration values are coded using integers, the specified length cannot be greater than 32 bits.

F 4.1.4 Alignment of an Enumeration Type

An enumeration type is byte-aligned if the size of the type is less than or equal to eight bits. An enumeration type is aligned on a 2-byte boundary (16 bit or half-word aligned) if the size of the type is in the range of 9..16 bits. An enumeration type is aligned on a 4-byte boundary (32 bit or word aligned) if the size of the type is in the range of 17..32 bits.

F 4.2 Integer Types

F 4.2.1 Predefined Integer Types

The HP 9000 Series 600, 700, and 800 Computer Systems provides these three predefined integer types:

```
type SHORT_SHORT_INTEGER
        is range -(2**7)  .. (2**7)-1;    --  8-bit signed
type SHORT_INTEGER
        is range -(2**15) .. (2**15)-1;   -- 16-bit signed
type INTEGER
        is range -(2**31) .. (2**31)-1;   -- 32-bit signed
```

An integer type declared by a declaration of the form

```
type T is range L .. U;
```

is implicitly derived from a predefined integer type. The compiler automatically selects the smallest predefined integer type whose range contains the values L to U, inclusive.

F 4.2.2 Internal Codes of Integer Values

The internal codes for integer values are represented using the two's complement binary method. The compiler does not represent integer values using any kind of a bias representation. Thus, one internal code will always represent the same literal value for any Ada integer type.

F 4.2.3 Minimum Size of an Integer Type or Subtype

The minimum size of an integer subtype is the minimum number of bits necessary for representing the internal codes of the subtype.

A static subtype of a null range has a minimum size of one. Otherwise, define m and M to be the smallest and largest values for the internal codes values of the subtype.

The minimum size L is determined as follows:

Value of m	**Calculation of L - smallest positive integer such that:**	**Representation**
$m >= 0$	$M \leq 2^L - 1$	Unsigned
$m < 0$	$-2^{L-1} \leq m$ and $M \leq 2^{L-1} - 1$	Signed two's complement

Example

```
type MY_INT is range 0 .. 31;
-- The minimum size of MY_INT is 5 bits using
-- an unsigned representation

subtype SOME_INT is MY_INT range 5 .. 7;
-- The minimum size of SOME_INT is 3 bits.
-- The internal representation of 7 requires three
-- binary bits using an unsigned representation.

subtype DYNAMIC_INT is MY_INT range L .. U;
-- Assuming that L and U are dynamic,
--(i.e. not known at compile time)
-- The minimum size of DYNAMIC_INT is the same as its base type,
-- MY_INT, which is 5 bits.

type ALT_INT is range -1 .. 16;
-- The minimum size of MY_INT is 6 bits,
-- because using a 5-bit signed integer we
-- can only represent numbers in the range -16 .. 15
-- and using a 6-bit signed integer we
-- can represent numbers in the range -32 .. 31
-- Since we must represent 16 as well as -1 the
-- compiler must choose a 6-bit signed representation
```

F 4.2.4 Size of an Integer Type

The sizes of the predefined integer types SHORT_SHORT_INTEGER, SHORT_INTEGER and INTEGER are 8, 16, and 32 bits, respectively.

When no size specification is applied to an integer type, the default size is that of the predefined integer type from which it derives, directly or indirectly.

Example

```
type S is range 80 .. 100;
-- Type S is derived from SHORT_SHORT_INTEGER
--  its default size is 8 bits.

type M is range 0 .. 255;
-- Type M is derived from SHORT_INTEGER
--  its default size is 16 bits.

type Z is new M range 80 .. 100;
-- Type Z is indirectly derived from SHORT_INTEGER
--   its default size is 16 bits.

type L is range 0 .. 99999;
-- Type L is derived from INTEGER
--  its default size is 32 bits.

type UNSIGNED_BYTE is range 0 .. (2**8)-1;
for UNSIGNED_BYTE'SIZE use 8;
-- Type UNSIGNED_BYTE is derived from SHORT_INTEGER
--  its actual size is 8 bits.

type UNSIGNED_HALFWORD is range 0 .. (2**16)-1;
for UNSIGNED_HALFWORD'SIZE use 16;
-- Type UNSIGNED_HALFWORD is derived from INTEGER
--  its actual size is 16 bits.
```

When a size specification is applied to an integer type, this integer type and all of its subtypes have the size specified by the length clause. The size specification must specify a value greater than or equal to the minimum size of the type. If the size specification specifies the minimum size and the lower bound of the range is not negative, the internal representation will be unsigned. Thus, when using a size specification of eight bits, you can represent an integer range from 0 to 255.

Using a size specification on an integer type allows you to define unsigned machine integer types. The compiler fully supports unsigned machine integer types that are either 8 or 16 bits. The 8-bit unsigned machine integer type is derived from the 16-bit predefined type SHORT_INTEGER. Using the 8-bit unsigned integer type in an expression results in it being converted to the predefined 16-bit signed type for use in the expression. This same method also applies to the 16-bit unsigned machine integer type, such that using the type in an expression results in a conversion to the predefined 32-bit signed type.

However, Ada does not allow the definition of an unsigned integer type that has a greater range than the largest predefined integer type. INTEGER is the largest predefined integer type and is represented as a 32-bit signed machine integer. Because the Ada language requires predefined integer types to be symmetric about zero (*Ada RM*, section 3.5.4), it is not possible to define a 32-bit unsigned machine integer type because the largest predefined integer type, INTEGER, is also a 32-bit type.

If the integer type is used as a component type in an array or record definition that is further constrained by a pragma PACK or record representation clause, the size of this component will be determined by the pragma PACK or record representation clause. This allows the array or record type to temporarily override any size specification that may have applied to the integer type.

The Ada compiler provides a complete implementation of size specifications. Nevertheless, because integers are coded using machine integers, the specified length cannot be greater than 32 bits.

F 4.2.5 Alignment of an Integer Type

An integer type is byte-aligned if the size of the type is less than or equal to eight bits. An integer type is aligned on a 2-byte boundary (16 bit or half-word aligned) if the size of the type is in the range of 9..16 bits. An integer type is aligned on a 4-byte boundary (32 bit or word aligned) if the size of the type is in the range of 17..32 bits.

F 4.2.6 Performance of an Integer Type

The type INTEGER is the most efficient of the integer types in Ada/800 because the hardware can access these integers and perform overflow checks on them with no additional cost. For the smaller integer types (SHORT_INTEGER and SHORT_SHORT_INTEGER), the compiler must emit additional instructions for access and overflow checking that increase both the execution time and the size of the generated code.

F 4.3 Floating Point Types

F 4.3.1 Predefined Floating Point Types

The HP 9000 Series 600/700/800 Computer Systems provides two predefined floating point types.

```
type FLOAT is digits 6 range
          -(2.0 - 2.0**(-23))*(2.0**127) ..
          +(2.0 - 2.0**(-23))*(2.0**127);
-- This expresses the decimal range -3.40282E+38 .. 3.40282E+38

type LONG_FLOAT is digits 15 range
          -(2.0 - 2.0**(-52))*(2.0**1023) ..
          +(2.0 - 2.0**(-52))*(2.0**1023);
-- This expresses the decimal range:
--  -1.797693134862315E+308 .. +1.797693134862315E+308
```

A floating point type declared by a declaration of the form

```
type T is digits D [range L .. U];
```

is implicitly derived from a predefined floating point type. The compiler automatically selects the smaller of the two predefined floating point types, FLOAT or LONG_FLOAT, whose number of digits is greater than or equal to D and that contains the values L to U inclusive.

F 4.3.2 Internal Codes of Floating Point Values

The internal codes for floating point values are represented using the IEEE standard formats for single precision and double precision floats.

The values of the predefined type FLOAT are represented using the single precision float format. The values of the predefined type LONG_FLOAT are represented using the double precision float format. The values of any other floating point type are represented in the same way as the values of the predefined type from which it derives, directly or indirectly.

The internal representation of the IEEE floating point types can be described by the following Ada specification

```
type BIT is range 0..1;
 for BIT'SIZE use 1;

--
--  IEEE representation for 32-bit FLOAT type
--

FLOAT32_BIAS : constant := 2**7-1;

type FLOAT32_EXPONENT is range 0 .. 2**8-1;
 for FLOAT32_EXPONENT'SIZE use 8;

type FLOAT32_MANTISSA is array(0..22) of BIT;
 for FLOAT32_MANTISSA'SIZE use 23;

type FLOAT32_REC is
   record
      SIGN_BIT    : BIT;
      EXPONENT    : FLOAT32_EXPONENT;
      MANTISSA    : FLOAT32_MANTISSA;
   end record;
 for FLOAT32_REC use
   record
      SIGN_BIT at  0 range  0 ..  0;
      EXPONENT at  0 range  1 ..  8;
      MANTISSA at  0 range  9 .. 31;
   end record;
 for FLOAT32_REC'SIZE use 32;

--
--  IEEE representation for 64-bit FLOAT type
--

FLOAT64_BIAS : constant := 2**10-1;

type FLOAT64_EXPONENT is range 0 .. 2**11-1;
```

```
 for FLOAT64_EXPONENT'SIZE use 11;

type FLOAT64_MANTISSA is array(0..51) of BIT;
 for FLOAT64_MANTISSA'SIZE use 52;

type FLOAT64_REC is
   record
      SIGN_BIT    : BIT;
      EXPONENT    : FLOAT64_EXPONENT;
      MANTISSA    : FLOAT64_MANTISSA;
   end record;
 for FLOAT64_REC use
   record
      SIGN_BIT at  0 range   0 ..  0;
      EXPONENT at  0 range   1 .. 11;
      MANTISSA at  0 range  12 .. 63;
   end record;
 for FLOAT64_REC'SIZE use 64;
```

F 4.3.3 Minimum Size of a Floating Point Type or Subtype

The minimum size of a floating point subtype is 32 bits if its base type is FLOAT or a type derived from FLOAT; it is 64 bits if its base type is LONG_FLOAT or a type derived from LONG_FLOAT.

F 4.3.4 Size of a Floating Point Type

The only size that can be specified for a floating point type in a size specification is its default size (32 or 64 bits).

F 4.3.5 Alignment of a Floating Point Type

A floating point type FLOAT is aligned on a 4-byte boundary (32 bit or word aligned). The floating point type LONG_FLOAT is aligned on an 8-byte boundary (64 bit or double-word aligned).

F 4.4 Fixed Point Types

F 4.4.1 Predefined Fixed Point Types

To implement fixed point types, the HP 9000 Series 600, 700, and 800 Computer System provides a set of three anonymous predefined fixed point types of this form:

4

```
type SHORT_FIXED is delta D range
                -(2**7)*SMALL .. +((2**7)-1)*SMALL;
for  SHORT_FIXED'SMALL use SMALL;
for  SHORT_FIXED'SIZE use 8;

type FIXED is delta D range
                -(2**15)*SMALL .. +((2**15)-1)*SMALL;
for FIXED'SMALL use SMALL;
for FIXED'SIZE use 16;

type LONG_FIXED is delta D range
                -(2**31)*SMALL .. +((2**31)-1)*SMALL;
for LONG_FIXED'SMALL use SMALL;
for LONG_FIXED'SIZE use 32;

-- In the above type definitions SMALL is the largest
-- power of two that is less than or equal to D.
```

A fixed point type declared by a declaration of the form

```
type T is delta D range L .. U;
```

is implicitly derived from one of the predefined fixed point types.

The compiler automatically selects the smallest predefined fixed point type using the following method:

- Choose the largest power of two that is not greater than the value specified for the delta to use as SMALL.
- Determine the ranges for the three predefined fixed point types using the value obtained for SMALL.
- Select the smallest predefined fixed point type whose range contains the values L+SMALL to U-SMALL, inclusive.

Using the above method, it is possible that the values L and U lie outside the range of the compiler-selected fixed point type. For this reason, the values used in a fixed point range constraint should be expressed as follows, to guarantee that the values of L and U are representable in the resulting fixed point type:

```
type ANY_FIXED is delta D range L-D .. U+D;
-- The values of L and U are guaranteed to be
-- representable in the type ANY_FIXED.
```

F 4.4.2 Internal Codes of Fixed Point Values

The internal codes for fixed point values are represented using the two's complement binary method as integer multiples of 'SMALL. The value of a fixed point object is 'SMALL multiplied by the stored internal code.

F 4.4.3 Small of a Fixed Point Type

The Ada compiler requires that the value assigned to 'SMALL is always a power of two. Ada does not support a length clause that specifies a 'SMALL for a fixed point type that is not a power of two.

If a fixed point type does not have a length clause that specifies the value to use for 'SMALL, the value of 'SMALL is determined by the compiler according to the rules in the *Ada RM*, section 3.5.9.

F 4.4.4 Minimum Size of a Fixed Point Type or Subtype

The minimum size of a fixed point subtype is the minimum number of binary digits necessary to represent the values in the range of the subtype using the 'SMALL of the base type.

A static subtype of a null range has a minimum size of one. Otherwise, define s and S to be the bounds of the subtype, define m and M to be the smallest and greatest model numbers of the base type, and let i and I be the integer representations for the model numbers m and M. The following axioms hold:

4

```
s <=  m < M <= S
m  - T'BASE'SMALL <= s
M  + T'BASE'SMALL >= S
M  = T'BASE'LARGE
i  =  m  / T'BASE'SMALL
I  =  M  / T'BASE'SMALL
```

The minimum size L is determined as follows:

Value of i	Calculation of L - smallest positive integer such that:	Representation
$i \geq 0$	$I \leq 2^L - 1$	Unsigned
$i < 0$	$-2^{L-1} \leq i$ and $I \leq 2^{L-1} - 1$	Signed two's complement

Example

```
type UF is delta 0.1 range 0.0 .. 100.0;
-- The value used for 'SMALL is 0.0625
-- The minimum size of UF is 11 bits,
-- seven bits before the decimal point
-- four bits after the decimal point
-- and no bits for the sign.

type SF is delta 16.0 range -400.0 .. 400.0;
-- The minimum size of SF is 6 bits,
-- nine bits to represent the range 0 to 511
-- less four bits by the implied decimal point of 16.0
-- and one bit for the sign.

subtype UFS is UF delta 4.0 range 0.0 .. 31.0;
-- The minimum size of UFS is 9 bits,
-- five bits to represent the range 0 to 31
-- four bits for the small of 0.0625 from the base type
-- and no bits for the sign.

subtype SFD is SF range X .. Y;
-- Assuming that X and Y are not static, the minimum size
-- of SFD is 6 bits. (the same as its base type)
```

F 4.4.5 Size of a Fixed Point Type

The sizes of the anonymous predefined fixed point types SHORT_FIXED, FIXED, and LONG_FIXED are 8, 16, and 32 bits, respectively.

When no size specification is applied to a fixed point type, the default size is that of the predefined fixed point type from which it derives, directly or indirectly.

Example

```
type Q is delta 0.01 range 0.00 .. 1.00;
-- Type Q is derived from an 8-bit predefined
-- fixed point type, its default size is 8 bits.

type R is delta 0.01 range 0.00 .. 2.00;
-- Type R is derived from a 16-bit predefined
-- fixed point type, its default size is 16 bits.

type S is new R range 0.00 .. 1.00;
-- Type S is indirectly derived from a 16-bit predefined
-- fixed point type, its default size is 16 bits.

type SF is delta 16.0 range -400.0 .. 400.0;
for SF'SIZE use 6;
-- Type SF is derived from an 8-bit predefined
-- fixed point type, its actual size is 6 bits.

type UF is delta 0.1 range 0.0 .. 100.0;
for UF'SIZE use 11;
-- Type UF is derived from a 16-bit predefined
-- fixed point type, its actual size is 11 bits.
-- The value used for 'SMALL is 0.0625
```

When a size specification is applied to a fixed point type, this fixed point type and all of its subtypes have the size specified by the length clause. The size specification must specify a value greater than or equal to the minimum size of the type. If the size specification specifies the minimum size and the lower bound of the range is not negative, the internal representation will be that of an unsigned type.

If the fixed point type is used as a component type in an array or record definition that is further constrained by a pragma PACK or record representation clause, the size of this component will be determined by the pragma PACK or record representation clause. This allows the array or record type to temporarily override any size specification that may have applied to the fixed point type.

The Ada compiler provides a complete implementation of size specifications. Nevertheless, because fixed point objects are coded using machine integers, the specified length cannot be greater than 32 bits.

F 4.4.6 Alignment of a Fixed Point Type

A fixed point type is byte-aligned if the size of the type is less than or equal to eight bits. A fixed point type is aligned on a 2-byte boundary (16 bit or half-word aligned) if the size of the type is in the range of 9..16 bits. A fixed point type is aligned on a 4-byte boundary (32 bit or word aligned) if the size of the type is in the range of 17..32 bits.

F 4.5 Access Types

F 4.5.1 Internal Codes of Access Values

In the program generated by the compiler, access values are represented using 32-bit machine addresses. The predefined generic function UNCHECKED_CONVERSION can be used to convert the internal representation of an access value into any other 32-bit type. You can also use UNCHECKED_CONVERSION to assign any 32-bit value into an access value. When interfacing with externally supplied data structures, it may be necessary to use the generic function UNCHECKED_CONVERSION to convert a value of the type SYSTEM.ADDRESS into the internal representation of an access value. Programs that use UNCHECKED_CONVERSION in this manner *cannot* be considered portable across different implementations of Ada.

F 4.5.2 Collection Size for Access Types

A length clause that specifies the collection size is allowed for an access type. This collection size applies to all objects of this type and any type derived from this type, as well as any and all subtypes of these types. Thus, a length clause that specifies the collection size is only allowed for the original base type definition and not for any subtype or derived type of the base type.

When no specification of collection size applies to an access type, the attribute STORAGE_SIZE returns zero. In this case, the compiler will dynamically manage the storage for the access type and it is not possible to determine directly the amount of storage available in the collection for the access type.

The recommended format of a collection size length clause is:

```
U_NUM:  constant := 50; -- The maximum number
--                         of elements needed
U_SIZE: constant := <U size>; -- Substitute the value
                              -- of U'SIZE here
--
-- The constant U_SIZE should also be:
--  1. a multiple of two
--  2. greater than or equal to four
--
--  Additionally, the type U must have a static size
--
type P is access U;  -- Type U is any
--                      non-dynamic user defined type.
for P'STORAGE_SIZE use (U_SIZE*U_NUM)+4;
```

In the above example we have specified a collection size that is large enough to contain 50 objects of the type U. There is a constant overhead of four bytes for each storage collection. Because the collection manager rounds the element size to be a multiple of two that is four or greater. you must ensure that U_SIZE is the smallest multiple of two that is greater than or equal to U'SIZE and is greater than or equal to four.

You can also provide a length clause that specifies the collection size for a type that has a dynamic size. It is only possible to specify an upper limit on the amount of memory that can be used by all instances of objects that are of this dynamic type. Because the size is dynamic, you cannot specify the number of elements in the collection.

F 4.5.3 Minimum Size of an Access Type or Subtype

The minimum size of an access type is always 32 bits.

F 4.5.4 Size of an Access Type

The size of an access type is 32 bits, the same as its minimum size.

The only size that can be specified for an access type in a size specification clause is its usual size (32 bits).

F 4.5.5 Alignment of an Access Type

An access type is aligned on a 4-byte boundary (32 bit or word-aligned).

F 4.6 Task Types

F 4.6.1 Internal Codes of Task Values

In the program generated by the compiler, task type objects are represented using 32-bit machine addresses.

4

F 4.6.2 Storage for a Task Activation

The value returned by the attribute `'STORAGE_SIZE` has three cases.

- For a task type without a length clause and using the default storage size at bind time, the attribute `'STORAGE_SIZE` returns the default task storage size.
- For a task type without a length clause and using the bind-time option `-W b,-t,`*nnn* to set the task storage size, the attribute `'STORAGE_SIZE` returns *nnn**1024.
- For a task type with a length clause, the attribute `'STORAGE_SIZE` returns the value used in the length clause.

When a length clause is used on a task type it specifies the number of storage units reserved for an activation of a task of this type (Ada RM 13.2 (10)). This space includes both the task stack space and a private data section of approximately 5400 bytes. The private data section contains the Task Control Block which has information used by the Ada runtime to manage the task. The size specified in the length clause must be greater than this minimum size. otherwise a TASKING_ERROR exception will be generated during the elaboration of the activation of a task object of this type. The stack space requirements for the task object must also be considered. If the stack space is insufficient during the execution of the task, the exception STORAGE_ERROR will be raised and the task object will be terminated.

An example that sets the storage usage for a task type that needs 4K bytes of stack space:

```
task type MY_TASK_TYPE is
      entry START;
      entry STOP;
   end MY_TASK_TYPE;

for MY_TASK_TYPE'STORAGE_SIZE use 5400 + (4 * 1024);
    -- Allocates a 4K stack.
```

4

F 4.6.3 Minimum Size of a Task Stack

The task object will use 800 bytes of stack space in the first stack frame. Some additional stack space is required to make calls into the Ada runtime. The smallest value that can be safely used for a task with minimal stack needs is approximately 2000 bytes. If the task object has local variables or if it makes calls to other subprograms, the stack storage requirements will be larger. The actual amount of stack space used by a task will need to be determined by trial and error. If a tasking program raises STORAGE_ERROR or behaves abnormally, you should increase the stack space for the tasks.

F 4.6.4 Limitation on Length Clause for Derived Task Types

If a task type has a storage size length clause, the storage size applies to all task objects of this type and any task type derived from this type. Thus, a length clause that specifies the storage size is only allowed for the original task type definition and not for any derived task type.

F 4.6.5 Minimum Size of a Task Type or Subtype

The minimum size of a task type is always 32 bits.

F 4.6.6 Size of a Task Type

The size of a task type is 32 bits, the same as its minimum size.

The only size that can be specified for a task type in a size specification clause is its usual size (32 bits).

F 4.6.7 Alignment of a Task Type

A task type is aligned on a 4-byte boundary (32 bit or word aligned).

F 4.7 Array Types

F 4.7.1 Layout of an Array

Each array is allocated in a contiguous area of storage units. All the components have the same size. A gap may exist between two consecutive components (and after the last component). All the gaps are the same size, as shown in Figure 4-1.

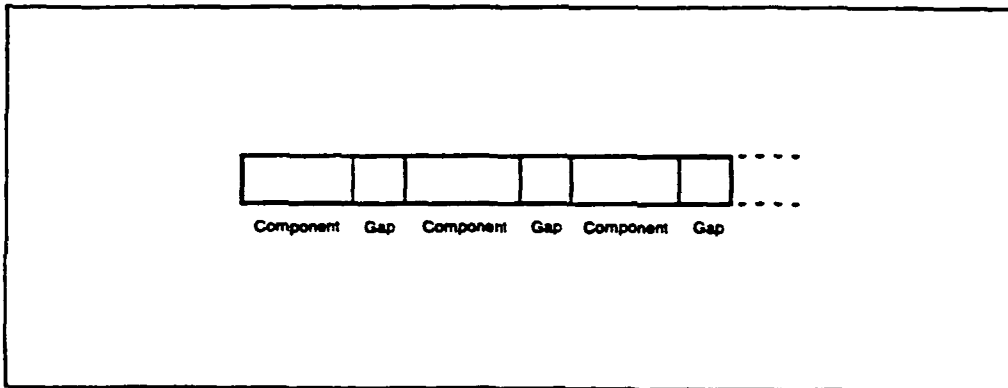


Figure 4-1. Layout of an Array

F 4.7.2 Array Component Size and Pragma PACK

If the array is not packed, the size of each component is the size of the component type. This size is the default size of the component type unless a size specification applies to the component type.

If the array is packed and the array component type is neither a record nor array type, the size of the component is the minimum size of the component type. The minimum size of the component type is used even if a size specification applies to the component type.

Packing the array has no effect on the size of the components when the component type is a record or array type.

Example

```
type A is array(1..8) of BOOLEAN;
-- The component size of A is the default size
-- of the type BOOLEAN: 8 bits.

type B is array(1..8) of BOOLEAN;
pragma PACK(B);
-- The component size of B is the minimum size
-- of the type BOOLEAN: 1 bit.

type DECIMAL_DIGIT is range 0..9;
-- The default size for DECIMAL_DIGIT is 8 bits
-- The minimum size for DECIMAL_DIGIT is 4 bits

type BCD_NOT_PACKED is array(1..8) of DECIMAL_DIGIT;
-- The component size of BCD_NOT_PACKED is the default
-- size of the type DECIMAL_DIGIT: 8 bits.

type BCD_PACKED is array(1..8) of DECIMAL_DIGIT;
pragma PACK(BCD_PACKED);
-- The component size of BCD_PACKED is the minimum
-- size of the type DECIMAL_DIGIT: 4 bits.
```

F 4.7.3 Array Gap Size and Pragma PACK

If the array type is not packed and the component type is a record type without a size specification clause. the compiler may choose a representation for the array with a gap after each component. Inserting gaps optimizes access to the array components. The size of the gap is chosen so that each array component begins on an alignment boundary.

If the array type is packed, the compiler will generally not insert a gap between the array components. In such cases, access to array components can be slower

because the array components will not always be aligned correctly. However, in the specific case where the component type is a record and the record has a record representation clause specifying an alignment, the alignment will be honored and gaps may be inserted in the packed array type.

Example

```
type R is
   record
      K : INTEGER;    -- Type Integer is word aligned.
      B : BOOLEAN;    -- Type Boolean is byte aligned.
   end record;
-- Record type R is word aligned.  Its size is 40 bits.

type A is array(1..10) of R;
-- A gap of three bytes is inserted after each array
-- component in order to respect the alignment of type R.
-- The size of array type A is 640 bits.

type PA is array(1..10) of R;
pragma PACK(PA);
-- There are no gaps in an array of type PA because
-- of the pragma PACK statement on type PA.
-- The size of array type PA is 400 bits.

type NR is new R;
for NR'SIZE use 40;

type B is array(1..10) of NR;
-- There are no gaps in an array of type B because
-- of the size specification clause on type NR.
-- The size of array type B is 400 bits.
```

F 4.7.4 Size of an Array Type or Subtype

The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the component and the size of the gap.

The size of an array type or subtype cannot be computed at compile time if any of the following are true:

- If the array has non-static constraints or if it is an unconstrained type with non-static index subtypes (because the number of components can then only be determined at run time)
- If the components are records or arrays and their constraints or the constraints of their subcomponents are not static (because the size of the components and the size of the gaps can then only be determined at run time). Pragma PACK is not allowed in this case.

As indicated above, the effect of a pragma PACK on an array type is to suppress the gaps and to reduce the size of the components, if possible. The consequence of packing an array type is thus to reduce its size.

Array packing is fully implemented by the Ada compiler with this limitation: if the components of an array type are records or arrays and their constraints or the constraints of their subcomponents are not static, the compiler ignores any pragma PACK statement applied to the array type and issues a warning message.

A size specification applied to an array type has no effect. The only size that the compiler will accept in such a length clause is the usual size. Nevertheless, such a length clause can be used to verify that the layout of an array is as expected by the application.

F 4.7.5 Alignment of an Array Type

If no pragma PACK applies to an array type and no size specification applies to the component type, the array type is aligned as the component type would have been.

If a pragma PACK applies to an array type or if a size specification applies to the component type (so that there are no gaps), the alignment of the array type is as given in Table 4-2

4

Table 4-2. Alignment and Pragma PACK

Component (Normal Alignment)	Component (Alignment Within the Array)				
	Double-Word	**Word**	**Half-Word**	**Byte**	**Bit**
Double-Word	Double-Word	Word	Half-Word	Byte	Bit
Word	Word	Word	Half-Word	Byte	Bit
Half-Word	Half-Word	Half-Word	Half-Word	Byte	Bit
Byte	Byte	Byte	Byte	Byte	Bit
Bit	Bit	Bit	Bit	Bit	Bit

F 4.8 Record Types

Syntax (record representation clause)

for *record-type-name* **use**
  record [*alignment-clause*]
    [*component-clause*]
    ...
  end record;

4

Syntax (alignment clause)

at mod *static-expression*

Syntax (component clause)

record-component-name **at** *static-expression*
    range *static-expression* .. *static-expression* ;

F 4.8.1 Layout of a Record

A record is allocated in a contiguous area of storage units. The size of a record depends on the size of its components and the size of any gaps between the components. The compiler may add additional components to the record. These components are called implicit components.

The positions and sizes of the components of a record type object can be controlled using a record representation clause as described in the *Ada RM*, section 13.4. If the record contains compiler-generated implicit components, their position also can be controlled using the proper component clause. For more details, see section "F 4.8.6 Implicit Components". In the implementation for the HP 9000 Series 600, 700, and 800 Computer Systems, there is no restriction on the position that can be specified for a component of a record. If the component is not a record or an array, its size can be any size from the minimum size to the default size of its base type. If the component is a record or an array, its size must be the size of its base type.

Example (Record with a representation clause):

```
type PSW_BIT is new BOOLEAN;
for PSW_BIT'SIZE use 1;

type CARRY_BORROW is array (1..8) of PSW_BIT;
pragma PACK (CARRY_BORROW);

FIRST_BYTE  : constant := 0;
CABO_BYTE   : constant := 1;
THIRD_BYTE  : constant := 2;
SYSMASK_BYTE: constant := 3;

type PSW is
   record
      T  : PSW_BIT;
      H  : PSW_BIT;
      L  : PSW_BIT;
      N  : PSW_BIT;
      X  : PSW_BIT;
      B  : PSW_BIT;
      C  : PSW_BIT;
      V  : PSW_BIT;
      M  : PSW_BIT;
      CB : CARRY_BORROW;
      R  : PSW_BIT;
      Q  : PSW_BIT;
      P  : PSW_BIT;
      D  : PSW_BIT;
      I  : PSW_BIT;
   end record;

-- This type can be used to map the status register of
-- the HP-PA processor.
```

```
for PSW use
   record at mod 4;
      T  at FIRST_BYTE   range 7..7;
      H  at SECOND_BYTE  range 0..0;
      L  at SECOND_BYTE  range 1..1;
      N  at SECOND_BYTE  range 2..2;
      X  at SECOND_BYTE  range 3..3;
      B  at SECOND_BYTE  range 4..4;
      C  at SECOND_BYTE  range 5..5;
      V  at SECOND_BYTE  range 6..6;
      M  at SECOND_BYTE  range 7..7;
      CB at CABO_BYTE    range 0..7;
      R  at SYSMASK_BYTE range 3..3;
      Q  at SYSMASK_BYTE range 4..4;
      P  at SYSMASK_BYTE range 5..5;
      D  at SYSMASK_BYTE range 6..6;
      I  at SYSMASK_BYTE range 7..7;
   end record;
```

In the above example, the record representation clause explicitly tells the compiler both the position and size for each of the record components. The optional alignment clause specifies a 4-byte alignment for this record. In this example every component has a corresponding component clause, although it is not required. If one is not supplied, the choice of the storage place for that component is left to the compiler. If component clauses are given for all components, including any implicit components, the record representation clause completely specifies the representation of the record type and will be obeyed exactly by the compiler.

F 4.8.2 Bit Ordering in a Component Clause

The HP Ada compiler for the HP 9000 Series 800 Computer System numbers the bits in a component clause starting from the most significant bit. Thus, bit zero represents the most significant bit of an 8-bit byte and bit seven represents the least significant bit of the byte.

F 4.8.3 Value used for SYSTEM.STORAGE_UNIT

The smallest directly addressable unit on the HP 9000 Series 600/700/800 Computer Systems is the 8-bit byte. This is the value used for SYSTEM.STORAGE_UNIT that is implicitly used in a component clause. A component clause specifies an offset and a bit range. The offset in a component clause is measured in units of SYSTEM.STORAGE_UNIT, which for the HP 9000 Series 600/700/800 Computer Systems is an 8-bit byte.

The compiler determines the actual bit address for a record component by combining the byte offset with the bit range. There are several different ways to refer to the same bit address. In the following example, each of the component clauses refer to the same bit address.

Example

```
COMPONENT at 0  range  16 .. 18;
COMPONENT at 1  range   8 .. 10;
COMPONENT at 2  range   0 .. 2;
```

F 4.8.4 Compiler-Chosen Record Layout

If no component clause applies to a component of a record, its size is the size of the base type. Its location in the record layout is chosen by the compiler so as to optimize access to the component. That is, each component of a record follows the natural alignment of the component's base type. Moreover, the compiler chooses the position of the components to reduce the number of gaps or holes in the record and additionally to reduce the size of the record.

Because of these optimizations, there is no connection between the order of the components in a record type declaration and the positions chosen by the compiler for the components in a record object.

F 4.8.5 Change in Representation

It is possible to apply a record representation clause to a derived record type. This allows a record type to possibly have several alternative representations. Thus, the compiler fully supports the "Change in Representation" as described in the *Ada RM*, section 13.6.

F 4.8.6 Implicit Components

In some circumstances, access to a record object or to a component of a record object involves computing information that only depends on the discriminant values or on a value that is known only at run time. To avoid unnecessary recomputation, the compiler reserves space in the record to store this information. The compiler will update this information whenever a discriminant on which it depends changes. The compiler uses this information whenever the component that depends on this information is accessed. This information is stored in special components called implicit components. There are three different kinds of implicit components:

- Components that contain an offset value.
- Components that contain information about the record object.
- Components that are descriptors.

Implicit components that contain an offset value from the beginning of the record are used to access indirect components. Implicit components of this kind are called *offset components*. The compiler introduces implicit offset components whenever a record contains indirect components. These implicit components are considered to be declared before any variant part in the record type definition. Implicit components of this kind *cannot* be suppressed by using the pragma IMPROVE.

Implicit components that contain information about the record object are used when the record object or component of a record object is accessed. Implicit components of this kind are used to make references to the record object or to make record components more efficient. These implicit components are considered to be declared before any variant part in the record type definition. There are two implicit components of this kind: RECORD_SIZE and VARIANT_INDEX. Implicit components of this kind *can* be suppressed by using the pragma IMPROVE.

The third kind of implicit components are descriptors that are used when accessing a record component. The implicit component exists whenever the record has an array or record component that depends on a discriminant of the record. An implicit component of this kind is considered to be declared immediately before the record component it is associated with. There are two implicit components of this kind: ARRAY_DESCRIPTOR and RECORD_DESCRIPTOR. Implicit components of this kind *cannot* be suppressed by using the pragma IMPROVE.

