

FILE COPY 12

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
s regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

AD-A233 625

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED

Final: 11 Feb 1991 to 01 Mar 1993

4. TITLE AND SUBTITLE

Ada Compiler Validation Summary Report: Hewlett Packard, HP 9000 series 300
Ada Compiler, Version 5.35, HP 9000 Series 300 Model 370 (Host & Target),
901022W1.11049

5. FUNDING NUMBERS

6. AUTHOR(S)

Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB
Dayton, OH 45433

8. PERFORMING ORGANIZATION
REPORT NUMBER

AFV-VSR-399-0191

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 300 Ada Compiler, Version 5.35, Wright-Patterson AFB, OH, HP 9000 300 Model 370
(Host & Target), ACVC 1.11

14. SUBJECT TERMS

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

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION
OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION
UNCLASSIFIED

19. SECURITY CLASSIFICATION
OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT

Certificate Information

The following Ada implementation was tested and determined to pass ACVC

1.11. Testing was completed on 22 October 1990.

```
Compiler Name and Version: HP 9000 Series 300 Ada Compiler
                           Version 5.35
```

```
Host Computer System:      HP 9000 Series 300 Model 370
                           HP-UX, Version A.07.00
```

Target Computer System: HP 9000 Series 300 Model 370
HP-UX, Version A.07.00

Customer Agreement Number: 90-08-14-HPC

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

As a result of this validation effort, Validation Certificate 901022W1.11049 is awarded to Hewlett Packard. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.

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

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

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

A-1

AVF Control Number: AVF-VSR-399-0191
11 February 1991
90-08-14-HPC

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901022W1.11049
Hewlett Packard
HP 9000 Series 300 Ada Compiler, Version 5.35
HP 9000 Series 300 Model 370 => HP 9000 Series 300 Model 370

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

DECLARATION OF CONFORMANCE

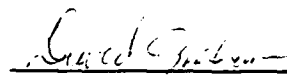
Compiler Implementor: Hewlett Packard Company, California Language Lab
Ada Validation Facility: ASD/SCEL, Wright Patterson AFB, OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.11.

Base Configuration

Base Compiler Name: HP 9000 Series 300 Ada Compiler, Version 5.35.
Host Architecture ISA: HP 9000 Series 300 Model 370
Host OS and Version: HP-UX, Version A 07.00
Target Architecture ISA: HP 9000 Series 300 Model 370
Target OS and Version: HP-UX, Version A.07.00

Implementer's Declaration

I, the undersigned, representing Hewlett Packard Company, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compilers listed in this declaration. I declare that Hewlett Packard Company is owner of record of the Ada language compilers listed above and, as such, is responsible for maintaining said compilers in conformance to ANSI/MIL-STD-1815A. All certificates and registration for the Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

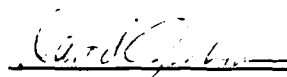


Hewlett Packard Company
David Graham
Ada R&D Section Manager

Date: 10/22/90

Owner's Declaration

I, the undersigned, representing Hewlett Packard Company, take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.



Hewlett Packard Company
David Graham
Ada R&D Section Manager

Date: 10/22/90

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	2-1
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

INTRODUCTION

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 SPRT13, 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 SPRT13 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.

INTRODUCTION

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

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

1.4 DEFINITION OF TERMS

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

INTRODUCTION

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

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AV0. The rationale for withdrawing each test is available from either the AV0 or the AVF. The publication date for this list of withdrawn tests is 02 September 1990.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
B83022B	B83022H	B83025B	B83025D	B83026B	B85001L
C83026A	C83041A	C97116A	C98003B	BA2011A	CB7001A
CB7001B	CB7004A	CC1223A	BC1226A	CC1226B	BC3009B
BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E	CD2A23E
CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C	BD3006A
CD4022A	CD4022D	CD4024B	CD4024C	CD4024D	CD4031A
CD4051D	CD5111A	CD7004C	ED7005D	CD7005E	AD7006A
CD7006E	AD7201A	AD7201E	CD7204B	BD8002A	BD8004C
CD9005A	CD9005B	CDA201E	CE2107I	CE2119B	CE2205B
CE2405A	CE3111C	CE3118A	CE3411B	CE3412B	CE3812A
CE3814A	CE3902B				

2.2 INAPPLICABLE TESTS

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

IMPLEMENTATION DEPENDENCIES

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

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

The following 21 tests check for the predefined type `LONG_INTEGER`:

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

C35702A, C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`.