Note The -S option (Assembly Option) to the ada(1) command is useful for finding out what implicit components are associated with the record type. This option will detail the exact representation for all record types defined in a compilation unit.

F 4.8.7 Indirect Components

If the offset of a component cannot be computed at compile time, the compiler will reserve space in the record for the computed offset. The compiler computes the value to be stored in this offset at run time. A component that depends on a run time computed offset is said to be an indirect component, while other components are said to be direct.

A pictorial example of a record layout with an indirect component is shown in Figure 4-2.

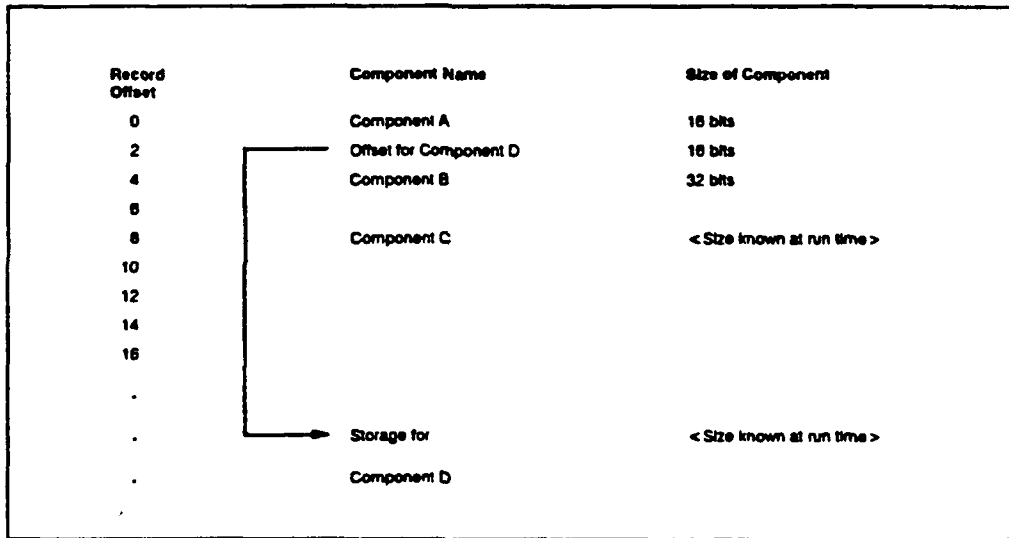


Figure 4-2. Record layout with an Indirect Component

In the above example. the component D has an offset that cannot be computed at compile time. The compiler then will reserve space in the record to store the computed offset and will store this offset at run time. The other components (A, B, and C) are all direct components because their offsets can all be computed at compile time.

F 4.8.8 Dynamic Components

If a record component is a record or an array, the size of the component may need to be computed at run time and may depend on the discriminants of the record. These components are called *dynamic components.*

Example (Record with dynamic components):

```
type U_RNG is range 0..255;

type UC_ARRAY is array(U_RNG range <>) of INTEGER;

--
-- The type GRAPH has two dynamic components: X and Y.
--
type GRAPH (X_LEN, Y_LEN: U_RNG) is
   record
      X : UC_ARRAY(1 .. X_LEN);
                        -- The size of X depends on X_LEN
      Y : UC_ARRAY(1 .. Y_LEN);
                        -- The size of Y depends on Y_LEN
   end record;

type DEVICE is (SCREEN, PRINTER);

type COLOR  is (GREEN, RED, BLUE);

Q : U_RNG;
```

```
--
-- The type PICTURE has two dynamic components: R and T.
--
type PICTURE (N : U_RNG; D : DEVICE) is
   record
      R  : GRAPH(N,N);  -- The size of R depends on N
      T  : GRAPH(Q,Q);  -- The size of T depends on Q
      case D is
         when SCREEN  => C : COLOR;
         when PRINTER => null;
      end case;
   end record;
```

4

Any component that is placed after a dynamic component has an offset that cannot be evaluated at compile time and is thus indirect. To minimize the number of indirect components, the compiler groups the dynamic components and places them at the end of the record. Due to this strategy, *the only indirect components are dynamic components.* However, all dynamic components are not necessarily indirect. The compiler can usually compute the offset of the first dynamic component and thus it becomes a direct component. Any additional dynamic components are then indirect components.

A pictorial example of the data layout for the record type PICTURE is shown in Figure 4-3.

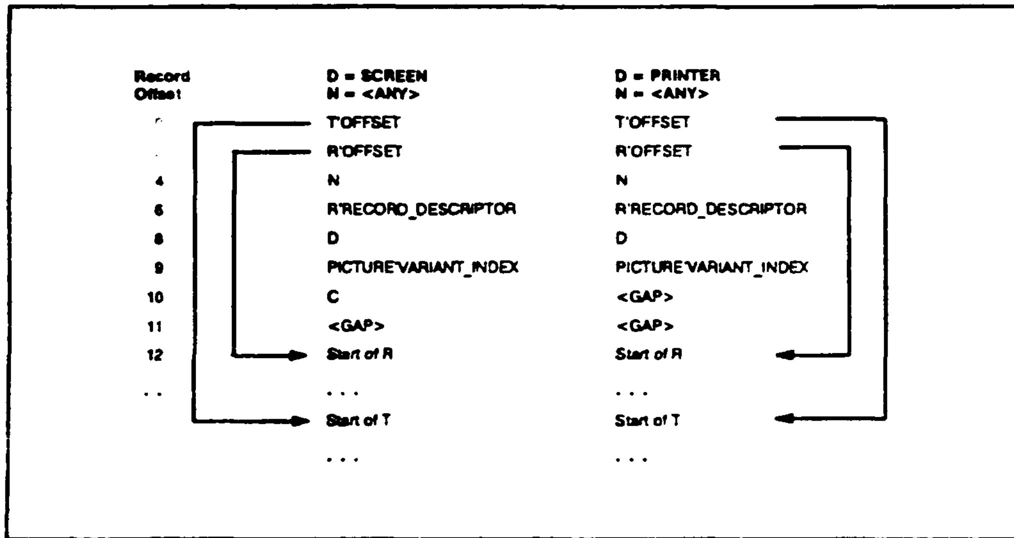


Figure 4-3. Example of a Data Layout

F 4.8.9 Representation of the Offset of an Indirect Component

The offset of an indirect component is always expressed in storage units, which for the HP 9000 Series 600, 700, and 800 Computer System are bytes. The space that the compiler reserves for the offset of an indirect component must be large enough to store the maximum potential offset. The compiler will choose the size of an offset component to be either an 8-, 16-, or 32-bit object. It is possible to further reduce the size in bits of this component by specifying it in a component clause.

If C is the name of an indirect component, the offset of this component can be denoted in a component clause by the implementation-generated name C'OFFSET.

4

Example (Record representation clause for the type GRAPH)

```
for GRAPH use
   record
      X_LEN    at 0 range 0..7;
      Y_LEN    at 1 range 0..7;
      X'OFFSET at 2 range 0..15;
    end record;
--
-- The bit range for the implicit component
-- X'OFFSET could have been specified as 0..11
-- This would make access to X much slower
--
```

In this example we have used a component clause to specify the location of an offset for a dynamic component. In this example the compiler will choose Y to be the first dynamic component and as such it will have a static offset. The component X will be placed immediately after the end of component Y by the compiler at run time. At run time the compiler will store the offset of this location in the field X'OFFSET. Any references to X will have additional code to compute the run time address of X using the X'OFFSET field. References to Y will be direct references.

F 4.8.10 The Implicit Component RECORD_SIZE

This implicit component is created by the compiler whenever a record with discriminants has a variant part and the discriminant that defines the variant part has a default expression (that is, a record type that possibly could be unconstrained.) The component 'RECORD_SIZE contains the size of the storage space required to represent the current variant of the record object. Note that the actual storage allocated for the record object may be more than this.

The value of a RECORD_SIZE component may denote a number of bits or a number of storage units (bytes). In most cases it denotes a number of storage units (bytes), but if any component clause specifies that a component of the record type has an offset or a size that cannot be expressed using storage units, the value designates a number of bits.

The implicit component RECORD_SIZE must be large enough to store the maximum size that the record type can attain. The compiler evaluates this size, calls it MS, and considers the type of RECORD_SIZE to be an anonymous integer type whose range is 0 .. MS.

If R is the name of a record type, this implicit component can be denoted in a component clause by the implementation-generated name R'RECORD_SIZE.

F 4.8.11 The Implicit Component VARIANT_INDEX

This implicit component is created by the compiler whenever the record type has a variant part. It indicates the set of components that are present in a record object. It is used when a discriminant check is to be done.

Within a variant part of a record type, the compiler numbers component lists that themselves do not contain a variant part. These numbers are the possible values for the implicit component VARIANT_INDEX. The compiler uses this number to determine which components of the variant record are currently valid.

Example (Record with a variant part):

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

type DESCRIPTION( KIND : VEHICLE := CAR) is
   record
      SPEED : INTEGER;
      case KIND is
         when AIRCRAFT | CAR =>
            WHEELS : INTEGER;
            case KIND is
               when AIRCRAFT =>      -- VARIANT_INDEX is 1
                  WINGSPAN : INTEGER;
               when others =>        -- VARIANT_INDEX is 2
                  null;
            end case;
         when BOAT =>                -- VARIANT_INDEX is 3
            STEAM : BOOLEAN;
         when ROCKET =>              -- VARIANT_INDEX is 4
            STAGES : INTEGER;
      end case;
   end record;
```

In the above example, the value of the variant index indicates which of the components are present in the record object; these components are summarized in the table below.

Variant Index	Legal Components
1	KIND, SPEED, WHEELS, WINGSPAN
2	KIND, SPEED, WHEELS
3	KIND, SPEED, STEAM
4	KIND, SPEED, STAGES

The implicit component VARIANT_INDEX must be large enough to store the number of component lists that do not contain variant parts. The compiler evaluates this size, calls it VS, and considers the type of VARIANT_INDEX to be an anonymous integer type whose range is 0 .. VS.

If R is the name of a record type, this implicit component can be denoted in a component clause by the implementation-generated name R'VARIANT_INDEX.

4

F 4.8.12 The Implicit Component ARRAY_DESCRIPTOR

An implicit component of this kind is associated by the compiler with each record component whose type is an array that has bounds that depend on a discriminant of the record.

The structure and contents of the implicit component ARRAY_DESCRIPTOR are not described in this manual. Nevertheless, if you are interested in specifying the location of a component of this kind in a component clause, you can obtain the size of the component by supplying the -S option (Assembly Option) to the ada(1) command.

If C is the name of a record component that conforms to the above definition, this implicit component can be denoted in a component clause by the implementation-generated name C'ARRAY_DESCRIPTOR.

F 4.8.13 The Implicit Component RECORD_DESCRIPTOR

An implicit component of this kind may be associated by the compiler when a record component is a record type that has components whose size depends on a discriminant of the outer record.

The structure and content of the implicit component RECORD_DESCRIPTOR are not described in this manual. Nevertheless, if you are interested in specifying the location of a component of this kind in a component clause, you can obtain the size of the component by applying the -S option (Assembly Option) to the ada(1) command.

If C is the name of a record component that conforms to the above definition, this implicit component can be denoted in a component clause by the implementation-generated name C'RECORD_DESCRIPTOR.

F 4.8.14 Suppression of Implicit Components

Ada provides the capability of suppressing the implicit components RECORD_SIZE and VARIANT_INDEX from a record type. This can be done using an implementation defined pragma called IMPROVE.

4

Syntax

```
pragma IMPROVE ( TIME | SPACE , [ON =>] record_type_name );
```

The first argument specifies whether TIME or SPACE is the primary criterion for the choice of representation of the record type that is denoted by the second argument.

If TIME is specified, the compiler inserts implicit components as described above. This is the default behavior of the compiler. If SPACE is specified, the compiler only inserts a VARIANT_INDEX component or a RECORD_SIZE component if a component clause for one of these components was supplied. If the record type has no record representation clause, both components will be suppressed. Thus, a record representation clause can be used to keep one implicit component while suppressing the other.

A pragma IMPROVE that applies to a given record type can occur anywhere that a record representation clause is allowed for this type.

F 4.8.15 Size of a Record Type or Subtype

The compiler generally will round up the size of a record type to a whole number of storage units (bytes). If the record type has a component clause that specifies a record component that cannot be expressed in storage units, the compiler will not round up and instead the record size will be expressed as an exact number of bits.

The size of a constrained record type is obtained by adding the sizes of its components and the sizes of its gaps (if any). The size of a constrained record will not be computed at compile time if:

- The record type has non-static constraints.
- A component is an array or record and its size cannot be computed at compile time (that is, if the component has non-static constraints.)

The size of an unconstrained record type is the largest possible size that the unconstrained record type could assume, given the constraints of the discriminant or discriminants. If the size of any component cannot be evaluated exactly at compile time, the compiler will use the maximum size that the component could possibly assume to compute the size of the unconstrained record type.

A size specification applied to a record type has no effect. The only size that the compiler will accept in such a length clause is the usual size. Nevertheless, such a length clause can be used to verify that the layout of a record is as expected by the application.

F 4.8.16 Size of an Object of a Record Type

A record object of a constrained record type has the same size as its base type.

A record object of an unconstrained record type has the same size as its base type if this size is less than or equal to 24576 bytes. The size of the base type is the largest possible size that the unconstrained record type could assume, given the constraints of the discriminant(s). If the size of the base type is larger than 24576 bytes, the record object only has the size necessary to store its current value. Storage space is then allocated and deallocated dynamically based on the current value of the discriminant or discriminants.

F 4.8.17 Alignment of a Record Subtype

When a record type does *not* have a record representation clause, or when a record type has a record representation clause *without* an alignment clause, the record type is word-aligned to the maximum alignment required by any component of the record (8, 16, 32, or 64 bits). Any subtypes of a record type also have the same alignment as their base type.

For a record type that has a record representation clause *with* an alignment clause, any subtypes of this record type also obey the alignment clause.

An alignment clause can specify that a record type is byte, half-word, word, or double-word aligned (specified as 1, 2, 4, or 8 bytes). Ada does not support alignments larger than an 8-byte alignment.

F 4.9 Data Allocation

This section explains how data objects are allocated by the HP Ada compiler for the HP 9000 Series 600, 700, and 800 Computer System.

Data objects are allocated into:

- A stack frame.
- The global data area.
- The heap.

or have no storage allocated to them.

A stack frame is used for objects declared within a subprogram or task body, or within a declare block. The stack frame contains the data objects that are dynamic if each invocation of the subprogram or block creates a new data object. Each object allocated in a stack frame has a lifetime that begins after the elaboration of the subprogram or block enclosing the object and ends after the subprogram or the block is exited.

The global data area is used for objects declared in library level packages, either in the specification or in the body. The global data area contains the data objects that are allocated in a static manner. Each object allocated in the global data area has a permanent lifetime. The global data area is allocated in the `$DATA$` segment of the HP-UX object file (see `a.out_800(4)`).

The heap is used for objects that are created by an Ada allocator as well as objects created via indirect allocation. Storage for task objects, including the task stack, are also allocated into the heap. The heap contains the data objects that are dynamic in the broadest sense. Each object allocated in the heap has a lifetime that begins after the allocator operation and ends only when an implicit or explicit deallocator operation is performed. The heap is allocated using the `sbrk(2)` system call.

For constants that are scalar in type, no storage is allocated or used. The values are stored in a compile-time data structure. Because these scalar constants do not have an allocation address, it is illegal to refer to their address (using the attribute 'ADDRESS) or to supply an address for them (using an address clause.) Constants that are aggregates or non-scalar are allocated into one of the above three locations.

Objects that are created by the compiler, such as temporaries, also obey the above rules.

F 4.9.1 Direct Allocation versus Indirect Allocation

The HP Ada compiler determines whether to allocate each object directly in the frame or in the global data area, or to allocate it dynamically in the heap and access it indirectly via a pointer. These two modes are called *direct* and *indirect*, respectively. The determination is based on the object's size or its maximum possible size. An allocation map will list the size of all objects and can be produced by using the `-S` assembly option.

Note that objects of the unconstrained type `STRING`, including those returned by functions that return the type `STRING`, are allocated in the heap.

F 4.9.2 Object Deallocation

This section describes compiler-generated objects, programmer-generated objects, and program termination.

F 4.9.2.1 Compiler-Generated Objects

All objects that the compiler chooses to represent in an indirect form will automatically be freed and their storage reclaimed when leaving the scope in which the objects are declared. Moreover, all compiler-generated temporaries that are allocated on the heap in a scope will be deallocated upon leaving the scope. These compiler temporaries are often generated by such operations as function calls returning unconstrained arrays, or by using the `STRING` concatenation operator (`&`). By enclosing the statements in a **begin** ... **end** block, you can force the heap to reclaim the temporaries associated with any statement.

The storage associated with a task object, including its stack space, is automatically freed when the task terminates.

F 4.9.2.2 Programmer-Generated Objects

Whether the storage for an object created with an Ada allocator is reclaimed depends on where the access type is declared.

For access types declared in a nested or non-library level scope, all objects created with an Ada allocator will automatically be freed and their storage reclaimed when leaving the scope in which the type was declared. Thus, pragma CONTROLLED is effectively applied to all access types by default. Upon exiting a scope that declares an access type, the lifetime of objects of this access type expires and the storage can be reclaimed safely.

For access types declared in a library level scope or with library package visibility, objects that are created using a type declared in a library level package will not be freed by the Ada Runtime System. The compiler cannot determine the lifetime of the object and thus must assume that a future reference to the object could occur at any time. For these kinds of objects, it is the programmer's responsibility to reclaim their storage through the careful use of UNCHECKED_DEALLOCATION.

F 4.9.2.3 Program Termination

All memory used by a program, including code, global data, I/O buffers, and so on, is released when the program terminates and returns to HP-UX. This is the standard behavior of any program under HP-UX.

F 4.9.3 Dynamic Memory Management

This section explains how dynamic memory is managed by the Ada Runtime System.

F 4.9.3.1 Collections of Objects

Every access type has a corresponding collection of objects associated with it. Each use of an allocator queries the corresponding collection and then reserves the correct amount of space within the heap. Each use of UNCHECKED_DEALLOCATION updates the collection data structures and effectively gives back the space for future use.

The size of the space taken from the heap depends on:

- The designated type.
- The access value type.
- Possibly, for an unconstrained record access type, the supplied value of the discriminant or discriminants either when the object is created or again when a new value is given to the object.

The effective size of the object can be obtained using the predefined attribute 'SIZE. For an unconstrained array access type, a descriptor is added that holds the unconstrained actual dimension or dimensions with the actual size of the array; thus, the descriptor size is the sum of the size container (generally 4 bytes) and all the actual constraints (array bounds) implemented the same way as their index type (either 1, 2, or 4 bytes each).

The heap manager applies the following rules to each object:

- The size of the object is rounded up to an even number of bytes.
- The minimum size is 12 bytes. Thus, if the object is less than 12 bytes, it is increased to 12 bytes.
- To the above size, the following is added: a 8-byte descriptor if the collection is global (is declared within a library level package) or a 40-byte descriptor if the collection is in a nested scope (declared within a procedure or task body).

A special rule applies for collections where the objects of the designated type are of static size and are smaller than 64 bytes. Instead of allocating one object at a time within the heap, blocks of several objects are allocated. The size of the block is either 128 bytes or 16 times the object size plus 10 bytes, whichever is greater. To the size of this block, the heap manager applies the above rules; that is, the heap manager adds either a 8- or 40-byte descriptor.

When a collection size is specified using a length clause for an access type (that is, for `T'STORAGE_SIZE use` <*nnn*>), the heap manager allocates a single block of the specified size. The size of this block will be exactly the size specified in the length clause. Individual objects will then be allocated from this block. The minimum size of an individual object allocated from this block is four bytes. Whenever `UNCHECKED_DEALLOCATION` is performed on an object, the collection will add the object to the free list associated with the collection. This free list is maintained as a linked list of pointers contained directly within the collection. No additional storage is required to maintain the free list. When space in the collection is exhausted, the exception `STORAGE_ERROR` is raised.

4

F 4.9.3.2 Global Dynamic Objects

A global dynamic object is a user-declared object whose size is not known until execution time and is declared within a library package (in a specification or body including nested packages and blocks, but not subprograms or tasks.)

The compiler also considers an object as dynamic if the size is bigger than 1024 bytes, even if this size is known statically at compile time. The compiler also considers any object as static if the maximum size is smaller than 128 bytes, even if this size must be dynamically computed at execution time.

All such global dynamic objects are allocated within the heap. The size of these objects can be obtained using the predefined attribute `'SIZE` and the heap manager applies the same rules to them as it does for collections of objects, as described in section "F 4.9.3.1 Collections of Objects".

F 4.9.3.3 Local Objects

Local objects, declared within a subprogram or task, are normally allocated in the stack. This is done either in the frame associated with the subprogram or task execution, or dynamically on top of the stack at the time of elaboration of the object declaration when the size of the object is dynamic.

The heap is used for an unconstrained record if the object discriminant or discriminants can be changed during the lifetime of the object. This discriminant change has a potentially large effect on the object size. In this case, the object is allocated in the heap, and when the discriminant changes, a new space of the desired size is allocated and the old space is given back to the heap.

The heap manager applies the same rules for object size as described in "F 4.9.3.1 Collections of Objects".

F 4.9.3.4 Temporary Objects

During code execution, it is sometimes necessary to take some memory space from the heap to hold temporary object values. This only happens when the memory size is not known at compiler time, or memory size is big enough (more than 128 bytes) to be considered dynamic rather than static for this implementation.

The following cases are possible:

- Function results that are of a dynamic size.
- Evaluation of large aggregates.
- Operations on dynamic arrays (such as catenation of object of type `STRING`; also the predefined operators **and**, **or**, **xor**, and **not** used on dynamic Boolean arrays).
- Evaluation of the predefined attribute `'IMAGE`.

F 4.9.3.5 Reclaiming Heap Storage

Heap storage is reclaimed as follows:

- For the collection of an access type, all storage allocated is returned upon exiting the scope in which access type was declared. Reclaiming occurs whether the exit is normal or abnormal (that is, due to exception propagation.)
- The storage of a task, (including its' stack) is reclaimed when the task terminates.
- For an object passed to an instance of the generic procedure `UNCHECKED_DEALLOCATION`, the storage associated with the object is reclaimed immediately.
- For a temporary object, storage is returned no later than on exit from the scope (subprogram or block) that contained the allocation of the temporary object.

For objects of an access type declared in a library package, automatic reclaiming is not performed. This would require automatic garbage collection with its' inherent overhead at runtime. You should perform `UNCHECKED_DEALLOCATION` to reclaim this storage.

Reclaimed heap storage is managed internally by the Ada Runtime System. Such memory is *never* released back to HP-UX using the `sbrk(1)` system call. Reclaimed heap storage is available for the Ada Runtime System to reuse; however, to the HP-UX kernel, it appears that the memory is still in use by the Ada Runtime System.

5

F 5. Names for Predefined Library Units

Names that are available, but should be avoided if you want access to packages that are provided by Hewlett-Packard:

```
ELEMENTARY_FUNCTIONS_EXCEPTIONS
GENERIC_ELEMENTARY_FUNCTIONS
MATH_LIB
MATH_LIB_LIBM
SYSTEM_ENVIRONMENT
```

The above packages are documented in the *Ada User's Guide*.

6

F 6. Address Clauses

This chapter describes the available address clauses.

F 6.1 Objects

An address clause can be used to specify an address for an object as described in the *Ada RM*, section 13.5. When such a clause applies to an object, no storage is allocated for it in the program generated by the compiler. Storage for the object *must* be allocated for the object outside of the Ada program unit unless the address is a memory mapped hardware address. The Ada program accesses the object by using the address specified in the address clause.

An address clause is not allowed for unconstrained records whose maximum size can be greater than 24576 bytes.

Note that the function `SYSTEM.VALUE`, defined in the package `SYSTEM`, is available to convert a `STRING` value into a value of type `SYSTEM.ADDRESS` (see section "F 3.1 The Package SYSTEM" for details). Note that the `IMPORT` attribute is available to provide the address of an external symbol (see section "F 2.2 Attribute SYSTEM.ADDRESS'IMPORT" for details).

F 6.2 Subprograms

Address clauses for subprograms are not implemented in the current version of the Ada compiler.

F 6.3 Constants

Address clauses for constants are not implemented in the current version of the Ada compiler.

F 6.4 Packages

Address clauses for packages are not implemented in the current version of the Ada compiler.

6

F 6.5 Tasks

Address clauses for tasks are not implemented in the current version of the Ada compiler.

F 6.6 Data Objects

An address clause can specify the address for an object as described in the *Ada RM*, section 13.5. The address supplied must be either an integer constant or the value returned by the implementation-defined attribute SYSTEM.ADDRESS'IMPORT. This attribute is defined to return a reference value that can be used as the address of an external static data object. This attribute takes two parameters: the language and the name of the external data object. Both of these parameters are Ada strings.

Example

```
IMPORT_OBJ:  INTEGER;

for IMPORT_OBJ use at SYSTEM.ADDRESS'IMPORT("c", "c_obj");

MEMORY_MAPPED_OBJ:  INTEGER;

for MEMORY_MAPPED_OBJ use at 16#6FFF_0400#;
```

See section "F 2.2 Attribute SYSTEM.ADDRESS'IMPORT" for more details.

F 6.7 Task Entries

An address clause can be supplied for an Ada task entry. The actual address of the Ada task entry is not bound to the value supplied by the address clause. Instead, this kind of address clause is used to provide the interrupt entry mechanism (see section "F 12. Interrupt Entries" for details).

7

F 7. Restrictions on Unchecked Type Conversions

The following limitations apply to the use of UNCHECKED_CONVERSION:

- Unconstrained arrays are not allowed as target types.
- Unconstrained record types without defaulted disciminants are not allowed as target types.
- Access types to unconstrained arrays or unconstrained strings are not allowed as source or target types.
- If the source and target types are each scalar types, the sizes of the types must be equal.
- If the source and target types are each access types, the sizes of the objects that the types denote must be equal.
- If the source or target type is a composite type, the sizes do not have to be equal. See the warning below for more details.

If the source and target types are each of scalar or access types or if they are both of composite type with the same static size, the effect of the function is to return the operand.

In other cases, the effect of unchecked conversion can be considered as a copy operation.

Caution When you do an UNCHECKED_CONVERSION among types whose sizes do not match, the code that is generated copies as many bytes as necessary from the source location to fill the target. If the target is larger than the source, the code copies all of the source plus whatever follows the source. Therefore, an UNCHECKED_CONVERSION among types whose sizes do not match can produce meaningless results, or can actually cause a trap and abort the program (if these memory locations do not actually exist).

8

F 8. Implementation-Dependent Input-Output

Characteristics

This chapter describes the I/O characteristics of Ada on the HP 9000 Series 600, 700, and 800 computers. Ada handles I/O with packages, which are discussed in section "F 8.1 Ada I/O Packages for External Files". File types are described in section "F 8.1.3 Standard Implementation of External Files" and the FORM parameter is discussed in section "F 8.2 The FORM Parameter".

F 8.1 Ada I/O Packages for External Files

In Ada, I/O operations are considered to be performed on *objects* of a certain file type rather than directly on external files. An external file is anything external to the program that can produce a value to be read or receive a value to be written. In Ada, values transferred to and from a given file must all be of the same type.

Generally, the term *file object* refers to an Ada file of a certain file type, whereas a physical manifestation is known as an *external file*. An external file is characterized by:

- Its NAME, which is a string defining a legal pathname for an external file on the underlying operating system. HP-UX is the underlying operating system for Ada. The rules that govern legal pathnames for external files in Ada programs are the same as those that govern legal pathnames in HP-UX. See section "F 8.1.2 Correspondence between External Files and HP-UX" for details.
- Its FORM, which allows you to supply implementation-dependent information about the external file characteristics.

Both NAME and FORM appear explicitly in the Ada CREATE and OPEN procedures. These two procedures perform the association of the Ada file object and the corresponding external file. At the time of this association, a FORM parameter is permitted to specify additional characteristics about the external file.

Ada I/O operations are provided by several predefined standard packages. See the *Ada RM*, Chapter 14 for more details. Table 8-1 describes the standard predefined Ada I/O packages.

Table 8-1. Standard Predefined I/O Packages

Package	Description and *Ada RM* Location
SEQUENTIAL_IO	A generic package for sequential files of a single element type. (*Ada RM*, section 14.2.3)
DIRECT_IO	A generic package for direct (random) access files of a single element type. (*Ada RM*, section 14.2.5)
TEXT_IO	A non-generic package for ASCII text files. (*Ada RM*, section 14.3.10)
IO_EXCEPTIONS	A package that defines the exceptions needed by the above three packages. (*Ada RM*, section 14.5)

The generic package LOW_LEVEL_IO is not implemented.

F 8.1.1 Implementation-Dependent Restrictions on I/O Packages

The upper bound for index values in DIRECT_IO and for line, column, and page numbers in TEXT_IO is:

```
COUNT'LAST = 2**31 - 1
```

The upper bound for field widths in TEXT_IO is:

```
FIELD'LAST = 255
```

F 8.1.2 Correspondence between External Files and HP-UX

Files

When Ada I/O operations are performed, data is read from and written to external files. Each external file is implemented as a standard HP-UX file. However, before an external file can be used by an Ada program, it must be associated with a file object belonging to that program. This association is achieved by supplying the name of the file object and the name of the external file to the procedures CREATE or OPEN of the predefined I/O packages. Once the association has been made, the external file can be read from or written to with the file object. Note that for SEQUENTIAL_IO and DIRECT_IO, you must first instantiate the generic package to produce a non-generic instance. Then you can use the CREATE or OPEN procedure of that instance. The example at the end of this section illustrates this instantiation process.

The name of the external file can be either of the following:

- an HP-UX pathname
- a null string (for CREATE only)

The exception USE_ERROR is raised by the procedure CREATE if the specified external file cannot be created. The exception USE_ERROR is also raised by the procedure OPEN if you have insufficient access rights to the file.

If the name is a null string, the associated external file is a temporary file created using the HP-UX facility tmpnam(3). This external file will cease to exist upon completion of the program.

When using OPEN or CREATE, the Ada exception NAME_ERROR is raised if any path component exceeds 255 characters or if an entire path exceeds 1023 characters. This limit applies to path components and the entire path during or after the resolution of symbolic links and context-dependent files (CDFs).

Caution The absence of NAME_ERROR does not guarantee that the path will be used as given. During and after the resolution of symbolic links and context-dependent files (CDFs), the underlying file system may truncate an excessively long component of the resulting pathname. For example, a fifteen character file name used in an Ada program OPEN or CREATE call will be silently truncated to fourteen characters without raising NAME_ERROR by an HP-UX file system that is configured for "short filenames".

If an existing external file is specified to the CREATE procedure, the contents of that file will be deleted. The recreated file is left open, as is the case for a newly created file, for later access by the program that made the call to create the file.

Example

```
-- This example creates a file using the generic
-- package DIRECT_IO.  It also demonstrates how
-- to close a file and reopen it using a
-- different file access mode.
--
with DIRECT_IO;
with TEXT_IO;
procedure RTEST is
     --
     -- here we instantiate DIRECT_IO on the type INTEGER
     --
     package INTIO is new DIRECT_IO (INTEGER);

     -- Define a file object for use in Ada
```

```
    IFILE  : INTIO.FILE_TYPE;

    IVALUE : INTEGER := 0;    -- Ordinary integer object

begin
    INTIO.CREATE ( FILE => IFILE,
                   -- Ada file is IFILE
                   MODE => INTIO.OUT_FILE,
                   -- MODE allows WRITE only
                   NAME => "myfile"
                   -- file name is "myfile"
                 );
    TEXT_IO.PUT_LINE ("Created : " &
                      INTIO.NAME (IFILE) &
                      ", mode is " &
                      INTIO.FILE_MODE'IMAGE
                         (INTIO.MODE (IFILE)) );

    INTIO.WRITE (IFILE, 21); -- Write the integer 21 to the file

    -- Close the external file
    INTIO.CLOSE ( FILE => IFILE);

    TEXT_IO.PUT_LINE("Closed file");

    INTIO.OPEN ( FILE => IFILE,
                 -- Ada file is IFILE
                 MODE => INTIO.INOUT_FILE,
                 -- MODE allows READ and WRITE
                 NAME => "myfile"
                 -- file name is "myfile"
               );
    TEXT_IO.PUT_LINE ("Opened : " &
                      INTIO.NAME (IFILE) &
                      ", mode is " &
                      INTIO.FILE_MODE'IMAGE
                         (INTIO.MODE (IFILE)) );
```

```
    INTIO.READ (IFILE, IVALUE);      -- Read the first item

    TEXT_IO.PUT_LINE("Read from file,
                     IVALUE = " & INTEGER'IMAGE(IVALUE));

    INTIO.WRITE (IFILE, 65); -- Write the integer 65 to the file

    TEXT_IO.PUT_LINE("Added an Integer to :
                     " & INTIO.NAME (IFILE));

    INTIO.RESET ( FILE => IFILE,
                  -- Set MODE to allow READ only
                  MODE => INTIO.IN_FILE);
                  -- and move to the beginning of the file.
                  -- (IFILE remains open)

    TEXT_IO.PUT_LINE ("Reset : " &
                      INTIO.NAME (IFILE) &
                      ", mode is " &
                      INTIO.FILE_MODE'IMAGE(INTIO.MODE (IFILE)) );

    while not INTIO.END_OF_FILE(IFILE) loop
        INTIO.READ (IFILE, IVALUE);
        TEXT_IO.PUT_LINE("Read from file, IVALUE = " &
                          INTEGER'IMAGE(IVALUE) );
    end loop;

    TEXT_IO.PUT_LINE("At the end of file, IFILE");

    INTIO.CLOSE ( FILE => IFILE);
    TEXT_IO.PUT_LINE("Close file");

end RTEST;
```

In the example above, the file object is IFILE, the external file name relative to your current working directory is myfile, and the actual rooted path could be /PROJECT/myfile. Error or informational messages from the Ada development system (such as the compiler or tools) may mention the actual rooted path.

Note The Ada development system manages files internally so that names involving symbolic links (see `ln(1)`) are mapped back to the actual rooted path. Consequently, when the Ada development system interacts with files involving symbolic links, the actual rooted pathname may be mentioned in informational or error messages rather than the symbolic name.

F 8.1.3 Standard Implementation of External Files

External files have a number of implementation-dependent characteristics, such as their physical organization and file access rights. It is possible to customize these characteristics through the FORM parameter of the CREATE and OPEN procedures, described fully in section "F 8.2 The FORM Parameter". The default of FORM is the null string.

The next three subsections describe the Ada implementation of these three types of external files: SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO files.

F 8.1.3.1 SEQUENTIAL_IO Files

A SEQUENTIAL_IO file is a sequence of elements that are transferred in the order of their appearance to or from an external file. Each element in the file contains one object of the type that SEQUENTIAL_IO was instantiated on. All objects in the file are of this same type. An object stored in a SEQUENTIAL_IO file has exactly the same binary representation as an Ada object in the executable program.

The information placed in a SEQUENTIAL_IO file depends on whether the type used in the instantiation is a constrained type or an unconstrained type.

For a SEQUENTIAL_IO file instantiated with a constrained type, each element is simply the object. The objects are stored consecutively in the file without separators. For constrained types, the number of bytes occupied by each element is the size of the constrained type and is the same for all elements. Files created using SEQUENTIAL_IO on constrained types *can* be accessed as

DIRECT_IO files at a later time. The representation of both SEQUENTIAL_IO and DIRECT_IO files are the same when using constrained types.

For a SEQUENTIAL_IO file instantiated with an unconstrained type, each element is composed of three parts: the size (in bytes) of the object is stored in the file as a 32-bit integer value, the object, and a few optional unused trailing bytes. These unused trailing bytes will only be appended if the FORM parameter RECORD_UNIT was specified in the CREATE call. This parameter instructs the Ada runtime to round up the size of each element in the file to be an integral multiple of the RECORD_UNIT size. The default value for RECORD_UNIT is one byte, which means that unused trailing bytes will *not* be appended. The principle use for the RECORD_UNIT parameter is in reading and writing external files that are in formats that already use this convention. Files created using SEQUENTIAL_IO on unconstrained types *cannot* be accessed as DIRECT_IO files at a later time. The representation of SEQUENTIAL_IO and DIRECT_IO files are *not* the same when using an unconstrained type. See section "F 8.2.10.2 The Structure of DIRECT_IO and SEQUENTIAL_IO Files" for more information on file structure.

A SEQUENTIAL_IO file can be buffered. Buffering is selected by specifying a non-zero value for the FORM parameter, BUFFER_SIZE. The I/O performance of an Ada program will be considerably improved if buffering is used. By default, no buffering takes place between the physical external file and the Ada program. See section "F 8.2.4 The FORM Parameter Attribute - File Buffering" for details on specifying a file BUFFER_SIZE.

F 8.1.3.2 DIRECT_IO Files

A DIRECT_IO file is a set of elements each occupying consecutive positions in a linear order. DIRECT_IO files are sometimes referred to as random-access files because an object can be transferred to or from an element at any selected position in the file. The position of an element in a DIRECT_IO file is specified by its index, which is a number in the range 1 to (2**31)-1 of the subtype POSITIVE_COUNT. Each element in the file contains one object of the type that DIRECT_IO was instantiated on. All objects in the file are of this same type. The object stored in a DIRECT_IO file has exactly the same binary representation as the Ada object in the executable program.

Elements within a DIRECT_IO file *always* have the same size. This requirement allows the Ada runtime to easily and quickly compute the location of any element in a DIRECT_IO file.

For a DIRECT_IO file instantiated with a constrained type, the number of bytes occupied by each element is the size of the constrained type. Files created using DIRECT_IO on constrained types *can* be accessed as SEQUENTIAL_IO files at a later time. The representation of both DIRECT_IO and SEQUENTIAL_IO files are the same when using a constrained type.

For DIRECT_IO files instantiated with an unconstrained type, the number of bytes occupied by each element is determined by the FORM parameter, RECORD_SIZE. All of the unconstrained objects stored in the file must have an actual size that is less than or equal to this size. The exception DATA_ERROR is raised if the size of an unconstrained object is larger than this size. Files created using DIRECT_IO on unconstrained types *cannot* be accessed as SEQUENTIAL_IO files at a later time. The representation of DIRECT_IO and SEQUENTIAL_IO files are *not* the same when using an unconstrained type. See section "F 8.2.10.2 The Structure of DIRECT_IO and SEQUENTIAL_IO Files" for more information on file structure.

If the file is created with the default FORM parameter attributes (see section "F 8.2 The FORM Parameter"), only objects of a constrained type can be written to or read from a DIRECT_IO file. Although an instantiation of DIRECT_IO is accepted for unconstrained types, the exception USE_ERROR is raised on any call to CREATE or OPEN when the default value of the FORM parameter is used. You *must* specify the maximum RECORD_SIZE for the unconstrained type.

A DIRECT_IO file can be buffered. Buffering is selected by specifying a non-zero value for the FORM parameter, BUFFER_SIZE. The I/O performance of an Ada program will normally be considerably improved if buffering is used. However, for a DIRECT_IO file that is accessed in a random fashion, performance can actually be degraded. The buffer will always reflect a contiguous set of elements in the file and if subsequent I/O requests lie outside of the current buffer, the entire buffer will be updated. This could cause performance to degrade if a large buffer is used and each I/O request requires that the buffer be updated. By default, no buffering takes place between the physical external file and the Ada program. See section "F 8.2.4 The FORM Parameter Attribute - File Buffering" for details on specifying a file BUFFER_SIZE.

F 8.1.3.3 TEXT_IO Files

A TEXT_IO file is used for the input and output of information in readable form. Each TEXT_IO file is read or written sequentially as a sequence of characters grouped into lines and as a sequence of lines grouped into pages. All TEXT_IO column numbers, line numbers, and page numbers are in the range 1 to (2**31)-1 of subtype POSITIVE_COUNT. The line terminator (end-of-line) is physically represented by the character ASCII.LF. The page terminator (end-of-page) is physically represented by a succession of the two characters, ASCII.LF and ASCII.FF, in that order. The file terminator (end-of-file) is physically represented by the character ASCII.LF and is followed by the physical end of file. There is no ASCII character that marks the end of a file. An exception to this rule occurs when reading from a terminal device. In this case, the EOF character defined by HP-UX is used by the Ada runtime to indicate the end-of-file (see stty(1) and termio(7) for details.) See "F 8.2.10.1 The Structure of TEXT_IO Files"in this section for more information about the structure of text files.

If you leave the control of line, page, and file terminators to the Ada runtime and use only TEXT_IO subprograms to create and modify the text file, you need not be concerned with the above terminator implementation details. However, you *must not* output the characters ASCII.LF or ASCII.FF when using TEXT_IO.PUT operations because these characters would be interpreted as line terminators or as page terminators when the file was later read using TEXT_IO.GET. If you affect structural control by explicitly outputting these control characters, it is your responsibility to maintain the integrity of the external file.

If your text file was not created using TEXT_IO, your text file may not be in a format that can be interpreted correctly by TEXT_IO. It may be necessary to filter the file or perform other modifications to the text file before it can be correctly interpreted as an Ada text file. See section "F 8.2.10.1 The Structure of TEXT_IO Files" for information on the structure of TEXT_IO files.

The representation of a TEXT_IO file is a sequence of ASCII characters. It is possible to use DIRECT_IO or SEQUENTIAL_IO to read or write a TEXT_IO file. The Ada type CHARACTER *must* be used in the instantiation of DIRECT_IO or SEQUENTIAL_IO. It is *not* possible to use DIRECT_IO or SEQUENTIAL_IO on the Ada type STRING to read or write a TEXT_IO file.

A TEXT_IO file can be buffered. Buffering is selected by specifying a non-zero value for the FORM parameter, BUFFER_SIZE. The I/O performance of an Ada program will be considerably improved if buffering is used. By default, no buffering takes place between the physical external file and the Ada program. However, terminal input is line buffered by default. See sections "F 8.2.4 The FORM Parameter Attribute - File Buffering" and "F 8.2.8 The FORM Parameter - FIFO Control" for details.

F 8.1.4 Default Access Protection of External Files

HP-UX provides protection of a file by means of access rights. These access rights are used within Ada programs to protect external files. There are three levels of protection:

- User (the owner of the file).
- Group (users belonging to the owner's group).
- Others (users belonging to other groups).

For each of these levels, access to the file can be limited to one or several of the following rights: read, write, or execute. The default HP-UX external file access rights are specified by using the umask(1) command (see umask(1) and umask(2) in the *HP-UX Reference*.) Access rights apply equally to sequential, direct, and text files. See section "F 8.2.3 The FORM Parameter Attribute - File Protection" for information about specifying file permissions at the time of CREATE.

F 8.1.5 System Level Sharing of External Files

Under HP-UX, several programs or processes can access the same HP-UX file simultaneously. Each program or process can access the HP-UX file either for reading or for writing. Although HP-UX can provide file and record locking protection using fcntl(2) or lockf(2), Ada does not utilize this feature when it performs I/O on external files. Thus, the external file that Ada reads or writes is not protected from simultaneous access by non-Ada processes, or by another Ada program that is executing concurrently. Such protection is

outside the scope of Ada. However, you can limit access to a file by specifying a file protection mask using the FORM parameter when you create the file. See section "F 8.2.3 The FORM Parameter Attribute - File Protection" for more information.

The effects of sharing an external file depend on the nature of the file. You must consider the nature of the device attached to the file object and the sequence of I/O operations on the device. You also must consider the effects of file buffering if you are attempting to update a file that is being shared.

For shared files on random access devices, such as disks, the data is shared. Reading from one file object does not affect the file positioning of another file object, nor the data available to it. However, writing to a file object may not cause the external file to be immediately updated; see section "F 8.2.5.1 Interaction of File Sharing and File Buffering" for details.

For shared files on sequential devices or interactive devices, such as magnetic tapes or keyboards, the data is no longer shared. In other words, a magnetic record or keyboard input character read by one I/O operation is no longer available to the next operation, whether it is performed on the same file object or not. This is simply due to the sequential nature of the device.

By default, file objects represented by STANDARD_INPUT and STANDARD_OUTPUT are preconnected to the HP-UX streams `stdin` and `stdout` (see `stdio(5)`), and thus are of this sequential variety of file. The HP-UX stream `stderr` is not preconnected to an Ada file, but is used by the Ada runtime system for error messages. An Ada subprogram called PUT_TO_STANDARD_ERROR is provided in the package SYSTEM_ENVIRONMENT which allows your program to output a line to the HP-UX stream `stderr`.

Note The sharing of external files is system-wide and is managed by the HP-UX operating system. Several programs may share one or more external files. The file sharing feature of HP Ada using the FORM parameter SHARED, which is discussed in section "F 8.2.5 The FORM Parameter Attribute - File Sharing", is not system-wide, but is a file sharing within a single Ada program and is managed by that program.

F 8.1.6 I/O Involving Access Types

When an object of an access type is specified as the source or destination of an I/O operation (read or write), the 32-bit binary access value is read or written unchanged. If an access value is read from a file, make sure that the access value read designates a valid object. This *will only* be the case if the access value read was previously written by the *same execution* of the program that is reading it, and the object which it designated at the time it was written still exists (that is, the scope in which it was allocated has not been exited, nor has an `UNCHECKED_DEALLOCATION` been performed on it). A program may execute erroneously or raise `PROGRAM_ERROR` if an access type read from a file does not designate a valid object. In general, I/O involving access types is strongly discouraged.

F 8.1.7 I/O Involving Local Area Networks

This section assumes knowledge of networks. It describes Ada program I/O involving Local Area Network (LAN) services available on the Series 600, 700, and 800 computers.

The Ada programs discussed here are executed on a local (host) computer. These programs access or create files on a remote system, which is connected to a mass storage device not directly connected to the host computer. The remote file system can be mounted and accessed by the host computer using NFS† LAN services. NFS systems are described in the manuals *Using NFS Services* and *Installing and Administering NFS Services*.

† NFS is a trademark of Sun Microsystems, Inc.

If an Ada program expects to access or create a file on a remote file system using NFS LAN services, the remote volumes that contain the file system must be mounted on the host computer prior to the execution of the Ada program.

For example, assume that a remote system exports a file system `/project`. `/project` is mounted on the host computer as `/ada/project`. Files in this remote file system are accessed or created by references to the files as if they were part of the local file system. To access the file `test.file`, the program would reference `/ada/project/test.file` on the local system. Note that `test.file` appears as `/project/test.file` on the remote system.

The remote file system must be exported to the local system before it can be locally mounted using the `mount(1m)` command.

F 8.1.8 Potential Problems with I/O From Ada Tasks

In an Ada tasking environment on the HP 9000 Series 600, 700, and 800, the Ada runtime must ensure that a file object is protected against attempts to perform multiple simultaneous I/O operations on it. If such protection was not provided, the internal state of the file object could become incorrect. For example, consider the case of two tasks each writing to `STANDARD_OUTPUT` simultaneously. The internal values of a text file object include information returned by `TEXT_IO.COL`, `TEXT_IO.LINE`, and `TEXT_IO.PAGE` functions. These internal values are volatile and any I/O operations that change these values must be completed before any other I/O operations are begun on the file object. Thus, the file object is protected by the Ada runtime for the duration of the I/O operation. If another task is scheduled and runs before the I/O operation has completed and this task attempts to perform I/O on the protected file object, the exception `PROGRAM_ERROR` is generated at the point of the I/O operation. If this exception is not caught by the task, the task will be terminated.

Note that the file protection provided by the Ada runtime is *not* the same as the protection provided by the use of the SHARED attribute of the FORM parameter of CREATE or OPEN calls. The FORM parameter either prohibits or allows multiple Ada file objects to share the same external file. In contrast, the file protection provided by the Ada runtime prohibits the simultaneous sharing of the *same* Ada file object between tasks. The SHARED attribute always deals with *multiple* Ada file objects.

The file protection provided by the Ada runtime will only be a problem when the *same* Ada file object is used by different tasks. When each task uses a separate file object, it is not necessary to provide explicit synchronization when performing I/O operations. This is true even when the file objects are sharing the same external file. However, for this case, you will need to consider the effects of the SHARED attribute and/or file buffering.

Caution It is your responsibility to utilize proper synchronization and mutual exclusion in the use of shared resources. Note that shared access to a common resource (in this case, a file object) could be achieved by a rendezvous between tasks that share that resource. If you write a program in which two tasks attempt to perform I/O operations on the same logical file without proper synchronization, that program is erroneous. (See *Ada RM*, section 9.11)

F 8.1.9 I/O Involving Symbolic Links

Some caution must be exercised when using an Ada program that performs I/O operations to files that involve symbolic links. For more detail on the use of symbolic links to files in HP-UX, see ln(1).

Creating a symbolic link to a file creates a new name for that file; that is, an alias for the actual file name is created. If you use the actual file name or its alias (that is, the name involving symbolic links), Ada I/O operations will work correctly. However, the NAME function in the TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO packages will always return the actual rooted path of a file and *not* a path involving symbolic links.

F 8.1.10 Ada I/O System Dependencies

Ada has a requirement (see *Ada RM*, section 14.2.1(21)) that the `NAME` function must return a string that uniquely identifies the external file in HP-UX. In determining the unique file name, the Ada runtime system may need to access directories and directory entries not directly associated with the specified file. This is particularly true when the path to the file specified involves an NFS remote file system. This access involves HP-UX operating system calls that are constrained by HP-UX access permissions and are subject to failures of the underlying file system, as well as by network behavior.

Caution It is during the Ada `OPEN` and `CREATE` routines that the name which uniquely identifies the external file is determined for later use by the `NAME` function. If it is not possible to determine that name, an exception is raised by the call to the `OPEN` or `CREATE` routine.

If, during name determination, the underlying file system or network denies access (possibly due to a failed remote file system) or the access permissions are improper, the `OPEN` or `CREATE` call will raise an exception. Or, for some conditions of network failure, the call might not complete until the situation is corrected.

During file name determination, the Ada runtime temporarily changes the current working directory of the Ada program. However, it first determines the fully rooted path to the current working directory so it can restore the correct current working directory before returning control to the Ada program. If the runtime cannot determine the fully rooted path to the current working directory (usually because some directory in the fully rooted path to the current working directory lacks read (`r`) or search (`x`) permission for the effective user of the Ada program), the `OPEN` or `CREATE` of the file also fails with an exception.

If the permissions for any directory on the fully rooted path to the current working directory are changed while the Ada

runtime is in the process of determining the name of a file, the Ada runtime may be unable to restore the current working directory. This can only occur if the permission change denies the effective user of the Ada program read (`r`) or search (`x`) access to the directory whose permission changed. Because it is unsafe for the program to continue execution with an incorrect current working directory, the Ada runtime will abort the Ada program with a diagnostic message to `stderr` and a non-zero exit status. This abnormal termination can only occur if:

1. The permissions for a directory (such as `/A`) in the rooted path to a directory (such as `/A/B/C`) are changed while an Ada program is running with `/A/B/C` as its current working directory.
2. The effective user of the Ada program loses read (`r`) or search (`x`) permission (such as `/A`) because of the change.
3. The Ada runtime was in the process of determining the name of a file when the permission changed.

For example, when opening a file, the Ada exception `NAME_ERROR` is raised if there are any directories in the rooted path of the file that are not readable or searchable by the "effective uid" of the program. This restriction applies to intermediate path components that are encountered during the resolution of symbolic links.

Also, if access to an NFS "hard" mounted remote file system is lost (possibly due to a network failure), subsequent `OPEN` or `CREATE` calls on a file whose actual rooted path contains the parent directory of the NFS mount point might not complete until the NFS failure is corrected (whether or not the actual file being accessed is on the failed NFS volume.)

F 8.2 The FORM Parameter

For both the CREATE and OPEN procedures in Ada, the FORM parameter specifies the characteristics of the external file involved.

F 8.2.1 An Overview of FORM Attributes

The FORM parameter is a string composed from a list of attributes that specify the following:

- File protection.
- File buffering.
- File sharing.
- Appending.
- Blocking.
- FIFO control.
- Terminal input.
- File structuring.

F 8.2.2 The Format of FORM Parameters

Attributes of the FORM parameter have an attribute keyword followed by the Ada "arrow symbol" (=>) and followed by a qualifier or numeric value.

The arrow symbol and qualifier are not always needed and can be omitted. Thus, the format for an attribute specifier is

KEYWORD

or

KEYWORD => *QUALIFIER*

The general format for the FORM parameter is a string formed from a list of attributes with attributes separated by commas. (FORM attributes are distinct from Ada attributes and the two are not related.) The FORM parameter string is *not* case sensitive. The arrow symbol can be separated by spaces from the keyword and qualifier. The two forms below are equivalent:

KEYWORD => *QUALIFIER*

KEYWORD=>*QUALIFIER*

In some cases, an attribute can have multiple qualifiers that can be presented at the same time. In cases that allow multiple qualifiers, additional qualifiers are introduced with an underscore (_). Note that spaces are not allowed between the additional qualifiers; only underscore characters are allowed. Otherwise, a USE_ERROR exception is raised by CREATE. The two examples that follow illustrate the FORM parameter format.

The first example illustrates the use of the FORM parameter in the TEXT_IO.OPEN to set the file buffer size.

```
-- Example of opening a file using the non-generic
-- package TEXT_IO.  This illustrates the use of the
-- FORM parameter BUFFER_SIZE.
-- Note: "input_file" must exist or NAME_ERROR will be
-- raised.
with TEXT_IO;
procedure STEST is

--Define a file object for use in Ada
  TFILE : TEXT_IO.FILE_TYPE;
```

```
begin -- STEST
   TEXT_IO.OPEN (FILE => TFILE,  -- Ada file is TFILE
                 MODE => TEXT_IO.IN_FILE, -- Access allows
                                          -- reading
                 NAME => "input_file", -- file name is
                                       -- "input_file"
                 FORM => "BUFFER_SIZE => 4096"
                         -- Buffer Size is 4096 bytes
                 );
end STEST;
```

The second example illustrates the use of the FORM parameter in TEXT_IO.CREATE. This example sets the access rights of the owner (HP-UX file permissions) on the created file and shows multiple qualifiers being presented at the same time.

```
TEXT_IO.CREATE (OUTPUT_FILE, TEXT_IO.OUT_FILE, OUTPUT_FILE_NAME,
                FORM=>"owner=>read_write_execute");
```

F 8.2.3 The FORM Parameter Attribute - File Protection

The file protection attribute is only meaningful for a call to the CREATE procedure.

File protection involves two independent classifications. The first classification specifies *which user* can access the file and is indicated by the keywords listed in Table 8-2.

Table 8-2. User Access Categories

Category	Grants Access To
OWNER	Only the owner of the created file.
GROUP	Only the members of a defined group.
WORLD	Any other users.

Note that WORLD is similar to "others" in HP-UX terminology, but was used in its place because OTHERS is an Ada reserved word.

The second classification specifies *access rights* for each classification of user. The four general types of access rights, which are specified in the FORM parameter qualifier string, are listed in Table 8-3.

Table 8-3. File Access Rights

Category	Allows the User To
READ	Read from the external file.
WRITE	Write to the external file.
EXECUTE	Execute a program stored in the external file.
NONE	The user has no access rights to the external file. (This qualifier overrides any prior privileges).

More than one access right can be specified for a particular file. Additional access rights can be indicated by separating them with an underscore, as noted earlier. The following example using the FORM parameter in TEXT_IO.CREATE sets access rights of the owner and other users (HP-UX file permissions) on the created file. This example illustrates multiple qualifiers being used to set several permissions at the same time.

```
TEXT_IO.CREATE (OUTPUT_FILE, TEXT_IO.OUT_FILE, OUTPUT_FILE_NAME,
                FORM=>"owner=>read_write_execute, world=>none");
```

Note that the HP-UX command umask(1) may have set the default rights for any unspecified permissions. In the previous example, permission for the users in the category GROUP were unspecified. Typically, the default umask will be set so that the default allows newly created files to have read and write permission (and no execute permission) for each category of user (USER, GROUP, and WORLD).

Consider the case where the users in WORLD want to execute a program in an external file, but only the owner may modify the file. The appropriate FORM parameter is:

```
WORLD => EXECUTE,

OWNER => READ_WRITE_EXECUTE
```

This would be applied as:

```
TEXT_IO.CREATE (OUTPUT_FILE, TEXT_IO.OUT_FILE, OUTPUT_FILE_NAME,
                FORM=>"world=>execute, owner=>read_write_execute");
```

Repetition of the same qualifier within attributes is illegal:

```
WORLD => EXECUTE_EXECUTE                -- NOT legal
```

But repetition of entire attributes is allowed:

```
WORLD => EXECUTE, WORLD => EXECUTE      -- legal
```

F 8.2.4 The FORM Parameter Attribute - File Buffering

The buffer size can be specified by the attribute:

BUFFER_SIZE => *size_in_bytes*

The default value for BUFFER_SIZE is 2048 bytes. A value of zero specifies no buffering. Note that a BUFFER_SIZE value of one also specifies no buffering and is treated identical to the value zero. Using the file buffering attribute will improve I/O performance by a considerable amount in most cases. If I/O performance is a concern for disk files, the attribute BUFFER_SIZE should be set to a value that is an integral multiple of the size of a physical disk block. The size of a physical disk block can be found in <sys/param.h> and is 1024 bytes for the HP 9000 Series 600, 700, and 800.

An example of using the FORM parameter in the TEXT_IO.OPEN to set the file buffer size is shown below:

```
-- An example of creating a file using the non-generic
-- package TEXT_IO.  This illustrates the use of the
-- FORM parameter BUFFER_SIZE.

with TEXT_IO;
procedure T_TEST is

  BFILE : TEXT_IO.FILE_TYPE; -- Define a file object
                             -- for use by Ada

begin -- T_TEST

  TEXT_IO.CREATE (FILE => BFILE,
                             -- Ada file is BFILE
                  MODE => TEXT_IO.OUT_FILE,
                             -- MODE is WRITE only
                  NAME => "txt_file",
                             -- External file "txt_file"
                  FORM => "BUFFER_SIZE => 8192"
                             -- Buffer size is 8192 bytes
                 );
end T_TEST;
```

The BUFFER_SIZE attribute can be applied to files associated with terminals operating in TERMINAL_INPUT => LINES mode and to files associated with pipes/FIFOs. However, there are additional considerations to take into account when doing this. See section "F 8.2.9 The FORM Parameter - Terminal Input" in this manual and the section "Ada I/O Operations on a Terminal or Pipe/FIFO" in Chapter 7 in the *Ada User's Guide* for additional information.

F 8.2.5 The FORM Parameter Attribute - File Sharing

The file sharing attribute of the FORM parameter allows you to specify what kind of sharing is permitted when multiple file objects access the same external file. This control over file sharing is *not* system-wide, but is limited to a single Ada program. The HP-UX operating system controls file sharing at the system level. See section "F 8.1.5 System Level Sharing of External Files" for information on system level file sharing between separate programs.

An external file can be shared; that is, the external file can be associated simultaneously with several logical file objects created by the OPEN or CREATE procedures. The file sharing attributes forbids or limits this capability by specifying one of the modes listed in Table 8-4.

Table 8-4. File Sharing Attribute Modes

Mode	Description
NOT_SHARED	Indicates exclusive access. No other logical file can be associated with the external file.
SHARED=> READERS	Only logical files of mode IN can be associated with the external file.
SHARED=> SINGLE_WRITER	Only logical files of mode IN and at most one file with mode OUT can be associated with the external file.
SHARED=> ANY	No restrictions; this is the default.

A USE_ERROR exception is raised if either of the following conditions exists for an external file already associated with at least one logical Ada file:

- The OPEN or CREATE call specifies a file sharing attribute *different* than the current one in effect for this external file. Remember the attribute SHARED => ANY is provided if the shared attribute is missing from the FORM parameter.
- A RESET call that changes the MODE of the file and violates the conditions imposed by the current file sharing attribute (that is, if SHARED => READERS is in effect, the RESET call cannot change a reader into writer).

The current restriction imposed by the file sharing attribute disappears when the last logical file linked to the external file is closed. The next call to CREATE or OPEN can and does establish a new file sharing attribute for this external file. See section "F 8.1.8 Potential Problems with I/O From Ada Tasks" for information about potential problems with I/O from Ada tasks.

F 8.2.5.1 Interaction of File Sharing and File Buffering

For files that are *not buffered* (the default), multiple I/O operations on an external file shared by several file objects are processed in the order they occur. Each Ada I/O operation will be translated into the appropriate HP-UX system call (read(2). write(2), creat(2), open(2), or close(2)) and the external file will be updated by the HP-UX I/O runtime. Note that if file access is performed across a network device, the external file may not be immediately updated. However, additional I/O operations on the file will be queued and must wait until the original operation has completed. This allows multiple readers and multiple writers for the external file.

For files that are *buffered*, multiple I/O operations each operate sequentially only within the buffer that is associated with the file object and each file object has its own buffer. For write operations, this buffer is flushed to the disk either when the buffer is full, or when the file index is positioned outside of the buffer, or when the file is closed. The external file only reflects the changes made by a write operation after the buffer is flushed to the disk. Any accesses to the external file that occur before the buffer is flushed will not reflect the changes made to the file that exist only in the buffer.

Due to this behavior, shared files should *not* be buffered if any write operations are to be performed on this file. This would be the case for file objects of the mode OUT_FILE or INOUT_FILE. Thus, when using buffered files safely, *no* writers are allowed, but multiple readers *are* allowed.

File buffering is enabled by using the FORM parameter attributes at the time you open or create the file. If file buffering is enabled for a file, you should also specify a file sharing attribute of either NOT_SHARED or SHARED=>READERS to prevent the effects of file buffering and file sharing interfering with one another. The Ada runtime will raise the exception USE_ERROR if any attempt is made to share the file or to share and write the file, when the above file sharing attributes are provided as FORM parameters.

If the possibility of shared access exists in your Ada program for sequential devices or interactive devices, you should specify a file sharing attribute of NOT_SHARED. This will prevent the negative effects of file sharing on these kinds of devices.

F 8.2.6 The FORM Parameter Attribute - Appending to a File

The APPEND attribute can only be used with the procedure OPEN. Its format is:

```
APPEND
```

Any output will be placed at the end of the named external file.

Under normal circumstances, when an external file is opened, an index is set that points to the beginning of the file. If the APPEND attribute is present for a sequential or text file, data transfer begins at the end of the file. For a direct access file, the value of the index is set to one more than the number of records in the external file.

The APPEND attribute is *not* applicable to terminal devices.

F 8.2.7 The FORM Parameter Attribute - Blocking

This attribute has two alternative forms:

```
BLOCKING
```

or

```
NON_BLOCKING
```

The default for this attribute depends on the Ada program. The default is BLOCKING for programs without any task declarations and is NON_BLOCKING for programs containing tasks.

The NON_BLOCKING default allows tasking programs to take advantage of their parallelism in the presence of certain I/O requests. There is normally little advantage in specifying NON_BLOCKING within a non-tasking program because the program must wait for the I/O request to complete before continuing its sequential execution. H wever, if a non-tasking program declares interrupt handlers, and if the interrupt handler or handlers are likely to be invoked frequently, NON_BLOCKING may be appropriate to provide more reliable data transfer (see the warning in section "F 8.2.7.1 Blocking" for additional information).

F 8.2.7.1 Blocking

If the FORM parameter BLOCKING is specified (or is the default), I/O operations will "block" at the HP-UX level awaiting completion of the I/O operation. However, this does not guarantee that all Ada tasks will be blocked from running until the "blocked" I/O operation is complete. If there are one or more active delay statements, or if time-slicing is enabled, or if interrupt handlers or entries have been enabled, HP-UX signals may be received by the program during the "blocked" I/O operation. These HP-UX signals will cause HP-UX to terminate the I/O operation as interrupted and may cause the Ada runtime system to permit other tasks to execute. When the task that was performing the "blocked" I/O operation is permitted by the Ada runtime to execute again, the interrupted "blocked" I/O operation will be automatically restarted.

Caution If a "blocked" I/O operation is frequently interrupted and restarted as described above, the operation may be unable to complete successfully. In addition, under some circumstances, data could be lost by HP-UX when an I/O operation is interrupted and restarted. To avoid these difficulties, either specify (or default) the `FORM` parameter to `NON_BLOCKING` or, if a "true blocking" I/O operation is needed, specify (or default) the `FORM` parameter to `BLOCK` and additionally surround the "blocking" I/O operation with calls to `SYSTEM_ENVIRONMENT.SUSPEND_ADA_TASKING` and `SYSTEM_ENVIRONMENT.RESUME_ADA_TASKING`. The `SYSTEM_ENVIRONMENT.SUSPEND_ADA_TASKING` call will disable or block the HP-UX signals associated with delay, time-slicing, and interrupt handlers or entries during the I/O operation.

F 8.2.7.2 Non-Blocking

The `NON_BLOCKING` attribute specifies that when a read or write request cannot be immediately satisfied, the Ada runtime should attempt to schedule another task to run and retry the I/O operation later. The current implementation of this attribute allows for the following three cases:

1. Non-blocking read operations are performed on all terminal devices. Refer to the section "Ada I/O Operations on a Terminal or Pipe/FIFO" in Chapter 7 in the *Ada User's Guide* for additional information.
2. For HP-UX pipes and FIFO special files, read requests will not block when the pipe/FIFO contains no data and write requests will not block when the pipe/FIFO is full. Refer to the section "Ada I/O Operations on a Terminal or Pipe/FIFO" in the Chapter 7 in the *Ada User's Guide* for additional information.
3. The non-blocking attribute sets the HP-UX flag `O_NONBLOCK`, a flag to `open(2)`, that allows non-blocking access to a normal disk file with enforcement-mode recording locking set (see `lockf(2)`).

   For this case, read and write requests will not block only if the portion of the file being accessed is currently locked by another process. Most files will *not* have enforcement-mode record locking enabled.

The normal behavior of HP-UX I/O operations on disk files is to block until the I/O request completes. Thus, in an Ada tasking program, when a single task performs a read operation upon a disk file, HP-UX blocks the process until the I/O request can be satisfied. However, HP-UX will automatically perform disk cacheing so that a write operation will return before data is physically written to the disk.

F 8.2.8 The FORM Parameter - FIFO Control

The FIFO control attribute has one of two alternate forms:

```
FIFO_EOF => YES

FIFO_EOF => NO
```

The default value of FIFO_EOF is YES.

The FIFO_EOF attribute controls the behavior of an Ada file associated with a FIFO special file, as follows:

- When a FIFO special file is opened for reading (IN_FILE), by aefault (FIFO_EOF => YES) the Ada END_OF_FILE condition becomes true when the last process having the FIFO open for writing closes the FIFO special file.
- When a FIFO special file is opened for reading (IN_FILE) and FIFO_EOF => NO is specified, the Ada END_OF_FILE condition never becomes true and so another mechanism is needed to signal that no further attempts to read from such a FIFO should be attempted. This mode of operation would most likely be used by a monitor or daemon type program that monitors (reads) a FIFO that does not always have a writing process attached to it. This mode of operation may also be necessary when a FIFO special file is opened for reading, in NON_BLOCKING mode, to avoid premature signaling of the Ada END_OF_FILE condition when there are currently no processes with the FIFO open for writing.
- When a FIFO special file is opened for writing (OUT_FILE), by default (FIFO_EOF => YES). There are no special conditions or behaviors if the FIFO is opened in BLOCKING mode (the default in non-tasking programs). If the FIFO is opened in NON_BLOCKING mode (the default in tasking programs), the open will fail with a USE_ERROR if the FIFO is not already open for reading by another (or the same) process.

- When a FIFO special file is opened for writing (OUT_FILE) and FIFO_EOF => NO is specified, there are no special conditions or behaviors if the FIFO is opened in BLOCKING mode (the default in non-tasking programs). If the FIFO is opened in NON_BLOCKING mode (the default in tasking programs), an attempt is made to open the FIFO such that a USE_ERROR does not occur if the FIFO is not already open for reading by another (or the same) process. The Ada I/O system attempts to avoid the USE_ERROR by performing the HP-UX open of the FIFO to permit both writing and reading (although Ada I/O only permits writing to the associated Ada file). Because opening a FIFO for both writing and reading provides a reader of the FIFO in the HP-UX sense, the open for writing will not fail with a USE_ERROR if no process already has the FIFO open for reading. Note that if FIFO_EOF => NO, but the FIFO cannot be successfully opened for both writing and reading (that is, you do not have read access to the FIFO), the Ada I/O system reattempts the open for writing only. If the Ada I/O must reattempt the open for writing only, it is then possible for the open to fail with USE_ERROR if there is not already a process with the FIFO open for reading even though FIFO_EOF => NO was specified.
- Once a FIFO special file is successfully opened *explicitly for writing* (OUT_FILE), and the open specified or defaulted to NON_BLOCKING mode, the Ada program receives no notification (that is, no exception is raied) when the last reading process closes the read end of the FIFO. If FIFO_EOF => NO was specified, there is always at least one reader in the HP-UX sense, and the last reader does not close the read end until the Ada file opened for writing is closed (for example, the Ada open for writing opened for FIFO for both reading and writing). If FIFO_EOF => YES was specified (or defaulted), a last reader can close the FIFO, but the Ada program is not notified. The Ada I/O system continues to write data to the FIFO until the FIFO fills up and internally retries further write requests. In a non-tasking Ada program, some Ada write operations to the FIFO eventually blocks the entire program. In a tasking Ada program, no single Ada write operation to the FIFO blocks the entire program, although Ada write operations to the FIFO eventually blocks the tasks performing them.

Refer to the section "Ada I/O Operations on a Terminal or Pipe/FIFO" in Chapter 7 in the *Ada User's Guide* for additional information.

F 8.2.9 The FORM Parameter - Terminal Input

The terminal input attribute has one of two alternative forms:

```
TERMINAL_INPUT => LINES,

TERMINAL_INPUT => CHARACTERS,
```

Terminal input is normally processed by Ada (and HP-UX) in units of one line at a time. An Ada program attempting to read from the terminal as an external file does not receive any data from the terminal until a complete line is typed. At that time, the outstanding read operation (and possibly subsequent read operations) is satisfied. In this mode, the HP-UX line editing characters "kill" and "erase" can be used to edit the input characters before the Ada program is given access to the characters.

The TERMINAL_INPUT => LINES attribute, the default case, specifies the one line at a time mode of transfer. Note that the BUFFER_SIZE attribute may be set to any value greater or equal to zero when LINES mode is in effect. However, if the BUFFER_SIZE is greater than one, no characters are available to the Ada program *until at least BUFFER_SIZE of them have* been entered (which may require multiple lines of input). Once BUFFER_SIZE number of characters has been entered, only BUFFER_SIZE number of characters are available to the Ada program until the next BUFFER_SIZE number of characters are entered. Because BUFFER_SIZE number of characters may be reached in the middle of a line, the behavior of the program may be confusing to the person entering data. A BUFFER_SIZE greater than one is not recommended if a person is entering data, although it may make sense if the "terminal" is a data communication line from a device that is producing the data. Refer to the section on "Ada I/O Operations on a Terminal or Pipe/FIFO" in Chapter 7 in the *Ada User's Guide* for additional information.

Terminal input can optionally be processed by Ada (and HP-UX) in units of one character at a time. An Ada program attempting to read from the terminal as an external file will receive data from the terminal as it is typed. In this mode, the HP-UX line editing characters "kill" and "erase" may not be used to edit the input characters before the Ada program is given access to the characters.

The TERMINAL_INPUT => CHARACTERS attribute specifies that data transfers occur character by character, so a complete line does not need to be entered before one or several read operations are satisfied. Note that the BUFFER_SIZE attribute can only be set to zero or one when CHARACTERS mode is in effect.

When CHARACTERS mode is in effect, the ICANON bit is cleared in the c_lflag of the HP-UX termio structure. This bit changes the line discipline of the terminal device. The line discipline state is not maintained on a per file descriptor basis, so changing the line discipline for one terminal file does affect the line discipline of all terminal files that refer the same physical terminal device or pseudo terminal process or terminal window. Care must be taken if the same terminal device is to be accessed via multiple Ada files; the line discipline caused by the most recent OPEN operation is applied to all Ada files associated with the same terminal. See termio(7) and the section "Ada I/O Operations on a Terminal or Pipe/FIFO" in Chapter 7 in the *Ada User's Guide* for additional information.

Because the TERMINAL_INPUT attribute is only available for explicitly opened files, the TERMINAL_INPUT attribute of the default STANDARD_INPUT file cannot be changed and the default STANDARD_INPUT file always operates in LINES mode. If a terminal is associated with STANDARD_INPUT, it can be accessed in CHARACTERS mode if opened explicitly with the FORM parameter TERMINAL_INPUT => CHARACTERS. The external file name of the terminal associated with STANDARD_INPUT can be obtained with the HP-UX ttyname(3C) function by passing it an argument of zero, or the file name /dev/tty can be used to open the same terminal device as that associated with STANDARD_INPUT.

F 8.2.10 The FORM Parameter Attribute - File Structuring

This section describes the structure of Ada files. It also describes how to use the FORM parameter to effect the structure of Ada files.

F 8.2.10.1 The Structure of TEXT_IO Files

There is no FORM parameter to define the structure of text files. A text file consists of a sequence of bytes containing ASCII character codes.

The usage of Ada terminators depends on the file's mode (IN_FILE or OUT_FILE) and whether it is associated with a terminal device or a mass-storage file.

8

Table 8-5 describes the use of the ASCII characters as Ada terminators in text files.

Table 8-5. Text File Terminators

File Type	TEXT_IO Functions	Characters
Mass storage files (IN_FILE)	END_OF_LINE	ASCII.LF Physical end of file
	END_OF_PAGE	ASCII.LF ASCII.FF ASCII.LF Physical end of file Physical end of file
	END_OF_FILE	ASCII.LF Physical end of file Physical end of file
Mass storage files (OUT_FILE)	NEW_LINE	ASCII.LF
	NEW_PAGE	ASCII.LF ASCII.FF ASCII.LF Physical end of file
	CLOSE	ASCII.LF Physical end of file
Terminal device (IN_FILE)	END_OF_LINE	ASCII.LF ASCII.FF ASCII.EOT
	END_OF_PAGE	ASCII.FF ASCII.EOT
	END_OF_FILE	ASCII.EOT
Terminal device (OUT_FILE)	NEW_LINE	ASCII.LF
	NEW_PAGE	ASCII.LF ASCII.FF
	CLOSE	ASCII.LF

See section "F 8.1.3.3 TEXT_IO Files" for more information about terminators in text files.

F 8.2.10.2 The Structure of DIRECT_IO and SEQUENTIAL_IO Files

This section describes use of the FORM parameter for binary (sequential or direct access) files. Two FORM attributes, RECORD_SIZE and RECORD_UNIT, control the structure of binary files.

Such a file can be viewed as a sequence or a set of consecutive RECORDS. The structure of a record is

```
[ HEADER ]  OBJECT  [ UNUSED_PART ]
```

A record is composed of up to three items:

1. A HEADER consisting of two fields (each 32 bits):
   - The length of the object in bytes.
   - The length of the descriptor in bytes; for this implementation of Ada, the length is always zero.
2. An OBJECT with the exact binary representation of the Ada object in the executable program, possibly including an object descriptor.
3. An UNUSED_PART of variable size to permit full control of the record's size.

The HEADER is implemented only if the actual parameter of the instantiation of the I/O package is unconstrained.

The file structure attributes take the form:

RECORD_SIZE => *size_in_bytes*

RECORD_UNIT => *size_in_bytes*

The attributes' meaning depends on the object's type (constrained or unconstrained) and the file access mode (sequential or direct access).

There are four types of access that are possible:

- Sequential access of fixed size, constrained objects.
- Sequential access of varying size, unconstrained objects, with objects rounded up to a multiple of the *RECORD_UNIT size.*
- Direct access of fixed size, constrained objects.
- Direct access of fixed size, unconstrained objects, with a maximum size for the object.

The consequences of the above are listed in Table 8-6.

Table 8-6. Structuring Binary Files with the FORM Parameter

Object Type	File Access Mode	RECORD_UNIT Attribute	RECORD_SIZE Attribute
Constrained	Sequential I/O Direct I/O	The RECORD_UNIT attribute is illegal.	If the RECORD_SIZE attribute is omitted, no UNUSED_PART is implemented. The default RECORD_SIZE is the object's size. If present, the RECORD_SIZE attribute must specify a record size greater than or equal to the object's size. Otherwise, the exception USE_ERROR is raised.

8

Continued on the next page.

Table 8-6.
Structuring Binary Files with the FORM Parameter (Continued)

Object Type	File Access Mode	RECORD_UNIT Attribute	RECORD_SIZE Attribute
Unconstrained	Sequential I/O	By default, the RECORD_UNIT attribute is one byte. The size of the record is the smallest multiple of the specified (or default) RECORD_UNIT that holds the object and its eight byte HEADER (which is always present in this case). This is the only case where different records in a file can have different sizes.	The RECORD_SIZE attribute is illegal.

Continued on the next page.

Table 8-6.
Structuring Binary Files with the FORM Parameter (Continued)

Object Type	File Access Mode	RECORD_UNIT Attribute	RECORD_SIZE Attribute
Unconstrained	Direct I/O	The RECORD_UNIT attribute is illegal.	The RECORD_SIZE attribute has no default value, and if a value is not specified, the exception USE_ERROR is raised. The RECORD_SIZE value must include the size of the eight byte HEADER, which is always present in this case. The minimum value for RECORD_SIZE for a file of objects of the unconstrained type OBJECT is listed below this table. If you attempt to input or output an object larger than the given RECORD_SIZE, a DATA_ERROR exception is raised.

The minimum value for RECORD_SIZE for a file of objects of the unconstrained type OBJECT accessed with Direct I/O is:

```
((OBJECT'SIZE + SYSTEM.STORAGE_UNIT - 1) / SYSTEM.STORAGE_UNIT) + 8
```

9

F 9. The Ada Development System and HP-UX Signals

The Ada runtime on the HP 9000 Series 600/700/800 uses HP-UX signals to implement the following features of the Ada language:

- Ada exception handling.
- Ada task management.
- Ada delay timing.
- Ada program termination.
- Ada interrupt entries.

F 9.1 HP-UX Signals Reserved by the Ada Runtime

Table 9-1 lists the HP-UX signals reserved and used by the Ada runtime.

Table 9-1. Ada Signals

Signal	Description
SIGALRM	Used for delay and optionally for time-slicing.
SIGVTALRM	Optionally used for time-slicing (default time-slicing signal).
SIGPROF	Optionally used for time-slicing.
SIGILL	Causes the PROGRAM_ERROR exception.
SIGSEGV	Causes the PROGRAM_ERROR exception.
SIGBUS	Causes the PROGRAM_ERROR exception.
SIGFPE	Causes the CONSTRAINT_ERROR, NUMERIC_ERROR, STORAGE_ERROR, or PROGRAM_ERROR exceptions.

Note

The signals SIGSEGV, SIGBUS, and SIGILL are not reserved by the Ada runtime. These signals are never deliberately produced by generated code or by the Ada runtime to cause an exception to be raised. If, due to a programming error, access is attempted on misaligned or protected data (causing SIGSEGV or SIGBUS) or an illegal instruction is executed (causing SIGILL), a PROGRAM_ERROR occurs. When receiving these signals, a PROGRAM_ERROR is raised unless the application has overridden the Ada runtime and an alternative action has been specified. If the Interrupt Entry mechanism (see Chapter 12) is used to specify an Ada handler for one or more of these signals, and the ORIGINAL_HANDLER parameter to INSTALL_HANDLER was not REPLACED, the "original handler" that will be invoked (either FIRST or LAST) will be the default Ada runtime handler that will raise PROGRAM_ERROR. Therefore, ORIGINAL_HANDLER => REPLACED is recommended when using INSTALL_HANDLER with one of these signals.

Note The signals listed as causing exceptions in Table 9-1 will induce an exception even if non-Ada code is executing at the time the signal is received. If interface code causes one of these signals or is running when a signal is received from an outside source, the Ada code that called the interface code will receive an Ada exception.

Note The alarm signals SIGALRM, SIGVTALRM, and SIGPROF are not always used or reserved in an Ada program. See the rest of this section for details.

The HP-UX signals SIGALRM, SIGVTALRM, and SIGPROF are reserved by the Ada runtime for some Ada application program configurations and are not reserved by the Ada runtime for other Ada application program configurations.

If the Ada program contains no tasks, the following is true:

If the Ada program contains			
no delay statements		**one or more delay statements**	
SIGALRM	*not reserved*	SIGALRM	*reserved*
SIGVTALRM	*not reserved*	SIGVTALRM	*not reserved*
SIGPROF	*not reserved*	SIGPROF	*not reserved*

If the Ada program contains tasks, the following is true:

	If the Ada program contains	
	no delay statements	**one or more delay statements**
If time-slicing is disabled with `-W b,-s,0`	`SIGALRM` *not reserved* `SIGVTALRM` *not reserved* `SIGPROF` *not reserved*	`SIGALRM` *reserved* `SIGVTALRM` *not reserved* `SIGPROF` *not reserved*
If time-slicing is enabled with `SIGALRM` timer with `-W b,-S,a`	`SIGALRM` *reserved* `SIGVTALRM` *not reserved* `SIGPROF` *not reserved*	`SIGALRM` *reserved* `SIGVTALRM` *not reserved* `SIGPROF` *not reserved*
If time-slicing is enabled with `SIGVTALRM` timer with `-W b,-S,v`	`SIGALRM` *not reserved* `SIGVTALRM` *reserved* `SIGPROF` *not reserved*	`SIGALRM` *reserved* `SIGVTALRM` *reserved* `SIGPROF` *not reserved*
If time-slicing is enabled with `SIGPROF` timer with `-W b,-S,p`	`SIGALRM` *not reserved* `SIGVTALRM` *not reserved* `SIGPROF` *reserved*	`SIGALRM` *reserved* `SIGVTALRM` *not reserved* `SIGPROF` *reserved*

If a timer signal is not reserved in an application program configuration shown above, the signal can be used for any application-defined purpose, including being associated with an interrupt entry (see "F 9.7 HP-UX Signals Used for Ada Interrupt Entries" and section "F 12. Interrupt Entries" for details.)

F 9.2 Using HP-UX Signals in External Interfaced Subprograms

When your Ada code uses external interfaced subprograms, you must take the following into consideration:

- If the external interfaced subprograms want to manipulate any of the signals reserved by the Ada runtime, they use the `sigvector` and `sigsetmask(2)/sigblock(2)` mechanism or a compatible mechanism. Using the non-compatible `signal(2)` mechanism might produce unpredictable program behavior.
- If the external interfaced subprograms change the signal handling action (that is, `SIG_DEL`, `SIG_IGN`, or user handler) for any HP-UX signal reserved by the Ada runtime, the original signal handling action must be restored before returning control to Ada code. Failure to restore the Ada signal action will produce unpredictable program behavior.
- If the external interfaced subprograms change the signal mask bits of any of the HP-UX signals reserved by the Ada runtime, the original mask bits for those signals must be restored before returning control to Ada code. Failure to restore the original signal mask will produce unpredictable program behavior.

Additional considerations are detailed in section "F 11.7 Potential Problems Using Interfaced Subprograms".

F 9.3 HP-UX Signals Used for Ada Exception Handling

The Ada implementation uses signals to raise exceptions. The Ada runtime handlers for these signals are set during the elaboration of the Ada runtime system. Defining a new handler for any of these signals subverts the normal exception handling mechanism of Ada and will most likely result in an erroneous runtime execution.

If your Ada program uses external interfaced subprograms, you must ensure that these external interfaced subprograms do not redefine the signal behavior for any of the HP-UX signals reserved by the Ada runtime. If you change the signal behavior for the signal used for Ada exception handling (SIGFPE) and your Ada program attempts to raise an exception, unpredictable program behavior will result.

The SIGFPE signal has a predefined meaning and is reserved for use by the Ada runtime for exception handling. The SIGFPE signal is generated in your compiled Ada code whenever one of the predefined runtime checks fail, including null access value checks. The runtime examines the context in which the signal occurred and raises the appropriate exception: CONSTRAINT_ERROR, NUMERIC_ERROR,STORAGE_ERROR, or PROGRAM_ERROR. An unexpected SIGFPE signal that was generated outside of Ada code or sent to the process from an outside source causes the exception PROGRAM_ERROR to be raised. If the unexpected signal occurred in a context where the runtime believed a legitimate exception could have occurred, that exception is raised instead of PROGRAM_ERROR.

The signals SIGSEGV and SIGBUS are not reserved by the Ada runtime. They are generated by access to an illegal or improperly aligned address. Normally these signals are not generated in an Ada program because access values

- are initialized to null
- are only assigned legal and properly aligned values by generated code
- have runtime checks performed on them to detect attempts to dereference a null access value (causing CONSTRAINT_ERROR using SIGFPE as mentioned above)

Such illegal or improperly aligned addresses are usually produced by the improper use of UNCHECKED_CONVERSION or are supplied by interfaced code. In response to receiving SIGSEGV, the Ada runtime raises PROGRAM_ERROR. An unexpected SIGSEGV signal that was generated outside of Ada code or was sent to the process from an outside source also causes the exception PROGRAM_ERROR to be raised.

The signal SIGILL is not reserved by the Ada runtime. It is generated by the execution of an illegal instruction. Normally this signal is not generated in an Ada program because generated Ada code does not contain any illegal instructions. Execution of an illegal instruction usually occurs in interfaced code. In response to receiving SIGILL, the Ada runtime raises PROGRAM_ERROR. An unexpected SIGILL signal that was generated outside of Ada code or was sent to the process from an outside source also causes the exception PROGRAM_ERROR to be raised.

Note User code can define its own handler (or change the signal action) for SIGSEGV, SIGBUS, and SIGILL without directly compromising the operation of the Ada program. However, ignoring a synchronous instance of one of the signals or continuing execution after handling a synchronous instance of one of these signals is not advised without a thorough understanding of the causes and continuation strategies for such signals under HP-UX on PA_RISC.

Note

The Ada binder does not specify `-z` or `-Z` to the linker (`ld(1)`) to control the system action on a dereference of a null pointer. Either the `ld(1)` default or a user-specified value (using `-W l`) will therefore take effect.

If Ada code is compiled with checks enabled (the default case), the Ada Runtime System will operate properly with either `-z` or `-Z` linker options. This is because Ada generates software checks for null pointer dereferencing.

If Ada code is compiled with pointer dereference checks disabled (using the `-C` or `-R` compiler options or using pragma `SUPPRESS`), some null pointer checking can be restored with no runtime overhead by using the `-z` linker option.

If `-z` is specified, the system will send `SIGSEGV` when a null pointer is dereferenced; the Ada Runtime System will map that signal to `PROGRAM_ERROR`. The Ada software checks for null pointer dereferencing are intended to handle all cases where a null pointer could appear. `CONSTRAINT_ERROR` will be raised if such a dereference occurs. `SIGSEGV`, enabled by the `-z` linker option, will only be sent when the final result of an address calculation is the null pointer. The resulting exception will be `PROGRAM_ERROR` instead of `CONSTRAINT_ERROR`.

F 9.4 HP-UX Signals Used for Ada Task Management

When an Ada program contains tasks and time-slicing was enabled (or enabled by default) at bind time, the Ada runtime system uses one of the following to control the time-slice interval: `SIGALRM`, `SIGVTALRM` (the default), or `SIGPROF`. The Ada runtime allocates the available processor time among ready-to-run tasks by giving each task one or more time-slice intervals.

When an Ada program does not contain tasks, or contains tasks but time-slicing was disabled at bind time, neither `SIGVTALRM` nor `SIGPROF` is reserved by the Ada runtime. If an Ada program contains tasks but time-slicing is disabled, `SIGALRM` may or may not be reserved (see "F 9.1 HP-UX Signals Reserved by the Ada Runtime" for more information).

If your Ada program uses external interfaced subprograms, you must ensure that these external interfaced subprograms do not redefine the signal behavior for any of the HP-UX signals reserved by the Ada runtime. If you change the signal behavior for the signal used for Ada task management (`SIGALRM`, `SIGVTALRM`, or `SIGPROF`) and your Ada program is using time-slicing, unpredictable program behavior will result.

F 9.5 HP-UX Signals Used for Ada Delay Timing

When an Ada program contains `delay` statements, the Ada runtime system uses `SIGALRM` to time the delay intervals. The resolution of the `SIGALRM` timer is 1/100 of a second. Thus, all `delay` statements are implemented using actual delays that are integral multiples of 1/100 of a second. Non-zero delays for periods smaller than 1/100 of a second will delay for at least 1/100 of a second. Zero delays will not cause an actual delay, but will provide an opportunity for the Ada runtime to change the currently running task to a different task (if appropriate).

If an Ada program contains `delay` statements, `SIGALRM` is reserved. If an Ada program contains tasks but does not contain any `delay` statements, `SIGALRM` may or may not be reserved (see "F 9.1 HP-UX Signals Reserved by the Ada Runtime" for details).

If your Ada program uses external interfaced subprograms, you must ensure that these external interfaced subprograms do not redefine the signal behavior for any of the HP-UX signals reserved by the Ada runtime. If you change the signal behavior used for Ada delay timing (`SIGALRM`) and your Ada program contains a `delay` statement, unpredictable program behavior will result.

F 9.6 HP-UX Signals Used for Ada Program Termination

The signals `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGTERM`, and `SIGPIPE` are recognized by the Ada runtime as attempts to terminate the Ada program. The Ada runtime initially arranges to catch each of these signals. If the Ada runtime catches one of these signals, Ada runtime cleanup actions are performed and the program is terminated in such a way that the parent program sees the Ada program as having been terminated by the signal. The Ada runtime cleanup actions include flushing file buffers and closing files, as well as restoring terminal characteristics that have been altered by the Ada I/O system. However, these signals are not reserved by the Ada runtime and the application is free to use one or more of these signals for application-defined purposes.

If the application-defined purpose is also to signal that the program should be terminated, when the application is finished handling the signal it should:

1. Restore the original Ada signal handler (the handler the application saved when it altered the signal behavior for its own purposes).
2. Ensure that the signal is not masked.
3. Send the same signal to itself again to invoke the Ada runtime signaled termination process.

Caution

If a signal is not currently reserved by the Ada runtime (see the appropriate sections of F 9) or is not recognized as an attempt to terminate the Ada program (see the list of such signals above) and is received by the Ada program, the HP-UX action may be to terminate the program. Such a termination will not be intercepted by the Ada runtime and the Ada runtime cleanup actions will not occur. This could cause corrupted files and/or corrupted terminal states. If an Ada program is likely to receive such signals, the program should arrange to ignore or mask such signals or to catch and handle such signals. If such a signal terminates the Ada program, the Ada program should arrange to catch and handle such a signal, and should then send one of the defined termination signals (see list above) to itself to trigger the Ada signaled termination process. The Ada program should ensure that the original Ada handler is in effect for that termination signal and that the signal is not masked before sending the signal to itself.

F 9.7 HP-UX Signals Used for Ada Interrupt Entries

Any HP-UX signal that is not reserved by the Ada runtime and that HP-UX permits to be caught can be associated with an interrupt entry. Interrupt entries provide a facility equivalent to that described by the *Ada RM*, section 13.5.1, although the actual mechanism supplied is more general.

The interrupt entry facility is described in detail in section "F 12. Interrupt Entries".

The HP-UX signals that are reserved by the Ada runtime are specified earlier in this section. Those subsections should be consulted to determine which signals can be safely associated with interrupt entries. The interrupt entry mechanism will actually not prohibit the use of signals reserved by the Ada runtime, but using such signals for interrupt entries will cause unpredictable program behavior.

Caution Associating an interrupt entry with a HP-UX signal that can be invoked synchronously (that is, by the execution of faulty code within the Ada program) should only be done with a thorough understanding of the behavior of the underlying hardware and of the behavior of HP-UX in the presence of such faults. Failure to correctly adjust the execution context before resuming after such faults can lead to repeated occurrences of the fault condition and/or other unpredictable program behavior.

F 9.8 Protecting Interfaced Code from Ada's Asynchronous Signals

The `SIGALRM`, `SIGVTALRM`, and `SIGPROF` signals (described in sections "F 9.4 HP-UX Signals Used for Ada Task Management" and "F 9.5 HP-UX Signals Used for Ada Delay Timing") occur asynchronously. Because of this, they may occur while your code is executing an external interfaced subprogram. For details on protecting your external interfaced subprogram from adverse effects caused by these signals, see the section in the *Ada User's Guide* on "Interfaced Subprograms and Ada's Use of Signals."

F 9.9 Programming in Ada With HP-UX Signals

If you intend to utilize signals in external interfaced subprograms, refer to section F 11.7, "Potential Problems Using Interfaced Subprograms." This version of HP Ada supports the association of an HP-UX *signal, such as* `SIGINT`, with an Ada signal handling procedure (and via such a procedure with a task entry). Refer to section "F 9.7 HP-UX Signals Used for Ada Interrupt Entries" and section "F 12. Interrupt Entries" for additional information.

10

F 10. Limitations

This chapter lists limitations of the compiler and the Ada development environment.

F 10.1 Compiler Limitations

Note It is impossible to give exact numbers for most of the limits listed in this section. The various language features may interact in complex ways to lower the limits.

The numbers represent "hard" limits in simple program fragments devoid of other Ada features.

Limit	Description
255	Maximum number of characters in a source line.
253	Maximum number of characters in a string literal.
255	Maximum number of characters in an enumeration type element.
32767	In an enumeration type, the sum of the lengths of the `IMAGE` attributes of all elements in the type, plus the number of elements in the type, must not exceed this value.
32768	Maximum number of enumeration elements in a single enumeration type (this limit is further constrained by the maximum number of characters in the `IMAGES`s of the elements of an enumeration type, as noted above).
2047	Maximum number of actual compilation units in a library.
2047	Maximum number of "created" units in a single compilation.
1020	Maximum number of units that a single unit can `with`.
1020	Maximum number of units that can be declared `separate` within a single unit.
1020	Maximum number of pragma `ELABORATE` specifications that can be present for units `withed` by a single unit.
1020	Maximum number of `other` dependencies that can exist in a single unit (typically generic instantiations or inlined subprograms). Refer to chapter 3 in the *Ada User's Guide* for additional information on `other` dependencies.
2**31-1	Maximum number of bits in any size computation.
2048	Links in a library.
2048	Libraries in the INSTALLATION family (250 of which are reserved).
2047	Libraries in either the PUBLIC or a user defined family. (For more information, see the *Ada User's Guide*, which discusses families of Ada libraries and the supported utilities (tools) to manage them).

Limit	Description
-	Maximum number of tasks is limited only by heap size.
255	Maximum number of characters in any path component of a file specified for access by the Ada compiler. If a component exceeds 255 characters, NAME_ERROR will be raised.
1023	The maximum number of characters in the entire path to a file specified for access by the Ada compiler. If the size of the entire path exceeds 1023 characters, NAME_ERROR will be raised.

The pathname limits apply to the entire path during and after the resolution of symbolic links and context-dependent files (CDFs) if they appear in the specified path.

The following items are limited only by overflow of internal tables (AIL or HLST tables). All internal data structures of the compiler that previously placed fixed limits are now dynamically created.

- Maximum number of identifiers in a unit. An identifier includes enumerated type identifiers, record field definitions, and (generic) unit parameter definitions.
- Maximum "structure" depth. Structure includes the following: nested blocks, compound statements, aggregate associations, parameter associations, subexpressions.
- Maximum array dimensions. Set to maximum structure depth/10. †
- Maximum number of discriminants in a record constraint. †
- Maximum number of associations in a record aggregate. †
- Maximum number of parameters in a subprogram definition. †
- Maximum expression depth. †
- Maximum number of nested frames. Library-level unit counts as a frame.
- Maximum number of overloads per compilation unit.
- Maximum number of overloads per identifier.

† A limit on the size of tables used in overloading resolution can potentially lower this figure. This limit is set at 500. It reflects the number of possible interpretations of names in any single construct under analysis by the compiler (procedure call, assignment statement, and so on.)

F 10.2 Ada Development Environment Limitations

The following limits apply to the Ada development environment (**ada.umgr(1)**, **ada.fmgr(1)**, and Ada tools).

Limit	Description
200	The number of characters in the actual rooted path of an Ada program **LIBRARY** or **FAMILY** of libraries.
200	The number of characters in the string (possibly after expansion by an HP-UX shell) specifying the name of an Ada program **LIBRARY** or **FAMILY** of libraries. This limit applies to strings (pathname expressions) specified for a **LIBRARY** or **FAMILY** that you submit to tools such as **ada.mklib(1)** or **ada.umgr(1)**.
512	Maximum length of an input line for the tools **ada.fmgr(1)** and **ada.umgr(1)**.
255	The maximum number of characters in any path component of a file specified for access by an Ada development environment tool. If a component exceeds 255 characters, **NAME_ERROR** will be raised.
1023	The maximum number of characters in the entire path to a file specified for access by an Ada program or an Ada development environment tool. If the size of the entire path exceeds 1023 characters, **NAME_ERROR** will be raised.

The pathname limits apply to the entire path during and after the resolution of symbolic links and context-dependent files (CDFs) if they appear in the specified path.

F 10.3 Limitations Affecting User-Written Ada Applications

The Ada compiler and Ada development environment is expected to be used on versions of the HP-UX operating system that support Network File Systems (NFS), diskless HP-UX workstations, long filename file systems and symbolic links to files. To accommodate this diversity within a file system used in both the development and target systems, the HP Ada compiler places some restrictions on the use of the OPEN and CREATE on external files. This section describes those restrictions.

F 10.3.1 Restrictions Affecting Opening or Creating Files

Unless you observe the following restrictions on the size of path components and file names, the OPEN or CREATE call will raise NAME_ERROR in certain situations.

F 10.3.1.1 Restrictions on Path and Component Sizes

The maximum number of characters in any path component of a file specified for access by an Ada program is 255.

The maximum number of characters in the entire path to a file specified for access by an Ada program is 1023.

The pathname limits apply to the entire path during and after the resolution of symbolic links and context-dependent files (CDFs) if they appear in the specified path.

F 10.3.1.2 Additional Conditions that Raise NAME_ERROR

When opening a file, the Ada exception NAME_ERROR will be raised if there are any directories in the rooted path of the file that are not readable by the "effective uid" of the program. This restriction applies to intermediate path components that are encountered during the resolution of symbolic links.

F 10.3.2 Restrictions on TEXT_IO.FORM

The function TEXT_IO.FORM will raise USE_ERROR if it is called with either of the predefined files STANDARD_INPUT or STANDARD_OUTPUT.

F 10.3.3 Restrictions on the Small of a Fixed Point Type

A length clause may be used to specify the value to use for 'SMALL on a fixed point type. However, this implementation requires that the value specified for 'SMALL is a power of two. The compiler rejects a compilation unit with a length clause specification with an IMPLEMENTATION RESTRICTION if 'SMALL is not an exact power of two.

F 10.3.4 Record Type Alignment Clause

A record type alignment clause can specify that a record type is byte, half-word, word, or double-word aligned (specified as 1, 2, 4, or 8 bytes). Ada DS does not support alignments larger than an 8-byte alignment.

11

F 11. Calling External Subprograms From Ada

In Ada, parameters of external interfaced subprograms are passed according to the standard PA-RISC calling conventions (see *PA-RISC Architecture Procedure Calling Convention Reference Manual*). This convention is used by Hewlett-Packard for other language products on the HP 9000 Series 600, 700, and 800 family of computers. The languages described in this section are the HP implementations of HP-PA Assembler, HP C, HP FORTRAN 77, and HP Pascal on the HP-UX Series 600, 700, and 800 systems.

When you specify the interfaced language name, that name is used to select the correct calling conventions for supported languages. Subprograms written in PA-RISC Assembler, HP C, HP FORTRAN 77, and HP Pascal interface correctly with the Ada subprogram caller. This section contains detailed information about calling subprograms written in these languages. If the subprogram is written in a language from another vendor, you must follow the standard calling conventions.

In the Ada implementation of external interfaced subprograms, the three Ada parameter passing modes (`in`, `out`, `in out`) are supported, with some limitations as noted below. Scalar and access parameters of mode `in` are passed by *value*. All other parameters of mode `in` are passed by *reference*. Parameters of mode `out` or `in out` are always passed by reference. (See Table 11-1 and section "F 11.1.2 Access Types" for details.)

Table 11-1. Ada Types and Parameter Passing Modes

Ada Type	Mode Passed By Value	Mode Passed By Reference
SCALAR, ACCESS	`in`	`out`, `in out`
All others except TASK and FIXED POINT		`in`, `out`, `in out`
TASK and FIXED POINT	(not passed)	(not passed)

The values of the following types *cannot* be passed as parameters to an external interfaced subprogram:

- Task types (*Ada RM*, sections 9.1 and 9.2),
- Fixed point types (*Ada RM*, sections 3.5.9 and 3.5.10).

A composite type (an array or record type) is always passed by reference (as noted above). A component of a composite type is passed according to its type classification (scalar, access, or composite).

Only scalar types (enumeration, character, Boolean, integer, or floating point) or access types are allowed for the result returned by an external function subprogram.

Caution All array and record type parameters are passed by reference from Ada code to non-Ada interfaced code. In particular, arrays and records occupying 64 bits or less of storage are passed by reference and are not passed by copy, as by the standard PA-RISC calling convention. Therefore, non-Ada code expecting to receive such array or record parameters must expect to receive them by reference, not by copy. C should declare parameters to be an appropriate pointer type; Pascal should declare parameters to be `VAR` parameters; FORTRAN always expects explicit parameters by reference. Note that array and record type parameters occupying more that 64 bits of storage are passed by reference, both by Ada and by the standard PA-RISC calling convention, and require no special precautions.

Note

There are no checks for consistency between the subprogram parameters (as declared in Ada) and the corresponding external subprogram parameters. Because external subprograms have no notion of Ada's parameter modes, parameters passed by reference are not protected from modification by an external subprogram. Even if the parameter is declared to be only of mode `in` (and not `out` or `in out`) but is passed by reference (that is, an array or record type), the value of the Ada actual parameter can still be modified.

The possibility that the parameter's actual value will be modified by an external interfaced subprogram exists when that parameter is not passed by value. Objects whose attribute `'ADDRESS` is passed as a parameter and parameters passed by reference are not protected from alteration and are subject to modification by the external subprogram. In addition, such objects will have no run-time checks performed on their values upon return from interfaced external subprograms.

Erroneous results may occur if the parameter values are altered in some way that violates Ada constraints for the actual Ada parameter. The responsibility is yours to ensure that values are not modified in external interfaced subprograms in such a manner as to subvert the strong typing and range checking enforced by the Ada language.

Caution

Be very careful to establish the exact nature of the types of parameters to be passed. The bit representations of these types can be different between this implementation of Ada and other languages, or between different implementations of the Ada language. Pay careful attention to the size of parameters because parameters must occupy equal space in the interfaced language. When passing record types, pay particular attention to the internal organization of the elements of a record because Ada semantics do not guarantee a particular order of components. Moreover, Ada compilers are free to rearrange or add components within a record. See section "F 4. Type Representation" for more information.

F 11.1 General Considerations in Passing Ada Types

Section F 11.1 discusses each data type in general terms. Sections F 11.2 through F 11.5 describe the details of interfacing your Ada programs with external subprograms written in PA-RISC Assembler, HP C, HP FORTRAN 77, and HP Pascal. Section F 11.6 provides summary tables.

The Ada types are described in the following order:

- Scalar
  - Integer
  - Enumeration
  - Boolean
  - Character
  - Real
- Access
- Array
- Record
- Task

F 11.1.1 Scalar Types

This section describes general considerations when you are passing scalar types between Ada programs and subprograms written in a different HP language. The class scalar types includes integer, real, and enumeration types. Because character and Boolean types are predefined Ada enumeration types, they are also scalar types.

Scalar type parameters of mode `in` are passed by value. Scalar type parameters of mode `in out` or `out` are passed by reference.

F 11.1.1.1 Integer Types

In Ada, all integers are represented in two's complement form. The type SHORT_SHORT_INTEGER is represented as an 8-bit quantity, the type SHORT_INTEGER is represented as a 16-bit quantity, and the type INTEGER is represented as a 32-bit quantity.

All integer types can be passed to interfaced subprograms. When an integer is used as a parameter for an interfaced subprogram, the call can be made either by reference or by value. If passed by reference, the value of the actual integer parameter is not copied or modified, but a 32-bit address pointer to the integer value is passed. If passed by value, a copy of the actual integer parameter value is passed, based on its size, as per the standard PA-RISC calling convention. If passed in a register, it will be sign extended as required. See sections "F 11.2.1.1 Integer Types and Assembly Language Subprograms", "F 11.3.1.1 Integer Types and HP C Subprograms", "F 11.4.1.1 Integer Types and HP FORTRAN 77 Subprograms", and "F 11.5.1.1 Integer Types and HP Pascal Subprograms"for details specific to interfaced subprograms written in different languages.

Integer types may be returned as function results from external interfaced subprograms.

F 11.1.1.2 Enumeration Types

Values of an enumeration type (*Ada RM*, section 3.5.1) without an enumeration representation clause (*Ada RM*, section 13.3) have an internal representation of the value's position in the list of enumeration literals defining the type. These values are non-negative. The first literal in the list corresponds to an integer value of zero.

An enumeration representation clause can be used to further control the mapping of internal codes for an enumeration identifier. See section "F 4.1 Enumeration Types," for information on enumeration representation clauses.

Values of enumeration types are represented internally as either an 8-, 16-, or 32-bit quantity (see section "F 4.1 Enumeration Types"). When an enumeration value is used as a parameter for an interfaced subprogram, the call can be made either by reference or by value. If passed by reference, the value of the actual enumeration parameter is not copied or modified, but

a 32-bit address pointer to the enumeration value is passed. If passed by value, a copy of the actual enumeration parameter value is passed, based on its size, as per the standard PA-RISC calling convention. If passed in a register, it will be zero extended as required. See sections "F 11.2.1.2 Enumeration Types and Assembly Language", "F 11.3.1.2 Enumeration Types and HP C Subprograms", "F 11.4.1.2 Enumeration Types and HP FORTRAN 77 Subprograms", and "F 11.5.1.2 Enumeration Types and HP Pascal Subprograms"for details specific to interfaced subprograms written in different languages.

Enumeration types may be returned as function results from external interfaced subprograms.

F 11.1.1.3 Boolean Types

Values of the predefined enumeration type BOOLEAN are represented internally as an 8-bit quantity. The Boolean value FALSE is represented by the 8-bit value 2#0000_0000# and the Boolean value TRUE is represented by the 8-bit value 2#0000_0001#. This representation is the same as that of any two-valued enumeration type whose size and internal code values have not been modified with a representation clause.

Boolean values are passed the same as any other enumeration values.

Boolean types can be returned as function results from external interfaced subprograms.

See sections "F 11.2.1.3 Boolean Types and Assembly Language Subprograms", "F 11.3.1.3 Boolean Types and HP C Subprograms", "F 11.4.1.3 Boolean Types and HP FORTRAN 77 Subprograms", and "F 11.5.1.3 Boolean Types and HP Pascal Subprograms"for details specific to interfaced subprograms written in different languages.

F 11.1.1.4 Character Types

The values of the predefined enumeration type CHARACTER are represented as 8-bit values in a range 0 through 127.

Values of the character type are passed as parameters and returned as function results, as are values of any other 8-bit enumeration type.

Character types may be returned as function results from external interfaced subprograms.

See sections "F 11.2.1.4 Character Types and Assembly Language Subprograms", "F 11.3.1.4 Character Types and HP C Subprograms", "F 11.4.1.4 Character Types and HP FORTRAN 77 Subprograms", and "F 11.5.1.4 Character Types and HP Pascal Subprograms"for details specific to interfaced subprograms written in different languages.

F 11.1.1.5 Real Types

Ada fixed point types and Ada floating point types are discussed in the following subsections.

Fixed Point Types

Ada fixed point types (*Ada RM*, sections 3.5.9 and 3.5.10) are not supported as parameters or as results of external interfaced subprograms.

Fixed point types *cannot* be returned as function results from external interfaced subprograms.

Floating Point Types

Floating point values (*Ada RM*, sections 3.5.7 and 3.5.8) in the HP implementation of Ada are of 32 bits (FLOAT) or 64 bits (LONG_FLOAT). These two types conform to the *IEEE Standard for Binary Floating-Point Arithmetic*.

The Ada type FLOAT is a 32-bit real type and is passed as a 32-bit real; this type is *never* extended to a 64-bit real. The Ada type LONG_FLOAT is a 64-bit real type and is passed as a 64-bit real.

Both floating point types can be passed to interfaced subprograms. When a floating point value is used as a parameter for an interfaced subprogram, the call can be made either by reference or by value. If passed by reference, the value of the actual floating point parameter is not copied or modified; a 32-bit address pointer to the floating point value is passed. If passed by value, a copy of the actual floating point parameter value is passed, based on its size, as per the standard PA-RISC calling convention.

See sections "F 11.2.1.5 Real Types and Assembly Language Subprograms", "F 11.3.1.5 Real Types and HP C Subprograms", "F 11.4.1.5 Real Types and HP FORTRAN 77 Subprograms", and "F 11.5.1.5 Real Types and HP Pascal Subprograms"for details specific to interfaced subprograms written in different languages.

Floating point types may be returned as function results from external interfaced subprograms, with some restrictions. See section "F 11.3.1.5 Real Types and HP C Subprograms" for details.

F 11.1.2 Access Types

Values of an access type (*Ada RM*, section 3.8) have an internal representation which is the 32-bit address of the underlying designated object.

If you need to get a SYSTEM.ADDRESS containing the address of the accessed object and have the following declarations