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

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

C45536A, C46031B, C46033B, and C46034B contain 'SMALL representation clauses which are not powers of two or ten.

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

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

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

LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests) check for pragma `INLINE` for procedures and functions.

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

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten type's small. (See section 2.3)

IMPLEMENTATION DEPENDENCIES

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

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

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

Test	File Operation	Mode	File Access Method
CE2102E	CREATE	OUT FILE	SEQUENTIAL IO
CE2102F	CREATE	INO $\bar{U}$ T FILE	DIRECT IO
CE2102J	CREATE	OUT F $\bar{I}$ LE	DIRECT IO
CE2102N	OPEN	IN F $\bar{I}$ LE	SEQUENTIAL 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 F $\bar{I}$ LE	TEXT IO
CE3102K	OPEN	OUT FILE	TEXT IO

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

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

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

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

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

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

IMPLEMENTATION DEPENDENCIES

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 28 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	B33001A	B36307A	B37004A	B45102A	B49003A
B49005A	B61012A	B62001B	B74304B	B74401F	B74401R
B91004A	B95004A	B95032A	B95069A	B95069B	BA1101B
BC2001D	BC3009A	BC3009C			

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AV0. The test contains a specification of a power-of-10 value as small for a fixed-point type. The AV0 ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

CHAPTER 3
PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

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

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

Larry N. McMahan
California Language Lab
19447 Pruneridge Avenue
Cupertino CA 95014
(408) 447-7234

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

Larry N. McMahan
California Language Lab
19447 Pruneridge Avenue
Cupertino CA 95014
(408) 447-7234

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

3.2 SUMMARY OF TEST RESULTS

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

PROCESSING INFORMATION

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

Total Number of Applicable Tests	3814
Total Number of Withdrawn Tests	74
Processed Inapplicable Tests	81
Non-Processed I/O Tests	0
Non-Processed Floating-Point Precision Tests	201
Total Number of Inapplicable Tests	282
Total Number of Tests for ACVC 1.11	4170

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

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 282 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 81 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

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

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

PROCESSING INFORMATION

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
-L	Produces an output listing.
-P 66	Sets the output page length to 66 lines.
-e 999	Sets the maximum number of errors to 999.
-W c, -SHOW=NONE	Suppresses printing of headers and of summary which would otherwise appear at the end of the compilation listing.
-u	Places information for generic instantiations into a separate symbol table.

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

APPENDIX A MACRO PARAMETERS

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

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

MACRO PARAMETERS

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

Macro Parameter	Macro Value
\$MAX_IN_LEN	255
\$ACC_SIZE	32
\$ALIGNMENT	2
\$COUNT_LAST	2147483647
\$DEFAULT_MEM_SIZE	2147483647
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	HP9000_300
\$DELTA_DOC	2#1.0#e-31
\$ENTRY_ADDRESS	TO_ADDRESS(16#40#)
\$ENTRY_ADDRESS1	TO_ADDRESS(16#80#)
\$ENTRY_ADDRESS2	TO_ADDRESS(16#100#)
\$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	1.0E308

MACRO PARAMETERS

```

$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        EXPORT_OBJECT

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006E1.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

```

MACRO PARAMETERS

\$NAME	SHORT_SHORT_INTEGER
\$NAME_LIST	HP9000_300
\$NAME_SPECIFICATION1	X2120A
\$NAME_SPECIFICATION2	X2120B
\$NAME_SPECIFICATION3	X3114A
\$NEG_BASED_INT	16#FF_FF_FF_FD#
\$NEW_MEM_SIZE	1048576
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	HP9000_300
\$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	0.020
\$VARIABLE_ADDRESS	TO_ADDRESS(16#0020#)
\$VARIABLE_ADDRESS1	TO_ADDRESS(16#0024#)
\$VARIABLE_ADDRESS2	TO_ADDRESS(16#0028#)
\$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.

The following options are recognized:

- a 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.

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

1. The Ada library specified when the bind was performed must be respecified.
2. The `.o` file generated by the binder must be specified before any HP-UX archive is specified (either explicitly or with `-l`).
3. 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)`). Only information required for debugging is saved; the source is not saved. (see `-D`.) By default, debug information is not stored.