```
type PTR is access <something>;
P: PTR;
A: SYSTEM.ADDRESS;
```

you can just declare

```
function CONV is
   new UNCHECKED_CONVERSION (PTR, SYSTEM.ADDRESS);
```

and then declare

```
A:= CONV(P);
```

This converts the pointer into a SYSTEM.ADDRESS.

An access type object has a value that is the address of the designated object. Therefore, when an access type is passed by value, a copy of this 32-bit address is passed. If an access type object is passed by reference, however, the address of the access type object itself is passed. This will effectively force references to the designated object to be double indirect references.

See Figure 11-1 for details.

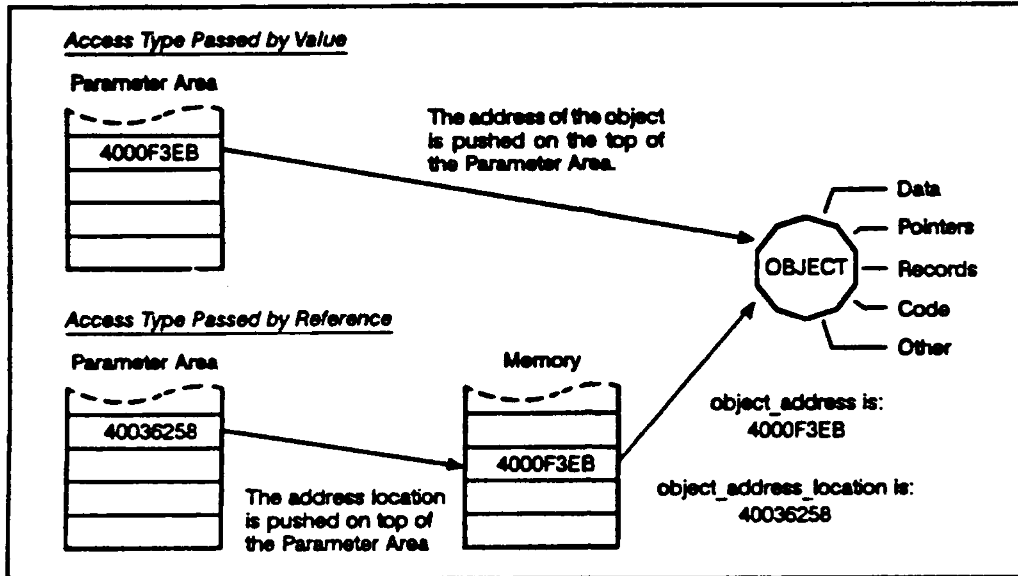


Figure 11-1. Passing Access Types to Interfaced Subprograms

Access types may be returned as function results from external interfaced subprograms.

Ada access types are pointers to Ada objects. In the implementation of HP Ada for the Series 600, 700, and 800 Computer System, an address pointer value will always point at the first byte of storage for the designated object and not at a descriptor for the object. This may not be the case for other implementations of the Ada language and should be considered when Ada source code portability is an issue.

Note If a pointer to an unconstrained array object is passed to interfaced code, the information that describes the run-time constraints needs to be passed explicitly.

F 11.1.3 Array Types

In the HP implementation of Ada, arrays (*Ada RM*, section 3.6) are always passed by reference. The value passed is the address of the first element of the array. When an array is passed as a parameter to an external interfaced subprogram, the usual checks on the consistency of array bounds between the calling program and the called subprogram are not enforced. You are responsible for ensuring that the external interfaced subprogram keeps within the proper array bounds. You may need to explicitly pass the upper and lower bounds for the array type to the external subprogram.

The external subprogram should access and modify such an array in a manner appropriate to the actual Ada type. Note that Ada will *not* range check the values that may have been stored in the array by the external subprogram. In Ada, range checks are only required when assigning an object with a constraint; thus, range checks are not performed when reading the value of an object with a constraint. If an external subprogram modifies elements in an Ada array object, it has the responsibility to ensure that any values stored meet the type constraints imposed by the Ada type.

Array element allocation, layout, and alignment are described in section "F 4.7 Array Types".

Values of the predefined type `STRING` (*Ada RM*, section 3.6.3) are unconstrained arrays and are passed by reference as described above. The address of the first character in the string is passed. You may need to explicitly pass the upper and lower bounds or the length of the string to the external subprogram.

Returning strings from an external interfaced subprogram to Ada (such as `OUT` parameters) is not supported. See section "F 11.3.3 Array Types and HP C Subprograms" for a complete example that shows how to return STRING type information from interfaced subprograms.

Array types *cannot* be returned as function results from external interfaced subprograms. However, an access type to the array type can be returned as a function result.

Caution All array type parameters are passed by reference from Ada code to non-Ada interfaced code. In particular, arrays occupying 64 bits or less of storage are passed by reference and are not passed by copy, as is the standard PA-RISC calling convention. Therefore, non-Ada code expecting to receive such array parameters must expect to receive them by reference, not by copy. C should declare such parameters to be an appropriate pointer type; Pascal should declare such parameters to be `VAR` parameters; FORTRAN always expects explicit parameters by reference. Note that array type parameters occupying more than 64 bits of storage are passed by reference, both by Ada and by the standard PA-RISC calling convention, and require no special precautions.

F 11.1.4 Record Types

Records (*Ada RM*, section 3.7) are always passed by reference in the HP implementation of Ada, passing the 32-bit address of the first component of the record. The external subprogram should access and modify such a record in a manner appropriate to the actual Ada type. Note that Ada will *not* range check the values that may have been stored in the record by the external subprogram. In Ada, range checks are only required when assigning an object with a constraint; thus, range checks are not performed when reading the value of an object with a constraint. If an external subprogram modifies a component in an Ada record object, it has the responsibility to ensure that any values stored meet the type constraints imposed by the Ada type for that component.

When interfacing with external subprograms using record types, it is recommended that you provide a complete record representation clause for the record type. It is also your responsibility to ensure that the external subprogram accesses the record type in a manner that is consistent with the record representation clause. For a complete description of record representation clauses, see section "F 4.8 Record Types".

Caution All record type parameters are passed by reference from Ada code to non-Ada interfaced code. In particular, records occupying 64 bits or less of storage are passed by reference and are not passed by copy, as is the standard PA-RISC calling convention. Therefore, non-Ada code expecting to receive such record parameters must expect to receive them by reference, not by copy. C should declare such parameters to be an appropriate pointer type; Pascal should declare such parameters to be VAR parameters; FORTRAN always expects explicit parameters by reference. Note that record type parameters occupying more than 64 bits of storage are passed by reference, both by Ada and by the standard PA-RISC calling convention, and require no special precautions.

If a record representation clause is *not* used, you should be aware that the individual components of a record may have been reordered internally by the Ada compiler. This means that the implementation of the record type may have components in an different order than the declarative order. Ada semantics do not require a specific ordering of record components.

When interfacing record types with external subprograms, you may want to communicate some or all of the offsets of individual record components. One reason for doing this would be to avoid duplicating the record information in two places: once in your Ada code and again in the interfaced code. Software maintenance is often complicated by this practice.

The attribute `'POSITION` returns the offset of a record component with respect to the starting address of the record. By passing this information to the external subprogram, you can avoid duplicating the record type definition in your external subprogram.

The starting address of a record type can be passed to an external subprogram in one of three ways:

- The record object passed as a parameter (records are always passed by reference).
- The attribute `'ADDRESS` of the record object passed as a parameter.
- A *value* parameter that is of an access type to the record object.

Direct assignment to a discriminant of a record is not allowed in Ada (*Ada RM*, section 3.7.1). A discriminant *cannot* be passed as an actual parameter of mode `out` or `in out`. This restriction applies equally to Ada subprograms and to external interfaced subprograms written in other languages. If an interfaced program is given access to the whole record (rather than individual components), that code should *not* change the discriminant value because that would violate the Ada standard rules for discriminant records.

In Ada, records are packed and variant record parts are overlaid; the size of the record is the longest variant part. If a record contains discriminants or composite components having a dynamic size, the compiler may add implicit components to the record. See section "F 4.8 Record Types" for a complete discussion of these components.

Dynamic components and components whose size depends upon record discriminant values are implemented indirectly within the record by using implicit `'OFFSET` components.

Record types *cannot* be returned as function results from external interfaced subprograms. However, an access type to the record type can be returned as a function result.

F 11.1.5 Task Types

A task type *cannot* be passed to an external procedure or external function as a parameter in Ada. A task type *cannot* be returned as a function result from an external function.

F 11.2 Calling Assembly Language Subprograms

When calling interfaced assembly language subprograms, specify the named external subprogram in a compiler directive:

pragma INTERFACE (ASSEMBLER, *Ada_subprogram_name*);

Note that the language type specification is ASSEMBLER and not ASSEMBLY. This description refers to the HP assembly language for the PA-RISC processor family upon which the Series 600, 700, and 800 family is based.

Interfaced subprograms written in PA-RISC Assembly Language that conform to the PA-RISC procedure calling conventions can be called from Ada with no special precautions. See the *PA-RISC Architecture Procedure Calling Convention Reference Manual* and the *Assembly Language Reference Manual* for additional information.

Only scalar types (integer, floating point, character, Boolean, and enumeration types) and access types are allowed as result types for an external interfaced function subprogram written in PA-RISC Assembly Language.

F 11.2.1 Scalar Types and Assembly Language Subprograms

See section "F 11.1.1 Scalar Types" for details.

F 11.2.1.1 Integer Types and Assembly Language Subprograms

See section "F 11.1.1.1 Integer Types" for details.

F 11.2.1.2 Enumeration Types and Assembly Language

Subprograms

See section "F 11.1.1.2 Enumeration Types" for details.

F 11.2.1.3 Boolean Types and Assembly Language Subprograms

See section "F 11.1.1.3 Boolean Types" for details.

F 11.2.1.4 Character Types and Assembly Language Subprograms

See section "F 11.1.1.4 Character Types" for details.

F 11.2.1.5 Real Types and Assembly Language Subprograms

See section "F 11.1.1.5 Real Types" for details.

F 11.2.2 Access Types and Assembly Language Subprograms

See section "F 11.1.2 Access Types" for details.

F 11.2.3 Array Types and Assembly Language Subprograms

See section "F 11.1.3 Array Types" for details.

F 11.2.4 Record Types and Assembly Language Subprograms

See section "F 11.1.4 Record Types" for details.

F 11.3 Calling HP C Subprograms

When calling interfaced HP C subprograms, the form

pragma INTERFACE (C, *Ada_subprogram_name*)

is used to identify the need to use the HP C parameter passing conventions.

To call the following HP C subroutine

```
void c_sub (val_parm, ref_parm)
int val_parm;
int *ref_parm;
{
    ...
}
```

Ada requires an interfaced subprogram declaration:

```
procedure C_SUB (VAL_PARAM : in INTEGER;
                 REF_PARAM : in out INTEGER);
pragma INTERFACE (C, C_SUB);
```

In the above example we provided the Ada subprogram identifier C_SUB to the pragma INTERFACE. If a pragma INTERFACE_NAME is not supplied, the HP C subprogram name is the name of the Ada subprogram specified in the pragma INTERFACE, with all alphabetic characters shifted to lowercase.

Note that the parameter in the preceding example, VAL_PARAM, must be of mode in to match the parameter definition for val_parm found in the HP C subroutine. Likewise, REF_PARAM, must be of mode in out to correctly match the C definition of *ref_parm. Also, note that the names for parameters *do not* need to match exactly. However, the mode of access and the data type *must* be correctly matched, but there is no compile-time or run-time check that can ensure that they match. It is your responsibility to ensure their correctness.

You must use pragma INTERFACE_NAME whenever the HP C subprogram name contains characters not acceptable within Ada identifiers or when the HP C subprogram name contains uppercase letter or letters. You can also use a pragma INTERFACE_NAME if you want your Ada subprogram name to be different than the HP C subprogram name.

Note that the Ada compiler does *not* automatically convert 32-bit real parameters to 64-bit real parameters. See section "F 11.3.1.5 Real Types and HP C Subprograms" for details.

Only scalar types (integer, floating point, character, Boolean, and enumeration types) and access types are allowed as result types for an external interfaced function subprogram written in HP C.

When binding and linking Ada programs with interfaced subprograms written in HP C, the libraries `libc.a` `libM.a`, and `libcl.a` are usually required. The Ada binder automatically provides the `-lM -lc -lcl` directives to the linker. You are not required to specify "`-lM -lc -lcl`" when binding and linking the Ada program on the `ada(1)` command line.

For more information about C language interfacing, see the *HP C/HP-UX Reference Manual* and the *HP C Programmer's Guide*. For general information about passing Ada types, see section "F 11.1 General Considerations in Passing Ada Types".

F 11.3.1 Scalar Types and HP C Subprograms

See section "F 11.1.1 Scalar Types" for details.

F 11.3.1.1 Integer Types and HP C Subprograms

See section "F 11.1.1.1 Integer Types" for details.

When passing integers by reference, note that an Ada SHORT_SHORT_INTEGER (eight bits) actually corresponds with the HP C type char, because C treats this type as a numeric type.

Table 11-2 summarizes the integer correspondence between Ada and C.

Table 11-2. Ada versus HP C Integer Correspondence

Ada	HP C	Bit Length
CHARACTER	char	8
SHORT_SHORT_INTEGER	char	8
SHORT_INTEGER	short and short int	16
INTEGER	int, long, and long int	32

All Ada integer types are allowed for the result returned by an external interfaced subprogram written in HP C if care is taken with respect to differences in the interpretation of 8-bit quantities.

F 11.3.1.2 Enumeration Types and HP C Subprograms

See section "F 11.1.1.2 Enumeration Types" for details.

HP C enumeration types have the same representation as Ada enumeration types. They both are represented as unsigned integers beginning at zero. In HP C, the size of an enumeration type is always 32 bits. When HP C passes enumeration types as *value* parameters, the values are zero extended to 32 bits. Because Ada also performs the zero extension to 32 bits for enumeration type values, they will be in the correct form for HP C subprograms. If a representation specification applies to the Ada enumeration type, the value specified by the representation clause (not the 'POS value) will be passed to the HP C routine.

F 11.3.1.3 Boolean Types and HP C Subprograms

See section "F 11.1.1.3 Boolean Types" for details.

Booleans are passed as other enumeration types are passed; see section "F 11.3.1.2 Enumeration Types and HP C Subprograms" for details.

The type Boolean is not defined in HP C and the Ada representation of Booleans does not directly correspond to any type in HP C. However, an Ada Boolean could be represented in C with an appropriate two-valued enumeration type or with an HP C integer type.

Boolean types are allowed for the result returned by an external interfaced subprogram written in HP C, when care is taken to observe the internal representation.

F 11.3.1.4 Character Types and HP C Subprograms

See section "F 11.1.1.4 Character Types" for details.

The Ada predefined type `CHARACTER` and any of its subtypes correspond with the type `char` in HP C. Both the Ada and HP C types have the same internal representation and size. However, in Ada the type `CHARACTER` is constrained to be within the 128 character ASCII standard.

F 11.3.1.5 Real Types and HP C Subprograms

This section discusses passing fixed point types and floating point types to HP C.

Fixed Point Types

Ada fixed point types are not supported as parameters or as results of external subprograms. Ada fixed point types *cannot* be returned as function results from interfaced subprograms written in HP C.

Floating Point Types

When HP C is operating in compatibility mode (non-ANSI mode), the default calling convention for passing parameters of floating point types by *value* requires that 32-bit single precision reals be converted to 64-bit double precision reals before being passed.

When HP C is operating in ANSI conformant mode (or in compatibility *mode with the +r flag specified*), 32-bit single precision reals are passed as parameters without being converted to 64-bit double precision.

Consequently, an interface parameter of type `FLOAT` or of a type derived from a type whose base type is `FLOAT`. can only be passed directly to `float` parameters of HP C code compiled in ANSI conformant mode or in compatibility mode with `+r` specified.

To interface with compatibility mode HP C code (no `+r` specified). the Ada type `LONG_FLOAT`, or a type derived from a type whose base type is `LONG_FLOAT`, must be used for all parameters of HP C type `float`.

This limitation on passing the Ada type `FLOAT` only applies to parameters that are of the type `FLOAT` or derived from a type whose base type is `FLOAT`. A parameter of a composite type, such as an array or record, can have components that are of the type `FLOAT`. Also, the type `FLOAT` can be passed by reference to an external HP C subprogram. The HP C calling convention does not require conversion in these cases in any compiler mode.

F 11.3.2 Access Types and HP C Subprograms

See section "F 11.1.2 Access Types" for details.

F 11.3.3 Array Types and HP C Subprograms

See section "F 11.1.3 Array Types" for details.

Note that constrained Ada arrays with SHORT_SHORT_INTEGER or with 8-bit enumeration type components can be most conveniently associated with an HP C type of the form char[] or char *.

In Ada, the predefined type STRING is an unconstrained array type. It is represented in memory as a sequence of consecutive characters without any gaps in between the characters. In HP C, the string type is represented as a sequence of characters that is terminated with an ASCII null character (\000). You will need to append a null character to the end of an Ada string if that string is to be sent to an external interfaced HP C subprogram. When retrieving the value of an HP C string object for use as an Ada string, you will need to dynamically allocate a copy of the HP C string. The HP C type char * is not compatible with the unconstrained array type STRING that is used by Ada.

The examples on the following pages illustrate the handling of strings in HP C and in Ada. In the first example, an Ada string is passed to HP C. Note the need to explicitly add a null character to the end of the string so that string is in the form that HP C expects for character strings.

HP C routine:

```
/*  Receiving an Ada string that has an ASCII.NULL appended
    to it in this C routine
*/

void receive_ada_str (var_str)
   char *var_str;
{
   printf ("C: Received value was : %s \n", var_str);
}
```

Ada routine:

```
-- passing an Ada string to a C routine

procedure SEND_ADA_STR is

   -- Declare an interfaced procedure that sends an
   -- Ada-String to a C-subprogram

   procedure RECEIVE_ADA_STR ( VAR_STR : STRING);
   pragma INTERFACE (C, RECEIVE_ADA_STR);

begin -- SEND_ADA_STR

   -- Test the passing of an Ada string to a C routine
   RECEIVE_ADA_STR ( "Ada test string sent to C " & ASCII.NUL);

end SEND_ADA_STR;
```

In the second example, a C string is converted to an Ada string. Note that Ada must compute the length of the C string and then it must dynamically allocate a new copy of the C string.

HP C routine:

```
/* Sending a C string value back to an Ada program */

char *send_c_str()
{
   char *local_string;

   local_string = "a C string for Ada.";
   return local_string;
}
```

Ada routine:

```
--------------------------------------------------------------
-- We import several useful functions from the package SYSTEM
--    the generic function FETCH_FROM_ADDRESS
--        to read a character value given an address
--    the function "+"(address,integer)
--        to allow us to index consecutive addresses
--
--      ( See section F 3.1, for the complete specification
--        of the package SYSTEM )
--------------------------------------------------------------
with SYSTEM;
with TEXT_IO;
procedure READ_C_STRING is

   type C_STRING is access CHARACTER;
                -- This is the C type char *

   type A_STRING is access STRING;
                -- The Ada type pointer to STRING
```

```
-- Declare an interfaced procedure that returns a pointer
-- to a C string (actually a pointer to a character)
function SEND_C_STR return C_STRING;
pragma INTERFACE (C, SEND_C_STR);

function FETCH_CHAR is
   new SYSTEM.FETCH_FROM_ADDRESS (TARGET => CHARACTER);
   -- Create a non-generic instantiation of the function FETCH

function C_STRING_LENGTH (SRC : C_STRING) return NATURAL is
   use SYSTEM; -- import the "+"(address,offset) operator
   LEN   : NATURAL := 0;
   START : SYSTEM.ADDRESS;
   CUR   : CHARACTER;
begin
   START := SRC.all'ADDRESS;
   loop
      CUR := FETCH_CHAR (FROM => START + OFFSET (LEN));
      exit when CUR = ASCII.NUL;
      LEN := LEN + 1;
   end loop;
   return LEN;
end C_STRING_LENGTH;
```

```
function CONVERT_TO_ADA (SRC : C_STRING) return A_STRING is
   use SYSTEM; -- import the "+"(address,offset) operator
   A_STORAGE : A_STRING;
   LEN       : NATURAL;
   C_START   : SYSTEM.ADDRESS;
   C_CUR     : CHARACTER;
begin
   LEN := C_STRING_LENGTH (SRC);
   A_STORAGE := new STRING (1 .. LEN);
   C_START   := SRC.all'ADDRESS;
   for INX in 0 .. LEN - 1 loop
      C_CUR := FETCH_CHAR (FROM => C_START + OFFSET (INX));
      A_STORAGE.all (INX + 1) := C_CUR;
   end loop;
   return A_STORAGE;
end CONVERT_TO_ADA;

begin -- Start of READ_C_STRING
   declare
      A_RESULT : A_STRING;
      C_RESULT : C_STRING;
   begin
                      -- Call the external C subprogram
      C_RESULT := SEND_C_STR;

                      -- Convert to an access Ada STRING
      A_RESULT := CONVERT_TO_ADA( C_RESULT );

                      -- Print out the result.
      TEXT_IO.PUT_LINE( A_RESULT.all );
   end;
end READ_C_STRING;
```

F 11.3.4 Record Types and HP C Subprograms

See section "F 11.1.4 Record Types" for details.

Ada records can be passed as parameters to external interfaced subprograms written in HP C if care is taken regarding the record layout and access to record discriminant values. See section "F 4.8 Record Types" for information on record type layout.

F 11.4 Calling HP FORTRAN 77 Language Subprograms

When calling interfaced HP FORTRAN 77 subprograms, the following form is used:

pragma INTERFACE(FORTRAN, *Ada_subprogram_name*)

This form is used to identify the need for HP FORTRAN 77 parameter passing conventions.

To call the HP FORTRAN 77 subroutine

```
SUBROUTINE fsub (parm)
INTEGER*4 parm
. . .
END
```

you need this interfaced subprogram declaration in Ada:

```
procedure FSUB (PARM : in out INTEGER);
pragma INTERFACE (FORTRAN, FSUB);
```

The external name specified in the Ada interface declaration can be any Ada identifier. If the Ada identifier differs from the FORTRAN 77 subprogram name, **pragma** INTERFACE_NAME is required.

Note that the parameter in the example above is of mode in out. In HP FORTRAN 77, all user-declared parameters are always passed by reference; therefore, mode in out or mode out must be used for scalar type parameters. The HP FORTRAN 77 compiler might expect some implicit parameters that are passed by value and not by reference. See section "F 11.4.4 String Types and HP FORTRAN 77 Subprograms" for details.

Only scalar types (integer, floating point, and character types) are allowed for the result returned by an external interfaced function subprogram written in HP FORTRAN 77. Access type results are not supported. For more information, see the following manuals:

- *HP FORTRAN 77/HP-UX Reference Manual*
- *HP FORTRAN 77/HP-UX Programmer's Reference*
- *HP FORTRAN 77/Quick Reference Guide*

For general information about passing types to interfaced subprograms, see section "F 11.1 General Considerations in Passing Ada Types".

F 11.4.1 Scalar Types and HP FORTRAN 77 Subprograms

FORTRAN expects all user-declared parameters to be passed by reference. Ada scalar type parameters will only be passed by reference if declared as mode in out or out; therefore, no scalar type parameters to a FORTRAN interface routine should be declared as mode in (except for certain implicit parameters; see section "F 11.4.4 String Types and HP FORTRAN 77 Subprograms" for details.) No error will be reported by Ada, but you will most likely get unexpected results.

F 11.4.1.1 Integer Types and HP FORTRAN 77 Subprograms

See section "F 11.1.1.1 Integer Types" for details.

See section "F 11.1.1 Scalar Types" for details.

Table 11-3 summarizes the correspondence between integer types in Ada and HP FORTRAN 77.

Table 11-3. Ada versus HP FORTRAN 77 Integer Correspondence

Ada	HP FORTRAN 77	Bit Length
SHORT_SHORT_INTEGER	BYTE	8
SHORT_INTEGER	INTEGER*2	16
INTEGER	INTEGER*4	32

The compatible types are the same for procedures and functions. Compatible Ada integer types are allowed for the result returned by an external interfaced function subprogram written in HP FORTRAN 77.

Ada semantics do not allow parameters of mode in out to be passed to function subprograms. Therefore, for Ada to call HP FORTRAN 77 external interfaced function subprograms, each scalar parameter's address must be passed. The use of the supplied package SYSTEM facilitates this passing of the

object's address. The parameters in an HP FORTRAN 77 external function must be declared as in the example below:

```
with SYSTEM;
VAL1   : INTEGER;  -- a scalar type
VAL2   : FLOAT  ;  -- a scalar type
RESULT : INTEGER;
function FTNFUNC (PARM1, PARM2 : SYSTEM.ADDRESS) return INTEGER;
```

The external function must be called from within Ada as follows:

```
RESULT := FTNFUNC (VAL1'ADDRESS, VAL2'ADDRESS);
```

F 11.4.1.2 Enumeration Types and HP FORTRAN 77 Subprograms

See section "F 11.1.1.2 Enumeration Types" for details.

The HP FORTRAN 77 language does not support enumeration types. However, objects that are elements of an Ada enumeration type can be passed to an HP FORTRAN 77 integer type because the underlying representation of an enumeration type is an integer. The appropriate FORTRAN type (BYTE, INTEGER*2, or INTEGER*4) should be chosen to match the size of the Ada enumeration type. If a representation specification applies to the Ada enumeration type, the value specified by the representation clause (not the 'POS value) will be passed to the FORTRAN routine.

F 11.4.1.3 Boolean Types and HP FORTRAN 77 Subprograms

See section "F 11.1.1.3 Boolean Types" for details.

An Ada Boolean that has the default 8-bit size is compatible with the default mode HP FORTRAN 77 type `LOGICAL*1` both as a parameter and as a function result.

An Ada Boolean type with a representation specification for a larger size (16 or 32 bits) is *not* compatible with the larger sized HP FORTRAN 77 logical types (`LOGICAL*2` or `LOGICAL*4`). Such Ada Booleans can be passed to the appropriately sized FORTRAN integer type (`INTEGER*2` or `INTEGER*4`) and treated as integers that have the value of `'POS` of the Ada Boolean value.

If the HP FORTRAN 77 routine is compiled with one of the HP FORTRAN 77 options that changes the size or representation of logical types to other than the default, you will have to determine what Ada types, if any, are compatible with the altered FORTRAN behavior by consulting the appropriate FORTRAN documentation.

F 11.4.1.4 Character Types and HP FORTRAN 77 Subprograms

See section "F 11.1.1.4 Character Types" for details.

There is no one-to-one mapping between an Ada character type and any HP FORTRAN 77 character type. An Ada character type can be passed to HP FORTRAN 77 or returned from HP FORTRAN 77 using one of several methods.

HP FORTRAN 77 considers all single character parameters to be single-element character arrays. The method that HP FORTRAN 77 uses to pass character arrays is described in section "F 11.4.4 String Types and HP FORTRAN 77 Subprograms". The method requires that an implicit *value* parameter be passed to indicate the size of the character array. Because HP FORTRAN 77 uses this method for passing character types, it might be more convenient to convert Ada character types into Ada strings and follow the rules that govern passing Ada string types to HP FORTRAN 77.

An Ada character that has the default 8-bit size can be passed to a default mode HP FORTRAN 77 parameter of type `CHARACTER*1`. This can be done if the interface declaration specifies the additional size parameters that HP FORTRAN 77 implicitly expects and passes the constant value one (the size of the character) when the HP FORTRAN 77 subprogram is called. See section "F 11.4.4 String Types and HP FORTRAN 77 Subprograms" for an example of implicit size parameters for strings; to pass an Ada character instead of a string, simply use the Ada character type in the Ada interface declaration in place of the Ada string type and `CHARACTER*1` in the HP FORTRAN 77 declaration in place of the `CHARACTER *(*)`. Note that the size parameter or parameters are not specified in the HP FORTRAN 77 subprogram declaration; they are implicit parameters that are expected by the HP FORTRAN 77 subprogram for each character array (or character) type parameter.

An Ada character type that has the default size *cannot* be returned from an HP FORTRAN 77 function that has a result type of `CHARACTER*1` (it can be returned as a `BYTE`; see below for details).

An Ada character type that has the default 8-bit size can also be passed to an HP FORTRAN 77 parameter of type `BYTE` without having to pass the additional length parameter. The `BYTE` will have the value of `'POS` of the Ada character value.

An Ada character type that has the default size can also be returned from an HP FORTRAN 77 function that has a return type of BYTE. The BYTE to be returned should be assigned the 'POS value of the desired Ada character.

An Ada character type with a representation specification for a larger size (16 or 32 bits) is *not* compatible with any HP FORTRAN 77 character type. Such Ada characters can be passed to the appropriately sized FORTRAN integer type (INTEGER*2 or INTEGER*4) and treated as integers that have the value of 'POS of the Ada character value.

F 11.4.1.5 Real Types and HP FORTRAN 77 Subprograms

This section discusses passing fixed and floating point types to subprograms written in FORTRAN.

Fixed Point Types

Ada fixed point types are not supported as parameters or as results of external interfaced subprograms written in HP FORTRAN 77. Ada fixed point types *cannot* be returned as function results from external interfaced subprograms written in HP FORTRAN 77.

Floating Point Types

See section "F 11.1.1.5 Real Types" for details.

The Ada type FLOAT corresponds to the REAL*4 format in HP FORTRAN 77. The Ada type LONG_FLOAT corresponds to the HP FORTRAN 77 type DOUBLE PRECISION (or REAL*8).

There is no Ada type that corresponds to the HP FORTRAN 77 type REAL*16.

F 11.4.2 Access Types and HP FORTRAN 77 Subprograms

See section "F 11.1.2 Access Types" for details.

Ada access types have no meaning in HP FORTRAN 77 subprograms because the types are address pointers to Ada objects. The implementation value of an Ada parameter of type ACCESS may be passed to an HP FORTRAN 77 procedure. The parameter in HP FORTRAN 77 is seen as INTEGER*4. The object pointed to by the access parameter has no significance in HP FORTRAN 77; the access parameter value itself would be useful only for comparison operations to other access values.

HP FORTRAN 77 can return an INTEGER*4 and the Ada program can declare an access type as the returned value type (it will be a matching size because in Ada, an access type is a 32-bit quantity.) However, care should be taken that the returned value can actually be used by Ada in a meaningful manner.

F 11.4.3 Array Types and HP FORTRAN 77 Subprograms

See section "F 11.1.3 Array Types" for details.

Arrays whose components have an HP FORTRAN 77 representation can be passed as parameters between Ada and interfaced external HP FORTRAN 77 subprograms. For example, Ada arrays whose components are of types INTEGER, SHORT_INTEGER, FLOAT, LONG_FLOAT, or CHARACTER may be passed as parameters.

Array types *cannot* be returned as function results from external HP FORTRAN 77 subprograms. However, an access type to the array type can be returned as a function result.

Caution Arrays with multiple dimensions are implemented differently in Ada and HP FORTRAN 77. To obtain the same layout of components in memory as a given HP FORTRAN 77 array, the Ada equivalent must be declared and used with the dimensions in reverse order.

Consider the components of a 2-row by 3-column matrix, declared in HP FORTRAN 77 as

```
INTEGER*4  a(2,3)
```

or

```
INTEGER*4 a(1:2,1:3)
```

This array would be stored by HP FORTRAN 77 in the following order:

```
a(1,1), a(2,1), a(1,2), a(2,2), a(1,3), a(2,3)
```

This is referred to as storing in *column major order*; that is, the first subscript varies most rapidly, the second varies next most rapidly, and so forth, and the last varies least rapidly.

Consider the components of a 2-row by 3-column matrix, declared in Ada as:

```
A : array (1..2, 1..3) of INTEGER;
```

This array would be stored by Ada in the following order:

```
A(1,1), A(1,2), A(1,3), A(2,1), A(2,2), A(2,3)
```

This is referred to as storing in *row major order*; that is, the last subscript varies most rapidly, the next to last varies next most rapidly, and so forth, while the first varies least rapidly. Clearly the two declarations in the different languages are not equivalent. Now, consider the components of a 2-row by 3-column matrix, declared in Ada as:

```
A : array (1..3, 1..2) of INTEGER;
```

Note the reversed subscripts compared with the FORTRAN declaration. This array would be stored by Ada in the following order:

```
A(1,1), A(1,2), A(2,1), A(2,2), A(3,1), A(3,2)
```

If the subscripts are reversed, the layout would be

```
A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3)
```

which is identical to the HP FORTRAN 77 layout. Thus, either of the language declarations could declare its component indices in *reverse* order to be compatible.

To illustrate that equivalent multi-dimensional arrays require a reversed order of dimensions in the declarations in HP FORTRAN 77 and Ada, consider the following:

The Ada statement

```
FOO : array (1..10,1..5,1..3) of FLOAT;
```

is equivalent to the HP FORTRAN 77 declaration

```
REAL*4  FOO(3,5,10)
```

or

```
REAL*4  FOO(1:3,1:5,1:10)
```

Both Ada and HP FORTRAN 77 store a one-dimensional array as a linear list.

F 11.4.4 String Types and HP FORTRAN 77 Subprograms

When a string item is passed as an argument to an HP FORTRAN 77 subroutine from within HP FORTRAN 77, extra information is transmitted in hidden (implicit) parameters. The calling sequence includes a hidden parameter (for each string) that is the actual length of the ASCII character sequence. This implicit parameter is passed in addition to the address of the ASCII character string. The hidden parameter is passed by value, not by reference.

These conventions are different from those of Ada. For an Ada program to call an external interfaced subprogram written in HP FORTRAN 77 with a string type parameter, you must explicitly pass the length of the string object. The length must be declared as an Ada 32-bit integer parameter of mode **in**.

The following example illustrates the declarations needed to call an external subroutine having a parameter profile of two strings and one floating point variable.

```
procedure FTNSTR is
   SA: STRING(1..6):= "ABCDEF";
   SB: STRING(1..2):= "GH";
   FLOAT_VAL: FLOAT:=  1.5;
   LENGTH_SA, LENGTH_SB : INTEGER;

  procedure FEXSTR ( S1  : STRING;         -- passed by reference
                     LS1 : in INTEGER;     -- len of string S1,
                                           -- must be IN
                     F   : in out FLOAT;   -- must be IN OUT
                     S2  : STRING;         -- passed by reference
                     LS2 : in INTEGER);    -- len of string S2,
                                           -- must be IN

   pragma INTERFACE (FORTRAN, FEXSTR);

begin -- procedure FTNSTR
  LENGTH_SA := SA'LENGTH;
  LENGTH_SB := SB'LENGTH;

FEXSTR (SA, LENGTH_SA, FLOAT_VAL, SB, LENGTH_SB);
end FTNSTR;
```

Note Note that the string lengths immediately follow the corresponding string parameter. The string lengths must be passed by value, *not* by reference.

The HP FORTRAN 77 external subprogram is the following:

```
SUBROUTINE Fextr (s1, r, s2)
CHARACTER *(*) s1, s2
REAL*4  r
  ...
END
```

Returning A String From FORTRAN

It is not possible to declare, in Ada, an external FORTRAN function that returns a result of type STRING (character*N or character*(*) in FORTRAN). However, such a FORTRAN function can be accessed from Ada by declaring the function to be an Ada procedure with two additional initial parameters. The first parameter should be declared as an out parameter of a constrained string type; the second parameter should be declared as an in parameter of type INTEGER. The string that is to hold the result is passed as the first parameter, and the length of that first parameter (the number of characters that FORTRAN can safely return in that first parameter string) is passed as the second parameter.

If the maximum number of characters specified by the second parameter is greater than the number of characters in the string being returned as the FORTRAN function result, the Ada string will be padded with blanks out to the number of characters specified as the second parameter. If the maximum number of characters specified by the second parameter is less than the number of characters in the string being returned as the FORTRAN function result, only the number of characters specified as the second parameter will be returned in the Ada string.

11

The following Ada program calls a FORTRAN function that returns a STRING function result:

```
procedure FORTRAN_STRING_FUNC is

subtype RESULT is STRING (1..80)

procedure FORTRAN_FOO (RES: out    RESULT;
                       MAX: in     INTEGER;
                       X  : in out INTEGER;
                       Y  : in out INTEGER);

pragma INTERFACE      (FORTRAN, FORTRAN FOO);
pragma INTERFACE_NAME (FORTRAN_FOO, "foo");

S : RESULT;
A : INTEGER;
B : INTEGER;

begin --FORTRAN_STRING_FUNC
  A := 28;
  B := 496;
  FORTRAN_FOO (S, S'LENGTH, A, B);
end FORTRAN_STRING_FUNC;
```

The FORTRAN function looks like this:

```
CHARACTER *(*) FUNCTION foo (x,y)
INTEGER*4 x,y
...
foo = 'RETURN THIS STRING TO ADA'
RETURN
END
```

F 11.4.5 Record Types and HP FORTRAN 77 Subprograms

See section "F 11.1.4 Record Types" for details.

Ada records may be passed as parameters to external interfaced subprograms written in HP FORTRAN if care is taken regarding the record layout and access to record discriminant values. See section "F 4.8 Record Types" for information on record type layout.

Record types are *not* allowed as function results in HP FORTRAN functions.

F 11.4.6 Other FORTRAN Types

The HP FORTRAN 77 types COMPLEX, COMPLEX*8, DOUBLE COMPLEX, and COMPLEX*16 have no direct counterparts in Ada. However, it is possible to declare equivalent types using either an Ada array or an Ada record type. For example, with type COMPLEX in HP FORTRAN 77, a simple Ada equivalent is a user-defined record:

```
type COMPLEX is
   record
      Real : FLOAT;
      Imag : FLOAT;
   end record;
```

Similarly, an HP FORTRAN 77 double complex number could be represented with the two record components declared as Ada type LONG_FLOAT.

While it is *not* possible to declare an Ada external function that returns the above record type, an Ada procedure *can* be declared with an out parameter of type COMPLEX. The Ada procedure would then need to interface with an HP FORTRAN 77 subroutine. which would pass the result back using an in out or out parameter.

F 11.5 Calling HP Pascal Language Subprograms

When calling interfaced HP Pascal subprograms, the form

pragma INTERFACE (Pascal, *Ada_subprogram_name*)

is used to identify the need to use the HP Pascal parameter passing conventions.

To call the following HP Pascal subroutine

```
module modp;
export
   procedure p_subr (val_parm : integer;
                     var ref_parm : integer);

implement
   procedure p_subr (val_parm : integer;
                     var ref_parm : integer);
   begin
      . . .
   end;
end.
```

Ada would use the interfaced subprogram declaration:

```
procedure P_SUB (VAL_PARAM : in INTEGER;
                 REF_PARAM : in out INTEGER);
pragma INTERFACE (Pascal, P_SUB)
```

In the above example we provided the Ada subprogram identifier P_SUB to the pragma INTERFACE.

Note that the parameter in the example, VAL_PARAM, must be of mode in to match the parameter definition for val_parm found in the HP Pascal subroutine. Likewise. REF_PARAM, must be of mode in out to correctly match the HP Pascal definition of var ref_parm. Also, note that the names for parameters *do not* need to match exactly. However, the mode of access and the data type *must* be correctly matched, but there is no compile-time or run-time check that can ensure that they match. It is your responsibility to ensure their correctness.

When Ada interfaces to HP Pascal, it refers to the HP Pascal procedure or function by the procedure or function name. In the above example, the pragma INTERFACE was sufficient to specify that name, although a pragma INTERFACE_NAME could also have been used (and would be necessary if the name given to the Ada routine did not map correctly to the desired HP Pascal name). Because Ada uses only the HP Pascal procedure or function name, there is a difficulty if that name is not unique.

The names of the procedures and functions declared within an HP Pascal module must be unique within a single module. A given module can only contain one procedure or function name (for example, FOO), but another module could also contain a procedure or function named FOO. If a single program uses both modules, it is necessary to properly resolve references to FOO. To properly resolve such references, procedure and function names in modules are qualified with the name of the module that contains them. This qualification is internal to the object file and is not accessible to user code. The linker (ld(1)) uses the qualification information to resolve references by HP Pascal code to identically named procedures or functions.

This qualification mechanism poses a difficulty when attempting to interface Ada to HP Pascal because Ada can only specify the unqualified HP Pascal procedure or function name. There will be no difficulty if the HP Pascal procedure or function being called has a name that is unique within all the HP Pascal modules used in the Ada program (if the qualification mechanism is not needed for the name). If the procedure or function name is not unique, the linker (ld(1)) will, without producing an error or warning, select one (usually the first one) of the multiple HP Pascal procedures or functions that it encounters during the link that has the name specified by Ada. As this unpredictable selection is likely to lead to an incorrect program, interfacing to HP Pascal procedures or functions that are not uniquely named is not recommended.

For more information on Pascal interfacing, see the *HP Pascal/HP-UX Reference Manual*. Additional information is available in the *HP-UX Portability Guide*.

For Pascal, scalar and access parameters of mode in are passed by value; the value of the parameter object is copied and passed. All other types of in parameters (arrays and records) and parameters of mode out and in out are passed be reference; the address of the object is passed. This means that, in

general, Ada in parameters correspond to Pascal value parameters, while Pascal **var** parameters correspond to the Ada parameters of either mode in out or mode out.

Only scalar types (integer, floating point, character, Boolean, and enumeration types) and access types are allowed for the result returned by an external interfaced Pascal function subprograms.

For general information about passing parameters to interfaced subprograms, see section "F 11.1 General Considerations in Passing Ada Types".

F 11.5.1 Scalar Types and HP Pascal Subprograms

See section "F 11.1.1 Scalar Types" for details.

F 11.5.1.1 Integer Types and HP Pascal Subprograms

See section "F 11.1.1.1 Integer Types" for details.

Integer types are compatible between Ada and HP Pascal provided their ranges of values are identical. Table 11-4 shows corresponding integer types in Ada and HP Pascal.

Table 11-4. Ada versus HP Pascal Integer Correspondence

Ada	HP Pascal	Bit Length
predefined type INTEGER	predefined type integer	32
predefined type SHORT_INTEGER	predefined type shortint or user type I16 = 0..65535;	16
predefined type SHORT_SHORT_INTEGER	user-defined type type I8 = 0..255;	8

Note In HP Pascal, any integer subrange that has a negative lower bound is always implemented in 32 bits. Integer subranges with a non-negative lower bound are implemented in 8-bits if the upper bound is 255 or less, in 16-bits if the upper bound is 65535 or less, and in 32-bits if the upper bound is greater than 65535. Therefore, in table Table 11-4, the user-defined subrange 0 ... 255 is shown as the HP Pascal equivalent to the Ada type `SHORT_SHORT_INTEGER`; however, the Ada type is a signed type with the range -128..127. To convert the unsigned value back to a signed value in HP Pascal, if the unsigned value is greater than 127, you will need to subtract 256 to obtain the actual negative value.

Whether passed from Ada to HP Pascal by value or by reference, the appropriate HP Pascal type, as shown in Table 11-4, must be used to properly access the Ada integer value from HP Pascal.

All Ada integer types are allowed for the result returned by an external interfaced subprogram written in HP Pascal if care is taken with respect to ranges defined for integer quantities.

F 11.5.1.2 Enumeration Types and HP Pascal Subprograms

See section "F 11.1.1.2 Enumeration Types" for details.

Ada and HP Pascal have similar implementations of enumeration types. In Ada and HP Pascal, enumeration types can have a size of 8, 16, or 32 bits. However, Ada normally considers enumeration types to be signed quantities and HP Pascal considers them to be unsigned. Table 11-5 shows corresponding enumeration types in Ada and HP Pascal.

Table 11-5. Ada versus HP Pascal Enumeration Correspondence

Ada	HP Pascal	Bit Length
<= 128 elements	<= 256 elements	8
<= 32768 elements	<= 65536 elements	16
> 32768 elements	> 65536 elements	32

If the Ada enumeration type has 129 through 256 elements or 32769 through 65536 elements, there are additional requirement to passing or returning values of such an Ada type. A size specification on a representation clause for the Ada enumeration type should be used to specify the minimum size for the enumeration type (see section F 4.1 for details.) When such a size is used and none of the internal codes are negative integers, the internal representation of the Ada type will be unsigned and will conform with the HP Pascal representation.

If such a size specification representation clause is not used, it is still possible to pass a simple variable or expression of such a type to HP Pascal, by value or reference, or to return one from HP Pascal. Although the Ada enumeration object is stored in a larger container than HP Pascal expects, the valid values are actually all stored within the part of the container that HP Pascal will access.

However, unless a size specification representation clause is used, there will be difficulty passing arrays of Ada enumeration values of such types or passing records containing fields of such types. HP Pascal will not properly access the correct elements of such arrays or fields of such records because it will assume the enumeration values to be smaller than they actually are and therefore will compute their location incorrectly.

If a representation specification is applied to the Ada enumeration type to alter the internal value of any enumeration elements, care must be taken that the values are within the HP Pascal enumeration type to which the Ada enumeration value is being passed.

Ada supports the return of a function result that is an enumeration type from an external interfaced function subprogram written in HP Pascal.

F 11.5.1.3 Boolean Types and HP Pascal Subprograms

See section "F 11.1.1.3 Boolean Types" for details.

F 11.5.1.4 Character Types and HP Pascal Subprograms

See section "F 11.1.1.4 Character Types" for details.

Values of the Ada predefined character type might be treated as the type CHAR in HP Pascal external interfaced subprograms.

F 11.5.1.5 Real Types and HP Pascal Subprograms

The following subsections discuss passing Ada real types to interfaced HP Pascal subprograms.

Fixed Point Types

Ada fixed point types are not supported as parameters or as results of external subprograms. Ada fixed point types *cannot* be returned as function results from interfaced subprograms written in HP Pascal.

Floating Point Types

See section "F 11.1.1.5 Real Types" for details.

Ada `FLOAT` values correspond to HP Pascal **real** values. Ada `LONG_FLOAT` values correspond to HP Pascal **longreal** values.

F 11.5.2 Access Types and HP Pascal Subprograms

See section "F 11.1.2 Access Types" for details.

Ada access values can be treated as pointer values in HP Pascal. The Ada heap allocation and the HP Pascal heap allocation are completely separate. There must be no explicit deallocation of an access or pointer object in one language of an object allocated in the other language.

F 11.5.3 Array Types and HP Pascal Subprograms

See section "F 11.1.3 Array Types" for details.

Arrays with components with the same representation have the same representation in Ada and HP Pascal.

Arrays *cannot* be passed by value from Ada to HP Pascal. An Ada array can only be passed to a VAR parameter in an HP Pascal subprogram.

Array types *cannot* be returned as function results from external interfaced subprograms written in HP Pascal.

Pascal conformant array parameters passed by reference (VAR) can be passed from Ada to Pascal. To pass such parameters, additional implicit parameters expected by Pascal must be added in the Ada declaration of the Pascal procedure or function. For each dimension, except the last dimension, these additional parameters are the bounds of the array followed by the size of the array elements in bytes and they must immediately follow the conformant array parameter or parameters. The bounds and element size parameters must be declared as in parameters of an integer or enumeration type.

When more than one conformant array parameter is declared in a comma-separated formal parameter list in a Pascal procedure or function heading, all the actual parameters passed to the formals in that list must have the same number of dimensions and the same lower and upper index bound in each dimension. Therefore, only one set of implicit bound parameters is needed for the two formal parameters A and B in the following example (no element size is passed to A and B because they have only one dimension). The actual parameters passed to the formal A and B parameters must have the same index bounds.

Note that two sets of implicit bound parameters and one element size parameter are needed for the two-dimensional conformant array formal parameter C (one set of bounds for each dimension and an element size for the second dimension). If there are additional two-dimensional conformant array parameters, and they are all declared in the same comma-separated parameter list in Pascal (with the C parameter), only the two sets of implicit bound parameters and the one element size parameter is required for the entire list. For example, if the Pascal function heading is

```
function VectorThing
   (VAR a,b: ARRAY [i..j: INTEGER] of INTEGER;
    VAR c  : ARRAY [p..q: INTEGER] of ARRAY [r..s: INTEGER]
      of INTEGER) : INTEGER;
```

The Ada declaration to call such a Pascal function follows:

```
with SYSTEM;
procedure PASCAL_CONFORM_FUNC is

  type VECTOR is array (INTEGER range -492..+500)
                        of INTEGER range -100..+100;

  type VECTOR2 is array (INTEGER range 1..10) of VECTOR;

  function VECTORTHING (A, B: VECTOR;
                           I, J:   INTEGER
                        C: VECTOR2;
                           P, Q:   INTEGER;
                           R, S:   INTEGER;
                           ELSIZE: INTEGER) return INTEGER;

  pragma INTERFACE (PASCAL, VECTORTHING);

  -- An appropriate pragma INTERFACE_NAME is needed here
  -- to properly access the Pascal function because the
  -- actual function name depends on the Pascal module
  -- in which the function appears.

  VECTOR_SIZE  : constant INTEGER
                 := VECTOR'SIZE  / SYSTEM.STORAGE_UNIT;

  W, V: VECTOR;
  X   : VECTOR2;
  I   : INTEGER;

begin -- PASCAL_CONFORM_FUNC
  I := VECTORTHING (W, V,
                       VECTOR'FIRST, VECTOR'LAST,
                    X,
                       VECTOR2'FIRST, VECTOR2'LAST,
                       VECTOR'FIRST, VECTOR'LAST,
                       VECTOR_SIZE);
end PASCAL_CONFORM_FUNC;
```

F 11.5.4 String Types and HP Pascal Subprograms

See section F 11.1.3 for details.

Passing variable length strings between Ada and HP Pascal is supported with some restrictions. Strings *cannot* be passed by value from Ada to HP Pascal. An Ada string can only be passed to a VAR parameter in an HP Pascal subprogram.

String types *cannot* be returned as function results from external HP Pascal subprograms.

Although there is a difference in the implementation of the type STRING in the two languages, with suitable declarations you can create compatible types to allow the passing of both Ada strings and HP Pascal strings. An Ada string corresponds essentially to a packed array of characters in Pascal. However, the Ada string type must be one character longer than the corresponding string type in the HP Pascal procedure or function. HP Pascal adds such an implicit extra byte to its own packed arrays of characters and expects to be able to utilize this extra byte during some string operations. The following example illustrates the declaration of compatible types for passing an Ada string between an Ada program and an HP Pascal subprogram.

HP Pascal subprogram:

```
(* passing an Ada STRING type to an HP Pascal routine *)
module p;
export
   type string80 = packed array [1..80] of char;
   procedure ex1 ( var s : string80; len : integer );
implement
   procedure  ex1;
   begin
      ... (* update/use the Ada string as a PAC *)
   end;
end.
```

Ada program:

```
-- Ada calling HP Pascal procedure with Ada STRING
procedure AP_1 is

   -- Define Ada string corresponding to
   -- HP Pascal packed array of char
   subtype STRING80 is STRING ( 1..81 ); -- 80+1 for HP Pascal

   -- Ada definition of HP Pascal procedure to be called,
   -- with an  Ada STRING parameter, passed by reference.
   procedure  EX1 (S     : in out STRING80;
                   LEN   :        INTEGER );
   pragma INTERFACE (PASCAL, EX1);
   pragma INTERFACE_NAME (EX1, "ex1");

   S      : STRING80;

begin -- AP_1
   S(1..26) := "Ada to HP Pascal Interface";
   EX1 (S, 26);      -- Call the HP Pascal subprogram
end AP_1;
```

An HP Pascal STRING type corresponds to a record in Ada that contains two fields: a 32-bit integer field containing the string length and an Ada STRING field containing the string value. The following example illustrates the declaration of compatible types for passing an HP Pascal string between an Ada program and a Pascal subprogram.

Pascal subprogram:

```
(* passing an HP Pascal STRING type from Ada to *)
(* an HP Pascal routine                          *)
module p;
export
   type string80 = string[80];
   procedure ex2 ( var s : string80 );
implement

   procedure  ex2;
   var
      str : string80 ;
   begin
     ... --update/use the HP Pascal string
   end;
end.
```

Ada program:

```
-- Ada calling HP Pascal procedure using a HP Pascal string[80]
procedure AP_2 is

   -- Define an Ada record that will correspond exactly
   -- with the HP Pascal type: string[80]

   type PASCAL_STRING80 is
      record
         LEN : INTEGER;
         S   : STRING ( 1..81 ); -- 80+1 for HP Pascal
      end record;

   -- Here we use a record representation clause to
   -- force the compiler to layout the record in
   -- the correct manner for HP Pascal

   for PASCAL_STRING80 use
      record
         LEN at 0 range 0 .. 31;
         S   at 1 range 0 .. 81*8; -- 80+1 for HP Pascal
      end record;

   -- The Ada definition of the HP Pascal procedure to be
   -- called, with an HP Pascal STRING parameter, passed
   -- by reference.

   procedure  EX2 (S : in out PASCAL_STRING80);
   pragma INTERFACE (PASCAL, EX2);
   pragma INTERFACE_NAME (EX2, "ex2");

   PS      : PASCAL_STRING80;

begin -- AP_2

   -- assign value field
   PS.S(1..26) := "Ada to HP Pascal Interface";
```

```
        PS.LEN := 26;   -- set string length field
        EX2 ( PS );     -- call the HP Pascal subprogram

    end AP_2;
```

F 11.5.5 Record Types and HP Pascal Subprograms

See section "F 11.1.4 Record Types" for details.

Records *cannot* be passed by value from Ada to HP Pascal. An Ada record can only be passed to a VAR parameter in an HP Pascal subprogram.

Record types *cannot* be returned as function results from external HP Pascal subprograms.

F 11.6 Summary

Table 11-6 shows how various Ada types are passed to subprograms.

Table 11-6.
Modes for Passing Parameters to Interfaced Subprograms

Ada Type	Mode	Passed By
ACCESS, SCALAR -INTEGER -ENUMERATION -BOOLEAN -CHARACTER -REAL	in	value
ARRAY, RECORD	in	reference
all types except TASK and FIXED POINT	in out	reference
all types except TASK and FIXED POINT	out	reference
TASK FIXED POINT	N/A	not passed

Table 11-7 summarizes general information presented in section "F 11.1 General Considerations in Passing Ada Types".

Table 11-7.
Types Returned as External Function Subprogram Results

Ada Type	Precision Architecture RISC Assembly Language	HP C	HP FORTRAN	HP Pascal
INTEGER	allowed	allowed	allowed	allowed
ENUMERATION	allowed	allowed	not allowed (1)	allowed
CHARACTER	allowed	allowed	not allowed	allowed
BOOLEAN	allowed	allowed	allowed	allowed
FLOAT	allowed	allowed (2)	allowed	allowed
FIXED POINT	not allowed	not allowed	not allowed	not allowed
ACCESS	allowed	allowed	not allowed (1)	allowed
ARRAY	not allowed	not allowed	not allowed	not allowed
STRING	not allowed	not allowed	not allowed (3)	not allowed
RECORD	not allowed	not allowed	not allowed	not allowed
TASK	not allowed	not allowed	not allowed	not allowed

Notes for Table 11-7:

(1) Pass as an integer equivalent.

(2) Some restrictions apply to Ada FLOAT types (in passing to HP C subprograms).

(3) Accessible if function called as a procedure with "extra" parameters.

Table 11-8 summarizes information presented in sections F 11.2 through F 11.5.

Table 11-8. Parameter Passing in the Ada Implementation

Ada Type	Precision Architecture RISC Assembly Language	HP C	HP FORTRAN	HP Pascal
INTEGER	allowed	allowed	allowed	allowed
ENUMERATION	allowed	allowed	not allowed (1)	allowed
CHARACTER	allowed	allowed	not allowed (2)	allowed
BOOLEAN	allowed	allowed	not allowed (1)	allowed
FLOAT	allowed	allowed	allowed	allowed
FIXED POINT	not allowed	not allowed	not allowed	not allowed
ACCESS	allowed	allowed	not allowed	allowed
ARRAY (3)	allowed	allowed	allowed (4)	allowed (8)
STRING	allowed	allowed (5)	allowed (6)	not allowed (7)
RECORD	allowed	allowed	allowed	allowed
TASK	not allowed	not allowed	not allowed	not allowed

Notes for Table 11-8:

(1) Can be passed as an equivalent integer value.

(2) Must be passed as a STRING.

(3) Using only arrays of compatible component types.

(4) See warning on layout of elements.

(5) Special handling of null terminator character is required.

(6) Requires that the length also be passed.

(7) Ada strings can be passed to a Pascal PAC (Packed Array of Characters)

(8) Conformant arrays require that "extra" parameters be passed.

F 11.7 Potential Problems Using Interfaced Subprograms

F 11.7.1. Signals and Interfaced Subprograms

The Ada run-time on the HP 9000 Series 600, 700, and 800 computers uses signals in a manner that generally does not interfere with interfaced subprograms. However, some HP-UX routines are interruptible by signals. These routines, if called from within interfaced external subprograms, may create problems. You need to be aware of these potential problems when writing external interfaced subprograms in other languages that will be called from within an Ada main subprogram. See `sigvector(2)` in the *HP-UX Reference* for a complete explanation of interpretability of operating system routines.

The following should be taken into consideration:

- `SIGALRM` is sent when a `delay` statement reaches the end of the specified interval.
- One of `SIGALRM`, `SIGVTALRM` (the default), or `SIGPROF` is sent periodically when time-slicing is enabled in a tasking program.
- Interruptible HP-UX routines (see `sigvector(2)`) may need to be protected from interruption by the signals used by the Ada run-time system. The `SYSTEM_ENVIRONMENT` routines `SUSPEND_ADA_TASKING` and `RESUME_ADA_TASKING` can be used to implement this protection. As an alternative, the knowledgeable user can use the `sigsetmask(2)` or `sigblock(2)` mechanism to implement the same protection.
- If a signal is received while it is blocked, one instance of the signal is guaranteed to remain pending and will be honored when the signal is unblocked. Any additional instances of the signal will be lost.
- Any signals blocked in interfaced code should be unblocked before leaving the interfaced code.

The alarm signals sent by `delay` statements and sent to implement time-slicing (noted above) are the most likely signals to cause problems with interfaced subprograms. These signals are asynchronous; that is, they can occur at any time and are not caused by the code that is executing at the time they occur. In addition, `SIGALRM` and `SIGPROF` (but not `SIGVTALRM`) can interrupt HP-UX routines that are sensitive to being interrupted by signals.

Problems can arise if an interfaced subprogram initiates a "slow" operating system function that can be interrupted by a signal (for example, a `read(2)` call on a terminal device or a `wait(2)` call that waits for a child process to complete). Problems can also arise if an interfaced subprogram can be called by more than one task and is not reentrant. If an Ada reserved signal occurs during such an operation or non-reentrant region, the program may function erroneously.

For example, an Ada program that uses `delay` statements and tasking constructs causes the generation of `SIGALRM` and optionally either `SIGVTALRM` or `SIGPROF`. If an intefaced subprogram needs to perform a potentially interruptible system call or if the interfaced subprogram can be called from more than one task and is not reentrant, you can protect the interfaced subprogram by blocking the potentially interrupting time signals around the system call or non-reentrant region. If one of these timer signals does occur while blocked, signifying either the end of a `delay` period or the need to reschedule due to time-slice expiration, that signal is not lost; it is effectively deferred until it is later unblocked.

Assuming a tasking program, which contains one or more **delay** statements, with time-slicing enabled using the default time-slicing signal (SIGVTALRM), the following example shows a protected **read(2)** call in the C language:

```
#include <signal.h>
void interface_rout();
{
   long mask;

...

   /* Add SIGALRM and SIGVTALRM to the list of currently
      blocked signals (see sigblock(2)).     */

  mask = sigblock (sigmask (SIGALRM) | sigmask (SIGVTALRM));

  ...    read (...);    /* or non-reentrant region */

  /* restore old mask so Ada run-time can function */
  sigsetmask (mask);

...
}
```

If any Ada reserved signal other than SIGALRM or the alarm signal (if any) being used for time-slicing is to be similarly blocked, SIGALRM and the alarm signal used for time-slicing must already be blocked or must be blocked at the same time as the other signal or signals.

Any Ada reserved signal blocked in interfaced code should be unblocked before leaving that code, or as soon as possible thereafter, to avoid unnecessarily stalling the Ada run-time executive. Failure to follow these guidelines will cause improper delay or tasking operation.

An alternative and preferred method of protecting interfaced code from signals is described in the *Ada User's Guide* in the section on "Execution-Time Topics." The two procedures SUSPEND_ADA_TASKING and RESUME_ADA_TASKING from the package SYSTEM_ENVIRONMENT supplied by Hewlett-Packard can be used within an Ada program to surround a critical section of Ada code or a call to external interfaced subprogram code with a critical section.

F 11.7.2 Files Opened by Ada and Interfaced Subprograms

An interfaced subprogram should *not* attempt to perform I/O operations on files opened by Ada. Your program should not use HP-UX I/O utilities intermixed with Ada I/O routines on the same file. If it is necessary to perform I/O operations in interfaced subprograms using the HP-UX utilities, open and close those files with HP-UX utilities.

12

F 12. Interrupt Entries

This chapter describes interrupt entry processing.

F 12.1 Introduction

The Ada compiler supports a limited form of interrupt entries as defined by the *Ada RM*, section 13.5.1. In addition, the compiler provides the following features:

- Interrupt entries are associated with HP-UX signals, but are not directly invoked by an HP-UX signal. Instead, the interrupt entry is called from an Ada signal handling procedure. An Ada signal handling procedure can be associated with one or more HP-UX signals. If an Ada signal handler wants to call an interrupt entry, it can only call the interrupt entry that is associated with the same HP-UX signal that caused the Ada signal handler itself to be invoked.
- Interrupt entries associated with HP-UX signals can have parameters.
- If the interrupt entry call cannot be processed immediately by the server task, the interrupt entry parameters are buffered so that the interrupt is not lost and the entry is processed as soon as conditions permit.
- All signals except the ones reserved by the Ada runtime and the HP-UX system can be handled with up to seven different priorities. The interrupt entry mechanism will not prohibit the use of signals reserved by the Ada runtime or by HP-UX. but using such signals for interrupt entries will cause unpredictable program behavior.

12

F 12.2 Immediate Processing

If an Ada handler has been associated with an HP-UX signal, when that signal occurs, an internal signal handler installed by the runtime system is entered. That internal handler then calls the user-defined Ada handler. One parameter of type SYSTEM.ADDRESS is passed to the user handler; the parameter is the "signal number" that caused it to be invoked (a function is provided to convert the integer representation of a "signal number" into an object of type SYSTEM.ADDRESS). The Ada handler can make an entry call to a task entry associated with the particular signal and/or it can update global state information (for example, variables) that is meaningful to the program. It should then return, giving control to the internal handler in the runtime system.

If the Ada handler makes an entry call to an entry previously declared with a representation clause as an interrupt entry, the rendezvous does not occur immediately. The kernel saves the parameters passed by the signal handler in a buffer taken from a pool of free buffers and links the buffer to the entry queue for the interrupt. The actual rendezvous will take place in deferred processing (see section "F 12.3 Deferred Processing"). The pool of free buffers is allocated once at the program startup by calling INIT_INTERRUPT_MANAGER with the number of buffers specified (see section "F 12.5 Initializing the Interrupt Entry Mechanism").

Note The Ada handler must not call any Ada Runtime System routines and must be compiled with checks off. See sections "F 12.6 Associating an Ada Handler with an HP-UX Signal" and "F 12.6.1 Determining If Your Ada Handler Makes Ada Runtime Calls" for details.

F 12.3 Deferred Processing

The deferred processing step is the execution of the accept statement for the interrupt entry. It is performed with signals enabled and with an Ada task priority specified by the user (but higher than any software priority as required by the *Ada RM*, section 13.5.1.2). The accept statement has access to the IN parameters provided by the Ada handler when the Ada handler made the entry call. There are no limitations on the code of the accept statement; run-time calls are allowed.

The connection between the immediate and deferred processing is made by the Ada runtime. At the end of the immediate processing step, when the Ada handler returns control to the internal handler in the Ada runtime, Ada runtime checks to see if any immediate processing steps remain active (that is, an Ada signal handler has been called in response to a signal but has not yet returned). If any immediate processing steps remain active, the Ada runtime simply returns control to the interrupted context, which will be one of the currently active Ada signal handlers.

If no immediate processing steps remain active (that is, the Ada signal handler that is currently returning control to the Ada runtime is the only currently active Ada signal handler), the Ada runtime identifies all of the tasks and entries that immediate processing steps have requested be called. There may be more than one interrupt entry call pending because multiple different signals may have been received, causing multiple Ada signal handlers to be simultaneously active. Only when the last active Ada signal handler returns control to the Ada runtime will the pending tasks or entries be considered as callable. The Ada runtime will determine for each pending interrupt entry call whether the accept statement can be executed immediately. If so, the current task is preempted unless it is of equal or higher priority than any of the pending interrupt entry calls (for example, the current task is itself executing an accept statement for a higher priority interrupt). Pending interrupt entry calls for which the accept can be executed, but which are of a lower priority than the currently running task, will be made as their priority permits (note that calls to interrupt entries with identical priorities may occur in an arbitrary order).

If the accept statement cannot be executed immediately, the rendezvous will take place according to normal Ada semantics when the server task executes an accept or select statement for the given entry.

There are no restrictions on the number of interrupt entries one task can use, nor on the number of tasks that can use interrupt entries. The only restriction is that only one entry may be associated with a given HP-UX signal and that signals reserved by the Ada runtime may not be associated with an interrupt entry.

The buffering of the interrupt entry call from the Ada handler to the interrupt entry attempts to ensure that no signal will be lost. It is important that the average execution time of the interrupt entry be smaller than the signal rate for the associated signal, otherwise the pool of buffers to hold interrupt entry parameters will be quickly exhausted. Each buffer is released immediately before execution of the accept body for the interrupt entry after the parameters have been copied to the stack of the acceptor task. It is also important that the execution time of the Ada signal handlers be minimized as the deferred processing step is not performed when any Ada signal handler remains active.

F 12.4 Handling an Interrupt Entirely in the Immediate Processing Step

Calling an interrupt entry in response to a signal is optional. Interrupts can be handled in a sequential program that has no tasks to call or in a tasking program without calling an interrupt entry if the Ada handler performs all the required processing. This can improve performance because the overhead of task switching is avoided. However, because the Ada handler cannot make Ada run-time calls and must be compiled with checks off (using the -R option), the amount of processing that an Ada handler can do is limited. In addition, if the Ada handler does all the processing, the Ada program must generally poll global state information to determine that the signal has been received.

F 12.5 Initializing the Interrupt Entry Mechanism

The compiler provides the package INTERRUPT_MANAGER to support interrupt entries. This package is in the predefined library.

To use interrupt entries, you must initialize the interrupt manager by calling this procedure:

```
procedure INIT_INTERRUPT_MANAGER
            (NUMBER_OF_BUFFERS     : in BUFFER_NUMBER;
             MAX_PARAM_AREA_SIZE   : in BYTE_SIZE;
             INTERRUPT_STACK_SIZE  : in BYTE_SIZE := 2048);
```

This procedure allocates the given number of buffers to hold parameters of interrupt entries that cannot be processed immediately and allocates a signal stack of the given size. The size of each buffer is the maximum parameter area size specified to the call, plus a fixed overhead of 28 bytes used by the Ada runtime. If the given signal stack size is zero, all signals are handled on the current stack; therefore, all stacks must have sufficient buffer space. Using an interrupt stack allows better usage of available memory.

The MAX_PARAM_AREA_SIZE parameter must be the size, in storage units, of the largest parameter block required by an interrupt entry call. A parameter block is an area of memory in which the generated code for a task entry call temporarily stores the actual parameters of the task entry call. The address of the parameter block is passed to the Ada run-time routine ENTRY_CALL which makes the parameters available to the called task entry when the rendezvous actually occurs. In the case of an interrupt entry, the Ada runtime copies the parameter block into one of the buffers allocated by INIT_INTERRUPT_MANAGER until the deferred processing step is reached.

The parameter block sizes for the task entries to be called by Ada signal handling procedures can be obtained by compiling the specifications of the task for such entries with the -H option. An informational message is produced indicating the parameter block size for each task entry specification that has an address clause, indicating it is a candidate for calling from an Ada signal handling procedure.

12

The procedure INIT_INTERRUPT_MANAGER must be called at program startup before any call is made to an interrupt entry from an Ada signal handler. Entry calls will be lost if the number of buffers is insufficient. The required number of buffers depends on the frequency of signals. A zero number of buffers can be used when the signal handler only buffers information and never calls an interrupt entry.

This procedure raises STORAGE_ERROR if there is not enough memory to allocate the required buffers and the interrupt stack.

F 12.6 Associating an Ada Handler with an HP-UX Signal

You can install a signal handler by calling the following procedure:

```
procedure INSTALL_HANDLER
      (HANDLER_ADDRESS  : in SYSTEM.ADDRESS;
       SIG              : in SYSTEM.ADDRESS;
       PRIORITY         : in INTERRUPT_PRIORITY
                          := INTERRUPT_PRIORITY'FIRST;
       ORIGINAL_HANDLER : in ACTION := REPLACED);
```


This procedure installs an Ada routine, specified via HANDLER_ADDRESS, as the Ada handler for the specified HP-UX signal (SIG) after saving the address of the original handler. If the Ada handler calls an interrupt task entry, the signal number passed to this procedure as SIG must be the same as the one specified in the interrupt entry address clause (see "F 12.10 Address Clauses for Entries") for that task entry. The PRIORITY parameter specifies the priority of the entry call to be made by the handler (all accept statements will run with this priority unless they are themselves within an accept statement executed at higher priority.) The ORIGINAL_HANDLER parameter controls whether the current signal handler, the one the Ada handler is replacing, is to be called before (FIRST) or after (LAST) the new Ada handler or not called at all (REPLACED).

The INSTALL_HANDLER procedure must be called from a scope that encloses the declaration of the Ada procedure that is being installed as the Ada handler. A convenient technique is to declare the Ada handler procedure immediately within a library level package and place the call to INSTALL_HANDLER in the package body block. INSTALL_HANDLER will not detect any error if this restriction is violated; however, unexpected program behavior or program failure may occur when an incorrectly installed Ada handler is invoked.

The Ada handler must be a procedure with one parameter of type SYSTEM.ADDRESS and without inner units. The procedure can only reference local or global objects, excluding objects of an enclosing frame, and must be compiled with checks off (using the -R option). If an entry call is made in an Ada handler, the task the entry belongs to must be a global object.

The Ada handler must not call any Ada Runtime System routines (either explicitly or implicitly) other than simple entry calls to interrupt entries because some of the Ada run-time routines update critical run-time data

structures and must not be reentered during such updates. Specifically, neither timed nor conditional entry calls may be made.

The Ada handler must not call HP-UX routines or other non-Ada code, either via pragma INTERFACE or via a binding.

If the Ada handler calls another Ada procedure or function, that procedure or function must follow these same constraints.

For certain complex data structures, the compiler produces Type Support Subprograms (TSS). These subprograms perform actions such as initializing record fields, comparing records, changing record representation, and so on. The compiler then adds implicit calls to these routines when needed. Some calls to TSS routines are not safe in Ada handlers. If the TSS calls Ada Runtime System routines, that TSS should not be called from an Ada handler. To remove the call, you have to simplify the code so that the action that needs the call is no longer present. The -H option, if used when compiling the Ada handler, causes an informational message to be produced if the generated code contains a call to one or more TSS routines. Additionally, the massages specify the type that the TSS supports.

In general, the code in an Ada handler should be kept extremely simple. It is recommended to only set a global flag or make an entry call to an interrupt entry to actually do the required processing.

The HP-UX signal currently being handled is masked for the duration of the Ada handler.

The address of the Ada procedure to use as the Ada handler can be obtained by the 'ADDRESS attribute, which is only valid after elaboration of the procedure body.

The procedure INSTALL_HANDLER raises STORAGE_ERROR if MAX_HANDLERS handlers have already been defined.

F 12.6.1 Determining If Your Ada Handler Makes Ada Runtime Calls

If you are not sure your Ada handler makes specific Ada Runtime System routine calls, you can compile the Ada source file containing the Ada handler with the -H option. The -H option causes the compiler to produce a warning message each time it generates code to call an Ada Runtime System routine. If you generate a complete compiler listing with the -B or -L options, the warnings appear in the compiler listing at the appropriate source lines. If you do not generate a complete compiler listing (with -B or -L options), only the source lines that apply to the warning appear in a listing with the warning. Check the names of the Ada Runtime System routines that appear in the warning messages; if calls to any of the following Ada Runtime System routines appear, your Ada handler is "unsafe".

```
ABORT_STMT                      END_ACTIVATION          FREE_TEMP_GH
ACCEPT_STMT                     ENUM1_PRED              FREE_VAR_SS_ELT
ACTIVATE_COLLECTION             ENUM1_SUCC              INIT_COLLECTION
ALLOC_FIX_SS_ELT                ENUM1_VAL_TO_POS        INIT_FIX_SS_ELT
ALLOC_GO                        ENUM2_PRED              INIT_HANDLER
ALLOC_LO                        ENUM2_SUCC              INIT_MASTER
ALLOC_SMALL_FIX_ELT             ENUM2_VAL_TO_POS        INIT_SMALL_FIX_ELT
ALLOC_TEMP                      ENUM4_PRED              INIT_VAR_SS_ELT
ALLOC_TEMP_GH                   ENUM4_SUCC              INTEGER_IMAGE
ALLOC_VAR_SS_ELT                ENUM4_VAL_TO_POS        INTEGER_VALUE
CALLABLE                        ENUM_POS                INTEGER_WIDTH
COMPLETE_MASTER                 ENUM_WIDTH              NULL_BODY_ACCEPT_STMT
COMPLETE_TASK                   ENV_TASK_MASTER         SELECT_WITH_TERMINATE
COND_CALL                       FIXED_FORE              SIMPLE_SELECT
COND_SELECT                     FIXED_LARGE             SIMPLE_TIMED_SELECT
COUNT                           FIXED_MANTISSA          TERMINATED
CREATE_TASK                     FREE_FIX_SS_ELT         TERMINATION_COMPLETE
CURRENT_OBJECT_OF_TASK_TYPE     FREE_LIST               TIMED_CALL
DELAY_STMT                      FREE_SMALL_FIX_ELT      TIMED_SELECT
DESTROY_COLLECTION              FREE_TEMP
```

A call to ENTRY_CALL is safe as long as it is calling the task entry declared as an interrupt entry for the HP-UX signal that caused the Ada signal handler to be invoked.

To help you understand what Ada language construct might cause such a call to be made, a description of each of the above Ada run-time routines is listed in section "F 12.13 Ada Runtime Routine Descriptions".

F 12.7 Disassociating an Ada Handler from an HP-UX Signal

The Ada handler for a given HP-UX signal can be removed and the original HP-UX signal handler (or signal behavior) restored with this procedure

```
procedure REMOVE_HANDLER (SIG : in SYSTEM.ADDRESS);
```

This procedure only needs to be called when it is no longer necessary to have a handler for a particular signal. All Ada handlers are automatically disassociated from their HP-UX signals when the main program terminates.

The procedure REMOVE_HANDLER raises PROGRAM_ERROR if no handler has been installed for the given signal.

F 12.8 Determining How Many Handlers are Installed

Use the following procedure to determine how many handlers have already been installed:

```
function HANDLER_COUNT return HANDLER_NUMBER;
```

F 12.9 When Ada Signal Handlers Will Not Be Called

When the procedure SYSTEM_ENVIRONMENT.SUSPEND_ADA_TASKING is called, the HP-UX signals for which Ada signal handlers have been installed, will be masked. That is, the Ada signal handlers will not be called if one of the signals should occur. At most one instance of any given signal will be remembered while the signals are masked. When the procedure SYSTEM_ENVIRONMENT.RESUME_ADA_TASKING is called, the signals for which Ada signal handlers have been installed will be unmasked. Any signal that occurred while the masking was in effect will then be delivered to the Ada program and will invoke the associated Ada signal handlers (at most one instance of any such signal will have been remembered while the signals were masked).

12

Caution The Ada program, as well as interface code called by the Ada program, should not unmask any of the HP-UX signals for which Ada signal handlers are installed while Ada tasking has been suspended. Doing so will cause unpredictable and possibly erroneous program behavior.

F 12.10 Address Clauses for Entries

According to section 13.5.1 of the *Ada RM*, an address clause for an interrupt entry has the following form:

```
task INTERRUPT_HANDLER is
   entry INTERRUPT(...);
   for INTERRUPT use at ...;
end INTERRUPT_HANDLER;
```

An interrupt entry may have zero or more parameters of mode `IN`. Parameters of mode `IN OUT` or `OUT` are not permitted; see section 13.5.1(1) in the *Ada RM* for details. The expression in the address clause must be of type `SYSTEM.ADDRESS` and is interpreted as a signal number by the runtime system. A function `SIGNAL` is provided by the `INTERRUPT_MANAGER` package to convert an integer to a `SYSTEM.ADDRESS`. Note that a `with` statement for the package `SYSTEM` must be specified for the context in which such an address clause appears.

F 12.11 Example of Interrupt Entries

The following is a program that uses interrupt entries. The program could be written using only Ada tasking and not using HP-UX signals or interrupt entries; however, this version provides a sample of declaring and using the interrupt entry mechanism. A machine readable copy of this program is provided in `$ADA_PATH/samples/intent/intentex1.ada` (although the text and line breaks are different).


```
-- ----------------------------------------------------------
-- This program simulates an elevator which takes passengers from
-- floor to floor.
--
-- There are three tasks :  PASSENGER, ELEVATOR and MANAGER.
--
-- The MANAGER task manages the elevator by determining the floor
-- number and the direction to go.  The PASSENGER task generates
-- passengers and sends a signal to the ELEVATOR task.  The
-- ELEVATOR loads and unloads passengers and sends a signal back
-- to the PASSENGER task after loading new arrivals or placing
-- them in a queue (the elevator has a maximum capacity, the
-- queue holds any passengers who have to wait).
-- ----------------------------------------------------------

with SYSTEM, INTERRUPT_MANAGER, TEXT_IO;
procedure MAIN is
   -- user signals
  SIGUSR1: constant SYSTEM.ADDRESS:= INTERRUPT_MANAGER.SIGNAL(16);
  SIGUSR2: constant SYSTEM.ADDRESS:= INTERRUPT_MANAGER.SIGNAL(17);

   subtype PROCESS_ID is INTEGER;

   -- HP-UX calls
   procedure KILL (PID : PROCESS_ID; SIG : SYSTEM.ADDRESS);
   pragma INTERFACE (C, KILL);

   function GETPID return PROCESS_ID;
   pragma INTERFACE (C, GETPID);
```

```
function RAND return INTEGER;
pragma INTERFACE (C, RAND);
--

MY_PID : PROCESS_ID := GETPID;

-- elevator capacity, directions, floor numbers,
-- arriving passengers

MAX_CAPACITY : constant := 16;

type DIRECTION is (NONE, UP, DOWN);
subtype MOTION is DIRECTION range UP .. DOWN;
ARR_DIR, CUR_DIP : MOTION;

subtype FLOOR is INTEGER range 1 .. 5;
ARR_FLR, CUR_FLR : FLOOR;

ARR_PASS : INTEGER;

-- random floor and direction selectors

function RAND_FLR return FLOOR is
begin
   return (RAND rem (FLOOR'LAST - FLOOR'FIRST + 1)) + 1;
end RAND_FLR;

function RAND_MOTION return MOTION is
   MOTIONS : constant INTEGER := MOTION'POS (MOTION'LAST)
     -MOTION'POS (MOTION'FIRST);
begin
   return MOTION'VAL (MOTION'POS(MOTION'FIRST)
                + (RAND rem MOTIONS));
end RAND_MOTION;

-- show passenger/floor information
```

```
procedure SHOW_PASS (WHERE: in FLOOR;  PASS:  in INTEGER;
              DOING: in STRING; WHICH: in DIRECTION := NONE) is

   FNUM : constant STRING := FLOOR'IMAGE (WHERE);
   PNUM : constant STRING := INTEGER'IMAGE (PASS);

begin
   TEXT_IO.PUT ("On floor " & FNUM ((FNUM'FIRST+1)..FNUM'LAST)
              & " there ");
   case PASS is
      when 0 =>      TEXT_IO.PUT ("are no passengers");
      when 1 =>      TEXT_IO.PUT ("is 1 passenger");
      when others => TEXT_IO.PUT
                     ("are " & PNUM ((PNUM'FIRST+1)..PNUM'LAST)
                       & " passengers");
   end case;
   TEXT_IO.PUT (DOING);
   if WHICH /= NONE then
      TEXT_IO.PUT (DIRECTION'IMAGE(WHICH) & ".");
   end if;
   TEXT_IO.NEW_LINE;
end SHOW_PASS;

task OUTPUT is
   entry LOCK;    -- This task implements a lock/unlock
                  -- mechanism for
   entry UNLOCK;  -- output, so that messages do not
                  -- get intermixed.
end OUTPUT;

task PASSENGER is
   -- This task generates passengers at various floor levels
   -- and sends SIGUSR1 to the ELEVATOR task.  The elevator
   -- task sends SIGUSR2 back to acknowledge that passengers
   -- have been served.
   entry START;
   entry CONTINUE;
   for CONTINUE use at SIGUSR2;
```

12

```
end PASSENGER;

task ELEVATOR is
   -- This task loads and unloads passengers.
   entry START;
   entry LOAD_ARRIVING_PASS;
   entry UNLOAD;
   entry LOAD_WAITING_PASS;
   for LOAD_ARRIVING_PASS use at SIGUSR1;
end ELEVATOR;

task MANAGER is
   -- This task determines the floor number and direction of
   -- the elevator.  It calls ELEVATOR task to load and unload
   -- passengers on each floor.
   entry START;
end MANAGER;

task body OUTPUT is
begin
    loop
       accept LOCK;
       accept UNLOCK;
    end loop;
end OUTPUT;

task body PASSENGER is
begin
   accept START;
   loop
      ARR_FLR := RAND_FLR;    -- Determine the floor number.
      if ARR_FLR = FLOOR'FIRST then
         ARR_DIR := UP;       -- The bottom floor only goes up.
      elsif ARR_FLR = FLOOR'LAST then
         ARR_DIR := DOWN;     -- The top floor only goes down.
      else
         ARR_DIR := RAND_MOTION;
                          -- Any other floor can go either way.
```

```
         end if;
         ARR_PASS := RAND rem MAX_CAPACITY;

         if ARR_PASS > 0 then
            OUTPUT.LOCK;
            SHOW_PASS (ARR_FLR, ARR_PASS,
                       " arriving to go ", ARR_DIR);
            OUTPUT.UNLOCK;
         end if;

         KILL (MY_PID, SIGUSR1);
                                   -- Send SIGUSR1 to the ELEVATOR task.

         accept CONTINUE;
                       -- Passengers loaded or waiting in the queue.
      end loop;
   end PASSENGER;

   task body ELEVATOR is
      -- This task loads and unloads passengers.
      -- It loads the arriving passengers when the elevator
      -- arrives on the right floor when going in the right
      -- direction.  Each passenger decides which floor to exit
      -- when he or she enters the elevator.  If the number of
      -- passengers exceeds the maximum capacity, the excess
      -- passengers have to wait for the next elevator visit.

      LOADED_PASS : INTEGER := 0;
      DESTINATION : array (FLOOR) of INTEGER := (others => 0);
      WAIT : array (FLOOR, MOTION)
                    of INTEGER := (others => (0, 0));

      procedure CHOOSE_DEST (FROM_FLOOR : FLOOR; GOING : MOTION;
                             PASS       : INTEGER) is
         -- Each passenger picks a destination floor.
         DEST_FLOOR : FLOOR;
      begin
         for I in 1 .. PASS loop
```

12

```
         loop
            DEST_FLOOR := RAND_FLR;
            -- The destination floor should be different
            -- from the starting floor and in the
            -- desired direction.
            if GOING = UP then
               exit when DEST_FLOOR > FROM_FLOOR;
            else
               exit when DEST_FLOOR < FROM_FLOOR;
            end if;
         end loop;
         DESTINATION(DEST_FLOOR):= DESTINATION(DEST_FLOOR) + 1;
      end loop;
   end CHOOSE_DEST;

   procedure LOAD_PASS (ARR_FLR  : FLOOR; ARR_DIR  : MOTION;
                        ARR_PASS : INTEGER) is
      -- Load the new arrivals and any people
      -- waiting in the queue.
      REQUEST_PASS : INTEGER :=
                     WAIT (ARR_FLR, ARR_DIR) + ARR_PASS;
      TAKE         : INTEGER;
   begin
      if REQUEST_PASS + LOADED_PASS > MAX_CAPACITY then
         TAKE := MAX_CAPACITY - LOADED_PASS;
      else
         TAKE := REQUEST_PASS;
      end if;

      CHOOSE_DEST (ARR_FLR, ARR_DIR, TAKE);
      WAIT (ARR_FLR, ARR_DIR) := REQUEST_PASS - TAKE;
      LOADED_PASS := LOADED_PASS + TAKE;

      OUTPUT.LOCK;
      if TAKE > 0 then
         SHOW_PASS (ARR_FLR, TAKE,
                    " being loaded to go ", ARR_DIR);
      end if;
```

```
      SHOW_PASS (ARR_FLR, LOADED_PASS, " on the elevator.");
      if WAIT (ARR_FLR, ARR_DIR) > 0 then
         SHOW_PASS (ARR_FLR, WAIT (ARR_FLR, ARR_DIR),
             " waiting for the next elevator to go ", ARR_DIR);
      end if;
      OUTPUT.UNLOCK;
   end LOAD_PASS;

begin -- ELEVATOR
   accept START do
      PASSENGER.START; -- Start the passengers.
   end START;

   loop
      select
         when CUR_FLR = ARR_FLR and then CUR_DIR = ARR_DIR =>
            accept LOAD_ARRIVING_PASS do
               -- Load new arrivals if possible.
               if ARR_PASS > 0 then
                  LOAD_PASS (ARR_FLR, ARR_DIR, ARR_PASS);
               end if;
               -- Send SIGUSR2 to the PASSENGER task to inform
               -- that passengers are either loaded or put in
               -- the queue.
               KILL (MY_PID, SIGUSR2);
            end LOAD_ARRIVING_PASS;
      or
         accept UNLOAD do
            -- Unload passengers whose destinations are the
            -- current floor.
            if DESTINATION (CUR_FLR) > 0 then
               OUTPUT.LOCK;
               SHOW_PASS (CUR_FLR, DESTINATION (CUR_FLR),
                   " being unloaded before going ", CUR_DIR);
               OUTPUT.UNLOCK;
               LOADED_PASS := LOADED_PASS -
                              DESTINATION (CUR_FLR);
               DESTINATION (CUR_FLR) := 0;
```

```
            end if;
         end UNLOAD;
      or
         accept LOAD_WAITING_PASS do
            -- Load any passengers waiting in the queue.
            if WAIT (CUR_FLR, CUR_DIR) > 0 then
               LOAD_PASS (CUR_FLR, CUR_DIR, 0);
            end if;
         end LOAD_WAITING_PASS;
      end select;
   end loop;
end ELEVATOR;

task body MANAGER is
   -- This task determines the floor number and direction of
   -- the elevator.  It calls ELEVATOR task to load and unload
   -- passengers on each floor.
begin
   accept START do
      -- Initialize the direction, and floor number.
      CUR_DIR := UP;
      CUR_FLR := FLOOR'FIRST;
      ELEVATOR.START; -- Start the elevator.
   end START;

   loop
      -- Unload passengers whose destinations are the
      -- current floor.
      ELEVATOR.UNLOAD;
      -- Load passengers waiting in the queue.
      ELEVATOR.LOAD_WAITING_PASS;

      if CUR_DIR = UP then
         CUR_FLR := CUR_FLR + 1;
         if CUR_FLR = FLOOR'LAST then
            CUR_DIR := DOWN; -- Reached the top floor,
                             -- change the direction.
         end if;
```

```
         else
            CUR_FLR := CUR_FLR - 1;
            if CUR_FLR = FLOOR'FIRST then
               CUR_DIR := UP; -- Reached the bottom floor,
                              -- change the direction.
            end if;
         end if;
      end loop;
   end MANAGER;

   procedure HANDLE_PASSENGER (SIG : SYSTEM.ADDRESS) is
      -- This is the handler for SIGUSR1.  It calls the ELEVATOR
      -- task to load the new arrivals.
   begin
      ELEVATOR.LOAD_ARRIVING_PASS;
   end HANDLE_PASSENGER;

   procedure ACKNOWLEDGE (SIG : SYSTEM.ADDRESS) is
      -- This is the handler for SIGUSR2.  It calls the PASSENGER
      -- task to continue generating passengers.
   begin
      PASSENGER.CONTINUE;
   end ACKNOWLEDGE;

begin -- MAIN

   -- Initialize the interrupt entry manager.
   INTERRUPT_MANAGER.INIT_INTERRUPT_MANAGER
      (NUMBER_OF_BUFFERS => 4,
       MAX_PARAM_AREA_SIZE => 1024,
       INTERRUPT_STACK_SIZE => 8024);

   -- Install handlers.
   INTERRUPT_MANAGER.INSTALL_HANDLER
      (HANDLER_ADDRESS => HANDLE_PASSENGER'ADDRESS,
       SIG             => SIGUSR1,
       PRIORITY => INTERRUPT_MANAGER.INTERRUPT_PRIORITY'LAST);
```

```
   INTERRUPT_MANAGER.INSTALL_HANDLER
      (HANDLER_ADDRESS => ACKNOWLEDGE'ADDRESS,
       SIG             => SIGUSR2,
       PRIORITY => INTERRUPT_MANAGER.INTERRUPT_PRIORITY'FIRST);

   MANAGER.START;
end MAIN;
```

F 12.12 Specification of the package INTERRUPT_MANAGER


```
package INTERRUPT_MANAGER is

   -- **********************************************************
   -- This package provides support for signal handlers.
   -- It must NOT be recompiled as it is already compiled in the
   -- predefined library.
   -- **********************************************************

   INTERRUPT_LEVELS : constant := 7;
   -- ------------------------------------------------------------
   -- Number of priority levels for interrupt entries.
   -- ------------------------------------------------------------

   type INTERRUPT_PRIORITY is range
      SYSTEM.PRIORITY'LAST + 1 .. SYSTEM.PRIORITY'LAST
                                  + INTERRUPT_LEVELS;
   for INTERRUPT_PRIORITY'SIZE use 32;
   -- ------------------------------------------------------------
   --  This type defines the range of allowed priorities for calls
   --  to interrupt entries.
   -- ------------------------------------------------------------

   type ACTION is (FIRST, LAST, REPLACED);
   for ACTION use (FIRST => 0, LAST => 1, REPLACED => 2);
   -- ------------------------------------------------------------
   -- This type defines the actions to be taken with regard to the
   -- previous handler for the signal (if any):
   --     * FIRST:    previous handler is to be called before
   --                 the Ada handler.
   --     * LAST:     previous handler is to be called after
   --                 the Ada handler.
   --     * REPLACED: previous handler is not to be called.
   -- ------------------------------------------------------------
```

```
type BUFFER_NUMBER is range 0..2**15-1;
-- -------------------------------------------------------------
-- Number of buffers to hold parameters of interrupt entry
-- calls that cannot be processed immediately.
-- -------------------------------------------------------------

type BYTE_SIZE is range 0..2**15-1;
-- -------------------------------------------------------------
-- Used to specify sizes in bytes.
-- -------------------------------------------------------------

MAX_HANDLERS : constant := 32;
-- -------------------------------------------------------------
-- Maximum number of installable handlers.
-- -------------------------------------------------------------

type HANDLER_NUMBER is range 0..MAX_HANDLERS;
-- -------------------------------------------------------------
-- Number of installed handlers.
-- -------------------------------------------------------------

NO_FREE_BUFFERS : BOOLEAN := FALSE;
-- -------------------------------------------------------------
-- Set to TRUE if a signal could not be handled because no
-- buffer was available to hold the parameters.  (In such cases
-- it is not possible to raise TASKING_ERROR because the entry
-- call was not made by a normal task.)  If NO_FREE_BUFFERS
-- becomes true it is recommended that the number of buffers
-- specified when calling INIT_INTERRUPT_MANAGER (see below)
-- be increased, or if possible increase the priority of the
-- called task.  The user can reset this variable to FALSE at
-- any time.
-- -------------------------------------------------------------
pragma SHARED (NO_FREE_BUFFERS);

TASK_NOT_CALLABLE : BOOLEAN := FALSE;
-- -------------------------------------------------------------
-- Set to TRUE if a signal could not be handled because the
```

```
-- called task was not callable (it was completed or
-- terminated).  The user can reset this variable to FALSE
-- at any time.
-- ------------------------------------------------------------

pragma SHARED (TASK_NOT_CALLABLE);

---------------------
-- Signal definition:
---------------------
type SIGNAL_NUMBER is range 0..32;
function SIGNAL is
   new UNCHECKED_CONVERSION (SIGNAL_NUMBER, SYSTEM.ADDRESS);

procedure INIT_INTERRUPT_MANAGER
            (NUMBER_OF_BUFFERS    : in BUFFER_NUMBER;
             MAX_PARAM_AREA_SIZE  : in BYTE_SIZE;
             INTERRUPT_STACK_SIZE : in BYTE_SIZE := 2048);
-- ------------------------------------------------------------
-- This procedure allocates the specified number of buffers to
-- hold the parameters of interrupt entry calls that cannot be
-- processed immediately, and allocates a signal stack of the
-- given size.  The size of each buffer is the maximum
-- parameter area size plus a fixed overhead of 28 bytes used
-- by the Ada runtime.  If the given signal stack size is zero,
-- all signals are handled on the current stack, and all stacks
-- must then have sufficient buffer space.
--
-- This procedure must be called before any Ada signal handler
-- can be installed and hence before any interrupt entry call
-- can be made from an Ada signal handler.  Signals can be
-- lost if the number of buffers is insufficient.  The number
-- of buffers required depends on the frequency of signals.
-- The number of buffers can be specified as zero if all Ada
-- signal handlers completely handle their signal and never
-- call an interrupt entry.
--
-- This procedure raises the exception STORAGE_ERROR if there
```

```
-- is not enough memory to allocate the required buffers and/or
-- the signal stack.
-- -----------------------------------------------------------
```

12

```
procedure INSTALL_HANDLER
     (HANDLER_ADDRESS  : in SYSTEM.ADDRESS;
      SIG              : in SYSTEM.ADDRESS;
      PRIORITY         : in INTERRUPT_PRIORITY
                            := INTERRUPT_PRIORITY'FIRST;
      ORIGINAL_HANDLER : in ACTION := REPLACED);
-- -----------------------------------------------------------
-- This procedure installs the Ada routine at the specified
-- address, as the Ada signal handler for the specified signal,
-- after saving the address of the current signal handler (if
-- any).  The specified priority determines the priority of all
-- entry calls made by the handler (all accept statements will
-- run with this priority).
--
-- The address of the Ada signal handler can be obtained with
-- the attribute 'ADDRESS (which is only valid after
-- elaboration of the procedure body).  The Ada signal handler
-- receives control with the signal it is handling blocked, but
-- other non-reserved signals are only blocked if they have an
-- Ada signal handler routine and it is currently active (has
-- been called in response to the signal but has not yet
-- returned).  The Ada signal handler must not make implicit
-- or explicit calls to the Ada runtime, other than a simple
-- entry call to the interrupt entry with the address clause
-- corresponding to the signal being handled.  Neither timed
-- nor conditional entry calls may be made from an Ada signal
-- handler.
--
-- The Ada signal handler must be a procedure with one
-- parameter of type ADDRESS, and without inner units.  The
-- procedure can only reference local or global objects
-- (excluding objects of enclosing frames).  The Ada signal
-- handler procedure must be compiled with checks off
-- (using the -R option).
```