Cause the binder to produce a debug information file for the program being bound so that the resulting program can be manipulated by the Ada debugger. The debug information file name will be the executable program file name with `.cui` appended. If the debug information file name would be truncated by the file system on which it would be created, an error will be reported.

Only sources compiled with the `-d` or `-D` option contribute information to the debug information file produced by the binder.

-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 "inline" 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. Also, if the value is too small, STORAGE_ERROR can be raised.

- n Cause the output file from the linker to be marked as shareable (see -N). 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.

The object file output by the binder is deleted if -c is not specified.

- 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)).

- 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 body (standard suffix is adabind)
c compiler body (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 "inline" 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, as well as 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.

-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 when comparing a compilation listing with a previous compilation listing of the same program, without the page headers causing mismatches. This option can not be specified in conjunction the -L option.

-C Only generate checks 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 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)). In addition to saving information required for debugging, an internal representation of the actual source is saved. This permits accurate source level debugging at the expense of a larger Ada library if the actual source file changes after it is compiled. (see **-d**.) By default, neither debug information nor source information is stored.

Cause the binder to produce a debug information file for the program being bound so that the resulting program can be manipulated by the Ada debugger. The debug information file name is the executable program file name with .cui appended. If the debug information file name would be truncated by the file system on which it would be created, an error will be reported.

Only sources compiled with the **-d** or **-D** option contribute information to the debug information file produced by the binder

-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.

-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 can not 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(UTIL)).

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. This is equivalent to +O eioE.

- 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 runtime 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.

-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. The argi are 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).

For example, the specification to pass the -r (preserve relocation information) option to the linker would be:

-W l,-r

For example, the following:

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

sends the options -m 10 -s 2 to the binder. 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 prepending the option with +; 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.

Binder Options

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

- W b,-b At execution time, interactive input blocks if data is not available. All tasks are suspended if input data is not available. This option is the default if the program contains no tasks (see -W b,-B).
- W b,-k Keep uncalled subprograms when binding. The default is to remove them.
- W b,-m,<nnn> Set the initial program stack size to <nnn> units of 1024 bytes (legal range 1..32767, default 10 units = 10 * 1024 bytes = 10240 bytes). The value is rounded up to the next multiple of 2.
- 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 20 milliseconds (<nnn> = 2).
- W b,-t,<nnn> Set task stack size of created tasks to <nnn> units of 1024 bytes.

Set the initial (and maximum) task stack size (legal range 1..32767, default 8 units = 8 * 1024 bytes = 8192 bytes).

- W b,-w Suppress warning messages.
- 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, interactive input does not block if data is not available. Only the task(s) doing interactive input are suspended if input data is not available. This option is the default if the program contains tasks (see -W b,-b).
- W b,-L Write a binder listing with warning/error diagnostics to standard error.
- W b,-T Suppress procedure traceback in response to runtime errors and unhandled exceptions.

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 may remove the lock.

Use ada.unlock(1) to unlock an Ada library and ada.funlock(1) to unlock an Ada family. However, unlocking

COMPILATION SYSTEM OPTIONS

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.

Options

ld recognizes the following options:

- d Forces definition of 'common' storage, i.e., assign addresses and sizes, even 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.

- lx Search a library libx.a, where x is one or more characters. 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.

- 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. (The 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. ld will not report undefined symbols.

- 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, that is, linking to enable the resulting object to 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**, and allows the newly linked segment to commence at the corresponding address. The default starting address is the old value of _end.

- L dir** Change the algorithm of searching for libx.a to look in dir before looking in the default places. 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 the text to be made 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.)
- X num** Define the initial size for the linker's global symbol table. Thus you can reduce link time for very large programs, i.e., those with very many external symbols.
- Z** Arrange for run-time dereferencing of null pointers to be permitted. (See in cc(1) the discussions of **-Z** and pointers.) (This is the complement of the **-z** option.)

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#0.000000000000001# range
-86400.0 .. 86400.0;

...

end STANDARD;

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 proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

Copyright © 1989, 1990 by Hewlett-Packard Company

HP 9000 Series 300 Computers
Reference Manual
for the
Ada Programming Language,
Appendix F for Ada 300



HP Part No. 98860-90011
Printed in U.S.A. October 1990

Second Edition
E1090

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	January 1990	92609A 04.35
Second Edition	October 1990	xxxxxx.xx.xx

Preface

This manual describes the implementation-dependent characteristics on the HP 9000 Series 300 System Ada Compiler. 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.
- Instruction and examples on calling external subprogram written in Precision Architecture RISC Assembly Language, HP C, HP FORTRAN 77, and HP Pascal.