```
   --
   -- This procedure raises the exception STORAGE_ERROR if
   -- MAX_HANDLERS handlers have already been defined.
   -- -------------------------------------------------------------

   procedure REMOVE_HANDLER (SIG : in SYSTEM.ADDRESS);
   -- -------------------------------------------------------------
   -- This routine removes the handler for the given signal and
   -- restores the original handler.  This procedure may be useful
   -- if for some reason the task that normally handles this
   -- signal is temporarily (or permanently) no longer able
   -- to do so.
   --
   -- This procedure raises the exception PROGRAM_ERROR if no
   -- handler has been installed for the given signal number.
   -- -------------------------------------------------------------

   function HANDLER_COUNT return HANDLER_NUMBER;
   -- -------------------------------------------------------------
   -- This function returns the number of installed Ada signal
   -- handlers.
   -- -------------------------------------------------------------

end INTERRUPT_MANAGER;
```

F 12.13 Ada Runtime Routine Descriptions

Tables 12-1 through 12-7 lists Ada Runtime System routines and their function. If calls to any of these routines appear in your Ada handler, the handler is "unsafe", as described in section "F 12.6 Associating an Ada Handler with an HP-UX Signal".

12

Table 12-1. Heap Management Routines

Routine	Description
ALLOC_GO	Allocates a global object.
ALLOC_LO	Allocates a local object.
ALLOC_TEMP	Allocates a temporary object.
ALLOC_TEMP_GH	Allocates a global temporary object.
FREE_LIST	Cleans up head objects at the end of a block.
FREE_TEMP	Frees a temporary object.
FREE_TEMP_GH	Frees a global temporary object.

Table 12-2.
Collection Management (no STORAGE_SIZE representation clause)

Routine	Description
ALLOC_SMALL_FIX_ELT	Allocates a space for a new object in the collection.
FREE_SMALL_FIX_ELT	Frees the space allocated to the object of the corresponding collection.
INIT_SMALL_FIX_ELT	Initializes the descriptor for a collection with small and fixed size elements.

Table 12-3. Collection Management (collections with a STORAGE_SIZE representation clause)

Routine	Description
Fixed element size	
ALLOC_FIX_SS_ELT	Allocates a space for a new object in the collection.
FREE_FIX_SS_ELT	Frees the space allocated to the object of the corresponding collection.
INIT_FIX_SS_ELT	Initializes the descriptor for a collection with fixed size elements.
Variable element size	
ALLOC_VAR_SS_ELT	Allocates a space for a new object in the collection.
FREE_VAR_SS_ELT	Frees the space allocated to the object of the corresponding collection.
INIT_VAR_SS_ELT	Initializes the descriptor for a collection with variable size elements.

12

Table 12-4. Tasking Routines

Routine	Description
ABORT_STMT	Aborts the tasks in the argument lists and abort all their dependents.
ACCEPT_STMT	Implementation of a simple **accept** statement.
ACTIVATE_COLLECTION	Called after the elaboration of a declarative region that contains task objects and at the end of the execution of an allocator of an object with one or more task components. This routine activates a collection of tasks in parallel.
COMPLETE_MASTER	Called when exiting a block or subprogram master unit to complete a master unit and deallocate its resource.
COMPLETE_TASK	Called when a task body completion point is reached and is about to execute the cleanup sequence of its task body to terminate the task and its dependents.
COND_CALL	Implementation of a conditional entry call.
COND_SELECT	Implementation of a **select** statement with an else part.
CREATE_TASK	Called when a single task specification or task object declaration is elaborated and when an allocator is executed that has task components. This routine creates a new task object.
CURRENT_OBJECT_ OF_TASK_TYPE	Called when a task is referenced from the task body of a task type. This routine maps a task unit name to the referenced task when the unit name is used to refer to a task object within its body.
DELAY_STMT	Implementation of a **delay** statement.

Continued on the next page.

Table 12-4. Tasking Routines (Continued)

Routine	Description
DESTROY_COLLECTION	Called when an exception is raised during the execution of an allocator for an object with one or more task components. This routine terminates any unactivated tasks in the collection.
END_ACTIVATION	Called at the end of a successful task activation to make the activated task eligible to run.
ENTRY_CALL	Implementation of a simple entry call. *This call is safe as long as it calls a task entry declared as an interrupt entry for the HP-UX signal that caused the Ada signal handler to be invoked.*
ENV_TASK_MASTER	Called when either the main program has not been invoked and a library package is being elaborated, or the main program has been invoked and an allocator is to be executed for an access type in a library package. This routine initializes an activation collection for tasks directly dependent on a library package.
INIT_COLLECTION	Called at the beginning of a unit that declares static task objects, and as the first action of the execution of an allocator for an object containing any tasks. This routine initializes an empty collection of tasks to be activated in parallel.
INIT_HANDLER	Called at the occurrence of an address clause for a task entry. This routine declares that an entry is associated with an interrupt in the calling task.

Continued on the next page.

Table 12-4. Tasking Routines (Continued)

Routine	Description
INIT_MASTER	Called at the beginning of a master unit to initialize an internal data structure MASTER_INFO.
NULL_BODY_ACCEPT_STMT	Implementation of an **accept** statement that has a **null** statement list in its body.
SELECT_WITH_TERMINATE	Implementation of a **select** statement with a terminate alternative.
SIMPLE_SELECT	Implementation of a **select** statement with only accept alternatives.
SIMPLE_TIMED_SELECT	Implementation of a **select** statement with one delay alternative without a guard or with a static open guard.
TERMINATE_COMPLETE	Called when task body is complete and cleanups have been performed. The routine propagates the termination information up the hierarchy and deallocates the work space; the run-time schedules another task.
TIMED_CALL	Implementation of a timed entry call.
TIMED_SELECT	Implementation of a **select** statement with several delay alternatives or with one guarded delay alternative.

Table 12-5. Attributes Routines

Routine	Description
ENUM_POS	Implementation of T'POS, where T is an enumeration type.
ENUM_WIDTH	Implementation of T'WIDTH, where T is an enumeration type.
FIXED_FORE	Implementation of T'FORE, where T is a fixed point subtype.
FIXED_LARGE	Implementation of T'LARGE, where T is a fixed point subtype.
FIXED_MANTISSA	Implementation of T'MANTISSA, where T is a fixed point subtype.
INTEGER_IMAGE	Implementation of T'IMAGE, where T is an integer type.
INTEGER_VALUE	Implementation of T'VALUE, where T is an integer type.
INTEGER_WIDTH	Implementation of T'WIDTH, where T is an integer type.

Table 12-6. Attributes for Tasks Routines

Routine	Description
CALLABLE	Implementation of T'CALLABLE, where T is a task.
COUNT	Implementation of E'COUNT, where E is an entry of a task.
TERMINATED	Implementation of T'TERMINATED, where T is a task.

Table 12-7.
Support for Enumeration Representation Clauses Routines

Routine	Description
ENUM1_PRED	Implementation of T'PRED attribute for types whose VAL is implemented on one byte; that is, types with no more than 128 values.
ENUM2_PRED	Implementation of T'PRED attribute for types whose VAL is implemented on two bytes; that is, types with more than 128 values but less than 32768 values.
ENUM4_PRED	Implementation of T'PRED attribute for types whose VAL is implemented on four bytes; that is, types with more than 32767 values.
ENUM1_SUCC	Implementation of T'SUCC attribute for types whose VAL is implemented on one byte; that is, types with no more than 128 values.
ENUM2_SUCC	Implementation of T'SUCC attribute for types whose VAL is implemented on two bytes; that is, types with more than 128 values. but less than 32768 values.
ENUM4_SUCC	Implementation of 'SUCC attribute for types whose VAL is implemented on four bytes; that is, types with more than 32767 values.
ENUM1_VAL_TO_POS	Convert T'VAL to T'POS for types whose VAL is implemented on one byte; that is, types with no more than 128 values.
ENUM2_VAL_TO_POS	Convert T'VAL to T'POS for types whose VAL is implemented on two bytes; that is, types with more than 128 values. but less than 32768 values.
ENUM4_VAL_TO_POS	Convert T'VAL to T'POS for types whose VAL is implemented on four bytes; that is, types with more than 32767 values.

Index

A

access
  direct, 8-38
  sequential, 8-38
access protection, 8-11
access rights, 8-3, 8-7
access types
  alignment, 4-24
  as function results, 11-12
  bit representation, 11-11
  caution, 11-12
  collection size specification, 4-22, 4-23
  FORTRAN, 11-38
  general considerations, 11-11
  illustration of passing methods, 11-11
  internal representation, 4-22
  I/O operations, 8-13
  minimum size, 4-24
  not returned as function results, 11-38
  passing to external subprograms, 11-11
  size, 4-24
  value of 'STORAGE_SIZE, 4-22
Ada
  handler, 12-7, 12-11
  subprograms, 1-14
Ada Runtime System (Ada RTS), 1-6, 12-2, 12-9
'ADDRESS, 2-2
address
  pointer, 11-38, 11-49, 11-52
  scalar parameter, 11-33
address clauses
  constants, 6-2
  data objects, 6-3
  objects, 6-1
  packages, 6-2
  subprograms, 6-2
  task entries, 6-3
  tasks, 6-2
alignment
  array types, 4-32
  enumeration types, 4-5
  integer types, 4-11
  record types, 4-35, 4-49
alignment clause, 10-7
APPEND, 8-27
'ARRAY_DESCRIPTOR, 2-1
'ARRAY_DESCRIPTOR implicit component, 4-46
array objects, 11-26
arrays
  as function results, 11-39
  caution, 11-39
  unconstrained, 7-1
array types
  alignment, 4-29
  as function results, 11-13, 11-53
  C language, 11-26
  correspondence with HP Pascal types, 11-53
  default size, 4-28
  gaps between components, 4-28
  general considerations, 11-13

layout, 1-23, 4-28
minimum size, 4-28
Pascal, 11-53
passing to external subprograms, 11-13, 11-39, 11-53
pragma PACK, 4-28
size of dynamic arrays, 4-31
string types, 11-13
ASCII, 3-21, 8-10
assembly language
access types, 11-20
array types, 11-20
Boolean types, 11-19
calling conventions, 11-18
character types, 11-19
enumeration types, 11-19
floating point types, 11-19
integer types, 11-19
record types, 11-20
scalar types, 11-19
subprograms, 11-18
asynchronous signals, 9-15
attributes
'ADDRESS, 2-2
'ARRAY_DESCRIPTOR, 2-1
for tasks routines, 12-33
implementation-dependent, 2-1
'OFFSET, 2-1
'RECORD_DESCRIPTOR, 2-1
'RECORD_SIZE, 2-1
routines, 12-33
SYSTEM.ADDRESS.
ADDRESS'IMPORT, 2-3
'VARIANT_INDEX, 2-1

B

binary files, 8-37
binder
and pragma INTERFACE, 1-6
bind-time issues, 1-19
bit ordering
component clause, 4-35
blocked signal, 11-65, 11-67
BOOLEAN'POS(FALSE), 11-51
BOOLEAN'POS(TRUE), 11-51
Boolean types
as function results, 11-8
bit representation, 11-8
C language, 11-24
converting to integer types, 11-24
general considerations, 11-8
Pascal, 11-51
passing to external subprograms, 11-8
predefined, 3-10
returned as function results, 11-24
buffering, 8-8, 8-11, 8-23
BUFFER_SIZE, 8-23

C

calling conventions, 1-3, 11-1
catenation operators, 3-21
change in representation, 4-37
char, 11-26, 11-51
CHARACTER, 11-23, 11-24, 11-39, 11-51
CHARACTER type, 3-19
character types
as function results, 11-9
bit representation, 11-9, 11-51
calling FORTRAN, 11-32
calling Pascal, 11-51
correspondence with HP C types, 11-24
general considerations, 11-9
checks, 11-4, 11-13
C language
access types as function results, 11-22
array types, 11-26
bit representation of parameters passed to, 11-21
Boolean types, 11-24
calling, 11-21

character types, 11-24
enumeration types, 11-23
integer correspondence, 11-23
integer types as function results, 11-23
real types, 11-21
record types, 11-31
scalar types, 11-21, 11-22
scalar types as function results, 11-22
types returned as function results, 11-22
collection management, 12-28, 12-29
collections of objects, 4-54
compiler
limitations, 10-1
compiler-generated objects, 4-52
COMPLEX, 11-45
components of a record, organization of, 11-5
composite types, 11-2
consistency checks, 11-4
constants, 6-2
CONSTRAINT_ERROR exception, 3-21, 9-7
CONTROLLED, 1-29
COUNT'LAST, 8-3
CREATE procedure, 8-1, 8-3, 8-7, 8-8, 8-18

D

data allocation, 4-50
data objects, 6-3
default access protection, 8-11
deferred processing, 12-3
delay statement, 9-11, 11-65
device
disks, 8-12
terminal, 8-10, 8-24
dimensions
declared in reverse order, 11-39
illustration of reversed order, 11-41
direct access, 8-38
direct files
elements, 8-8
index, 8-8
DIRECT_IO files, 8-8, 8-37
DOUBLE COMPLEX, 11-45
DOUBLE PRECISION, 11-37
double precision reals, 11-25
DURATION type, 3-19
dynamic components, 4-40, 11-17
dynamic memory management, 4-54

E

ELABORATE pragma, 1-1, 1-24
ELEMENTARY_FUNCTIONS_EXCEPTIONS, 5-1
elements, 8-8
END_OF_FILE, 8-10
END_OF_PAGE, 8-10
enumeration
alignment, 4-5
enumeration types
as function results, 11-8, 11-51
bit representation, 11-7, 11-50
default size, 4-5
FORTRAN, 11-34
general considerations, 11-7
internal representation, 4-2, 4-4, 4-5
minimum size, 4-4, 4-5
Pascal, 11-50
passing to external subprograms, 11-7, 11-34, 11-50
syntax, 4-2
unsigned representation, 4-5
errors
USE_ERROR exception, 8-25
exceptions, 1-17
CONSTRAINT_ERROR, 3-21, 9-7
handling, 9-6, 9-7
NUMERIC_ERROR, 3-21, 9-7
predefined, 3-21

PROGRAM_ERROR, 3-21, 9-7
STORAGE_ERROR, 3-21, 9-7
TASKING_ERROR, 3-21
execution
of delay statements, 11-65
EXPORT pragma, 1-1, 1-2, 1-8, 1-18
extension, 11-7, 11-23
external files
access rights, 8-7
appending to, 8-27
associating Ada file objects with, 8-1, 8-11
correspondence with HP-UX files, 8-3
definition, 8-1
errors, 8-25
existing file specified to CREATE, 8-4
names, 8-3
protection of, 8-11
shared, 8-11, 8-12
standard implementation of, 8-7
EXTERNAL_NAME pragma, 1-1, 1-2, 1-12, 1-18
external subprograms
calling conventions, 11-1
delay statements, 9-11
exceptions to calling conventions, 11-1
general aspects of calling, 11-1
I/O on files opened by Ada, 11-67
parameter values, 11-4
passing access types to, 11-11
passing array types to, 11-13, 11-39, 11-53
passing Boolean types to, 11-8
passing enumeration types to, 11-34, 11-50
passing floating point types to, 11-10, 11-25
passing integer types to, 11-7, 11-33, 11-49
passing record types to, 11-60
passing string types to, 11-13, 11-26, 11-41
passing task types to, 11-17
potential problems using, 11-64
pragma INTERFACE, 1-4
protecting code with a critical section, 11-67
signals, 9-11
types not passed as parameters to, 11-2
external subroutines names, 1-6

F

FIELD'LAST, 8-3
FIFO
control, 8-31
FIFO_EOF, 8-31
files
Ada definition, 8-1
appending, 8-27
associate NAME with file object, 8-1
associating external with file object, 8-1
binary, 8-37
blocking, 8-28
buffering, 8-23, 8-26
characteristics, 8-1
correspondence of Ada with external, 8-1
direct, 8-8, 8-11
DIRECT_IO, 8-37
disk, 8-23
external, 8-3, 11-4
HP-UX pathname, 8-1
I/O, 8-1
length of elements, 8-8
name of external, 8-3
object, 1-6, 8-1
protecting, 8-21
protection flags, 8-21

sequential, 8-7, 8-11, 8-12, 8-27
SEQUENTIAL_IO, 8-37
shared, 8-11, 8-12, 8-26
tasking, 8-11
terminal input, 8-18
terminator, 8-10
text, 8-10, 8-11, 8-27, 8-35
fixed point
predefined, 4-16
types, 4-16
fixed point types
alignment, 4-21
as function results, 11-10, 11-25, 11-52
default size, 4-20
external subprograms, 11-2
general considerations, 11-10
internal representation, 4-17
minimum size, 4-18, 4-21
parameters, 11-25
unsigned representation, 4-18, 4-21
value of 'SMALL, 4-17
FLOAT, 3-17, 11-25, 11-37, 11-39, 11-52
floating point types
alignment, 4-15
as function results, 11-10
bit representation, 11-10, 11-25
calling FORTRAN, 11-32
default size, 4-15
FORTRAN, 11-37
general considerations, 11-10
internal representation, 4-12
minimum size, 4-15
Pascal, 11-52
passing to external subprograms, 11-10
passing to HP C, 11-25
predefined, 4-12
FORM parameter, 8-1
appending to a file, 8-27
attributes, 8-18
blocking, 8-28
customizing characteristics, 8-7
FIFO control, 8-31
file buffering, 8-23
files protection flags, 8-21
file structuring, 8-35
format, 8-18
shared files, 8-25
terminal input, 8-33
FORTRAN
access types, 11-38
array types, 11-39
calling subprogram, 11-32
enumeration types, 11-34
equivalence of types, 11-45
fixed point types, 11-37
floating point type correspondence, 11-37
floating point types, 11-37
integer correspondence, 11-33
integer types, 11-33
other types, 11-45
record types, 11-45
scalar types, 11-33
string types, 11-41
function results
access types as, 11-38
array types as, 11-39, 11-53
Boolean types as, 11-24
enumeration types as, 11-51
fixed point types as, 11-52
floating point types as, 11-25
integer types as, 11-33
record types as, 11-31, 11-45, 11-60
types returned as, 11-22
types returned from HP C, 11-23
types returned from HP FORTRAN 77, 11-32
types returned from HP Pascal, 11-48

G

gap sizes, 4-29
generating code, 1-24
GENERIC_ELEMENTARY_FUNCTIONS, 5-1
global dynamic objects, 4-55
GROUP (user access categories), 8-21

H

handler, 12-11
HANDLER_COUNT, 12-11
heap management routines, 12-28
heap storage, 4-57
HP-UX
  library function, 1-6
  pragma INTERFACE, 1-6
  signals, 9-1, 9-2, 9-15
  system calls, 1-6
  utilities and routines, 11-67

I

immediate processing, 12-2
implementation-dependent
  attributes, 2-1
  characteristics of external files, 8-7
implicit components, 4-37
  'ARRAY_DESCRIPTOR, 4-46
  'OFFSET, 4-43
  'RECORD_DESCRIPTOR, 4-46
  'RECORD_SIZE, 4-44
  'VARIANT_INDEX, 4-44
implicit parameters, 11-32, 11-41
'IMPORT, 2-3
IMPROVE pragma, 1-1, 1-23
INDENT pragma, 1-1, 1-21
index
  direct files, 8-8
  set when file opened, 8-27
indirect components, 4-39
INLINE pragma, 1-1, 1-25
INSTALL HANDLER, 12-7
INTEGER, 11-7, 11-23, 11-39
integer types
  alignment, 4-11
  as function results, 11-7, 11-33
  bit representation, 11-49
  calling FORTRAN, 11-32
  compatibility with HP Pascal types, 11-48
  correspondence between Ada and HP C, 11-23
  correspondence with FORTRAN integers, 11-33
  default size, 4-9
  general considerations, 11-7
  internal representation, 4-6
  minimum size, 4-7, 4-10
  Pascal, 11-49
  passing to external subprograms, 11-7, 11-33, 11-49
  performance, 4-11
  predefined, 4-6
  size, 4-9
  unsigned representation, 4-7, 4-10
interactive devices, 8-12, 11-65
INTERFACE
  pragma, 11-21, 11-32, 11-46
INTERFACE_NAME pragma, 1-1, 1-2, 1-4
INTERFACE pragma, 1-1, 1-2, 1-3
internal codes of enumeration literals, 4-2
interrupt, 11-64, 11-65
interrupt entries
  address clauses, 12-12
  examples, 12-13
  HP-UX signals, 9-14
  immediate processing step, 12-4
  initializing, 12-5
INTERRUPT_HANDLER, 12-12
INTERRUPT_MANAGER pragma, 12-5

INTERRUPT_MANAGER specification, 12-23
I/O
  access types, 8-13
  calls to interactive devices, 11-65
  considered as external files, 8-3
  implementation-generated characteristics, 8-1
  intermixing HP-UX utilities and Ada routines, 11-67
  Local Area Networks, 8-13
  multiple operations, 8-11
  operation, 8-12
  packages, 8-1, 8-2
  performed on objects, 8-1
  readable, 8-10
  system dependencies, 8-16

K

keyboards, 8-12

L

LAN (Local Area Networks), 8-13
length clauses
  collection size specification, 4-22, 4-23
  size specification, 4-1, 4-5, 4-9, 4-10, 4-15, 4-20, 4-21, 4-28, 4-29, 4-31, 4-32, 4-33, 4-48
  'SMALL of a fixed point type, 4-17
  storage, 4-25
  task activation size specification, 4-25
libraries
  HP C, 11-22
library units, 5-1
limitations
  Ada development environment, 10-5
  compiler, 10-1
  on pragma PACK, 4-31
  on record representation clauses, 4-37
  on the value of 'SMALL, 4-17
  opening or creating files, 10-6
  path and component sizes, 10-6
  user-written applications, 10-6
line terminator, 8-10
link editor, 1-6
LIST pragma, 1-1, 1-22
Local Area Networks (LAN), 8-13
local objects, 4-56
LONG_FLOAT, 3-18, 11-25, 11-39, 11-52
LONGREAL, 11-52

M

MASK, 8-11
MATH_LIB, 5-1
MATH_LIB_LIBM, 5-1
MEMORY_SIZE, 1-29

N

NAME parameter, 8-1
names
  for predefined library units, 5-1
networks, 8-13
NFS, 8-13, 8-14
non-blocking, 8-30
NON_BLOCKING attribute, 8-30
NOT_SHARED mode, 8-25
NUMERIC_ERROR exception, 3-21, 9-7

O

object deallocation, 4-52
object files, 1-6
object library, 1-6
objects
  collections, 4-54
  compiler-generated, 4-52
  global dynamic, 4-55
  local dynamic, 4-56
  programmer-generated, 4-53
  temporary, 4-56
'OFFSET, 2-1

offset, 11-16
'OFFSET implicit component, 4-43
OPEN, 8-27
OPEN procedure, 8-1, 8-3, 8-7, 8-8, 8-18
operating system, 8-1, 11-64, 11-65
OPTIMIZE, 1-29
order
  reversed, 11-39
  row major, 11-40
other types, 11-45
OWNER (user access categories), 8-21

P

packages, 6-2
  ASCII, 3-21
  I/O, 8-2, 8-3
  STANDARD, 3-10, 3-11, 3-13, 3-15, 3-17, 3-19, 3-21, 3-24
  SYSTEM, 3-1
PACK pragma, 1-1, 1-23
PAGE pragma, 1-1, 1-22
page terminator, 8-10
parameters
  modification of values, 11-4
  passing conventions, 11-21, 11-32, 11-33, 11-41, 11-46, 11-49
  passing modes, 11-1, 11-4, 11-32, 11-33
  values and bit representations, 11-5
Pascal
  access types, 11-52
  array types, 11-53
  Boolean types, 11-51
  character types, 11-51
  enumeration types, 11-50
  fixed point types, 11-52
  floating point types, 11-52
  integer types, 11-48, 11-49
  record types, 11-60
  scalar types, 11-48
  string types, 11-56
  subprograms, 11-46
passing conventions, 1-2
pathname, 8-1, 8-3
performance
  integer types, 4-11
pointer, 11-7, 11-8, 11-11
POSITIVE_COUNT, 8-8, 8-10
pragmas
  CONTROLLED, 1-29
  ELABORATE, 1-1, 1-24
  EXPORT, 1-1, 1-2, 1-8, 1-18
  EXTERNAL_NAME, 1-1, 1-2, 1-12, 1-18
  implementation-specific, 1-1
  IMPROVE, 1-1, 1-23, 4-1, 4-38, 4-47
  INDENT, 1-1, 1-21
  INLINE, 1-1, 1-25
  INTERFACE, 1-1, 1-2, 1-3, 11-21, 11-32, 11-46
  INTERFACE_NAME, 1-1, 1-2, 1-4
  INTERRUPT_MANAGER, 12-5
  LIST, 1-1, 1-22
  MEMORY_SIZE, 1-29
  OPTIMIZE, 1-29
  PACK, 1-1, 1-23, 4-1, 4-5, 4-10, 4-21, 4-28, 4-29, 4-31, 4-32, 4-33
  PAGE, 1-1, 1-22
  predefined, 1-1
  PRIORITY, 1-1, 1-27
  SHARED, 1-1, 1-28
  STORAGE_UNIT, 1-29
  SUPPRESS, 1-1, 1-26
  SYSTEM_NAME, 1-29
  tasking programs, 1-27
  unimplemented, 1-29
predefined
  integer types, 4-6
  library units, 5-1
predefined exceptions
  CONSTRAINT_ERROR, 3-21

NUMERIC_ERROR, 3-21
PROGRAM_ERROR, 3-21
STORAGE_ERROR, 3-21
TASKING_ERROR, 3-21
predefined types
CHARACTER, 3-19
DURATION, 3-19
STRING, 3-20
PRIORITY pragma, 1-1, 1-27
procedures
CREATE, 8-1, 8-3, 8-7, 8-8, 8-18
INSTALL HANDLER, 12-7
OPEN, 8-1, 8-3, 8-7, 8-8, 8-18
PROGRAM_ERROR exception, 3-21, 9-7
programmer-generated objects, 4-53
program termination, 4-53, 9-12
protection
external files, 8-11
of interfaced code from signals, 9-15, 11-67
parameter, 11-4
PUT, 8-10

R

random access devices, 8-12
ranges, 11-26, 11-48
REAL, 11-65
real types
bit representation, 11-10
fixed point types, 11-10, 11-25, 11-37, 11-52
floating point types, 11-10, 11-25, 11-37, 11-52
general considerations, 11-10
'RECORD_DESCRIPTOR, 2-1
'RECORD_DESCRIPTOR implicit component, 4-46
records
representation clauses, 11-16
size, 11-17
'RECORD_SIZE, 2-1
RECORD_SIZE, FORM attribute, 8-37
'RECORD_SIZE implicit component, 4-44
record types
alignment clause, 10-7
as function results, 11-17, 11-31, 11-45, 11-60
assignment to a discriminant, 11-17
bit representation, 11-15
C language, 11-31
compiler adds implicit components, 11-17
components, default size, 4-33, 4-36
components, minimum size, 4-33
components reordered by compiler, 11-15
composite components, 11-17
direct components, 4-39
dynamic components, 11-17
FORTRAN, 11-45
general considerations, 11-15
implicit component 'ARRAY_DESCRIPTOR, 4-46
implicit component 'OFFSET, 4-43
implicit component 'RECORD_DESCRIPTOR, 4-46
implicit component 'RECORD_SIZE, 4-44
implicit components, 4-37
implicit component 'VARIANT_INDEX, 4-44
indirect components, 4-39
layout, 1-23, 4-33, 4-36, 11-15
offset, 11-16
parameter passing modes, 11-4
Pascal, 11-60
passing to external subprograms, 11-15, 11-60
record representation clauses, 11-15

size of record, 11-17
syntax, 4-33
unconstrained, 7-1
RECORD_UNIT, FORM attribute, 8-37
reformatter, 1-21
REMOVE_HANDLER, 12-11
representation clauses
data objects, 4-1
enumeration, 4-1, 4-2
enumeration types, 11-7
pragma PACK, 1-23
record, 4-1, 4-5, 4-10, 4-21, 4-33, 4-35, 4-36, 4-43, 4-44, 4-46, 4-47, 4-49
reserved signal, 11-67
restrictions, 7-1, 8-3, 10-1, 11-25
RESUME_ADA_TASKING, 11-67
reverse order, 11-39
row major order, 11-40
run-time, 11-64, 11-67
checks, 11-4
system, 1-6
runtime
descriptions, 12-28

S

scalar types
Boolean types, 11-8, 11-51
calling FORTRAN, 11-32
calling Pascal, 11-51
character types, 11-9
enumeration types, 11-7, 11-50
FORTRAN, 11-33
general considerations, 11-6
integer types, 11-7, 11-33, 11-48
Pascal, 11-48
real types, 11-10, 11-37, 11-52
string types, 11-41, 11-43
scheduling, 11-65
sequential access, 8-38
SEQUENTIAL_IO files, 8-7, 8-37
SHARED mode, 8-25
SHARED pragma, 1-1, 1-28
SHORT_INTEGER, 3-15, 4-10, 11-7, 11-23, 11-39, 11-49
SHORT_SHORT_INTEGER, 3-14, 11-7, 11-23, 11-26, 11-49
SIGALRM signal, 9-2, 9-10, 9-11, 9-15, 11-65, 11-67
SIGBUS signal, 9-2, 9-8
SIGFPE signal, 9-2, 9-7
SIGHUP signal, 9-12
SIGILL signal, 9-2, 9-8
SIGINT signal, 9-12
signals
asynchronous, 9-15
blocked, 11-67
handlers, 1-20
HP-UX, 9-1, 9-2, 9-15
interrupt entries, 9-14
interruption, 11-65
SIGALRM, 9-2, 9-10, 9-11, 9-15, 11-65, 11-67
SIGBUS, 9-2, 9-8
SIGFPE, 9-2, 9-7
SIGHUP, 9-12
SIGILL, 9-2, 9-8
SIGINT, 9-12
SIGPIPE, 9-12
SIGPROF, 9-2, 9-10, 9-11, 9-15
SIGQUIT, 9-12
SIGSEGV, 9-2, 9-8
SIGTERM, 9-12
SIGVTALRM, 9-2, 9-10, 9-11, 9-15, 11-65, 11-67
subprograms, 11-64
SIGPIPE signal, 9-12
SIGPROF signal, 9-2, 9-10, 9-11, 9-15
SIGQUIT signal, 9-12
SIGSEGV signal, 9-2, 9-8
SIGTERM signal, 9-12
sigvector(2), 11-64, 11-65

SIGVTALRM signal, 9-2, 9-10, 9-11,
 9-15, 11-65, 11-67
single precision reals, 11-25
size of record, 11-17
slow HP-UX routines, 11-65
stack, 11-21
STANDARD, 3-21
STANDARD_INPUT, 8-12
STANDARD_OUTPUT, 8-12
STANDARD package, 3-10, 3-11, 3-13,
 3-15, 3-17, 3-19, 3-24
stderr, 8-12
stdin, 8-12
stdout, 8-12
STORAGE_ERROR exception, 3-21,
 9-7
storage for a task activation, 4-25
STORAGE_UNIT
 pragma, 1-29
 record representation clause, 4-36
STRING type, 3-20
string types
 Pascal, 11-56
 passing to external subprograms,
  11-13, 11-41
 scalar types, 11-43
 special case of arrays, 11-26
subprograms, 1-14, 6-2
 predefined, 8-10
support for enumeration representation,
 12-34
SUPPRESS pragma, 1-1, 1-26
SUSPEND_ADA_TASKING, 11-67,
 12-11
symbolic links, 8-15
system
 dependencies, 8-16
SYSTEM, 3-1, 11-33
SYSTEM.ADDRESS'IMPORT, 2-3
SYSTEM_ENVIRONMENT, 5-1
SYSTEM_ENVIRONMENT.
 SUSPEND_ADA_TASKING, 12-11
SYSTEM_NAME, 1-29

T

tapes, 8-12
TASKING_ERROR exception, 3-21
tasking routines, 12-30
tasks, 6-2
 entries, 6-3
 management, 9-1, 9-10
 minimum size of stack, 4-26
 rescheduling, 11-65
 scheduling, 11-65
 time-slice amount, 11-65
task types
 alignment, 4-27
 as function results, 11-17
 external subprograms, 11-2
 general considerations, 11-17
 internal representation, 4-25
 minimum size, 4-27
 passing to external subprograms,
  11-17
temporary objects, 4-56
terminal
 device, 8-10, 8-24
 input, 8-33
terminating programs, 9-12
terminator
 character, 11-26
 file, 8-10
 line, 8-10
 page, 8-10
 representation, 8-35
text
 files, 8-35
 processing tools, 1-21
TEXT_IO files, 8-10
time-slicing, 11-65
TSS routines, 12-8

type conversions,unchecked, 7-1
type representation, 4-1
type support subprograms, 12-8

U

UNCHECKED_CONVERSION
  access types, 4-22
  enumeration types, 4-3
  interfacing with external data structures, 4-22
  limitations, 7-1
UNCHECKED_DEALLOCATION, 8-13
unchecked type conversions, 7-1
unconstrained record types, 7-1
unconstrained strings, 11-26
universal_fixed, 3-13
universal_integer, 3-11
universal types, 3-11
USE_ERROR, 8-3, 8-8, 8-19, 8-25
user access categories
  GROUP, 8-21
  OWNER, 8-21
  WORLD, 8-21
user-written applications, 10-6

V

VAR, 11-53, 11-60
'VARIANT_INDEX, 2-1
variant index, 11-17
'VARIANT_INDEX implicit component, 4-44

W

WORLD (user access categories). 8-21