-----> Note to reviewers: I'll check out the correct
titles and part numbers for the next review.

The following manuals provide additional information about the topics
indicated:

Ada

- *Reference Manual for the Ada Programming Language*
United States Department of Defense (1983)
MIL-STD-1815A
Order Number 008-000-00394-7
U.S. Government Printing Office, Washington DC 20402
(97055-90610)
- *Ada 300 User's Guide*
(98860-xxxxx)

Ada Tools

- *Ada 300 Tools Manual*
(98860-xxxxx)

HP-UX Operating System

- *HP-UX Reference*
(09000-90013)
- *HP-UX Concepts and Tutorials: Programming Environment*
(97089-90042)
- *HP-UX Concepts and Tutorials: Device I/O and User Interfacing*
(97089-90057)
- *HP-UX Portability Guide*
(98794-90047)

Assembly Language

- ■ *HP-UX Assembler Reference Manual and ADB Tutorial*
(98597-90020)
- *MC68020 32-Bit Microprocessor User's Manual*
(98615-90606)
- *MC68881 Floating-Point Coprocessor User's Manual*
(98615-90607)

C Language

- *HP C Reference Manual*
(92434-90001)
- *HP C Programmer's Guide*
(92434-90002)
- *HP C Quick Reference Guide*
(92434-90003)

HP FORTRAN 77

- *HP FORTRAN 77/HP-UX Reference Manual*
(92430-90005)
- *HP FORTRAN 77/HP-UX Programmer's Reference*
(92430-90004)
- *HP FORTRAN 77/HP-UX Quick Reference Guide*
(92430-90008)

HP Pascal

- *HP Pascal Reference Manual*
(31502-90001)
- *HP Pascal Programmer's Guide*
(31502-90002)
- *HP Pascal Quick Reference Guide*
(31502-90003)

NFS® Systems

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

NFS® is a trademark of SUN Microsystems, Inc.

Conventions

This manual uses the conventions described in this section for syntax and example programs.

Notation	Description
lowercase nonbold	Represents literals that must be entered exactly as shown.
lowercase boldface	Represents Ada language reserved words.
<i>italics</i>	Represents substitutable arguments names, program names, or user-designated strings.
[]	Specifies that an element inside brackets is optional.
...	A horizontal ellipsis in a syntax statement indicates that a previous element can be repeated. For example: <i>identifier</i> [, <i>identifier</i>] ... The above example will match any list of identifiers each separated by a comma.
	When several elements are separated vertically by bars in a syntax statement, you must select one of those elements. For example: A B C You must select A, B, or C

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	1-7
F 1.1.6 Pragma EXTERNAL_NAME	1-8
F 1.1.7 Example of EXPORT and EXTERNAL_NAME . . .	1-9
F 1.2 Using Text Processing Tools	1-10
F 1.2.1 Pragma INDENT	1-10
F 1.2.2 Pragma LIST	1-11
F 1.2.3 Pragma PAGE	1-11
F 1.3 Affecting the Layout of Array and Record Types	1-12
F 1.3.1 Pragma PACK	1-12
F 1.3.2 Pragma IMPROVE	1-12
F 1.4 Generating Code	1-13
F 1.4.1 Pragma ELABORATE	1-13
F 1.4.2 Pragma INLINE	1-14
F 1.4.3 Pragma SUPPRESS	1-15
F 1.5 Affecting Run Time Behavior	1-16
F 1.5.1 Pragma PRIORITY	1-16
F 1.5.2 Pragma SHARED	1-17
F 1.6 Pragmas Not Implemented	1-18

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	3-4
4. F 4. Type Representation	
F 4.1 Enumeration Types	4-2
F 4.1.1 Internal Codes of Enumeration Literals	4-3
F 4.1.2 Minimum Size of an Enumeration Type or Subtype	4-5
F 4.1.3 Size of an Enumeration Type	4-6
F 4.1.4 Alignment of an Enumeration Type	4-6
F 4.2 Integer Types	4-7
F 4.2.1 Predefined Integer Types	4-7
F 4.2.2 Internal Codes of Integer Values	4-7
F 4.2.3 Minimum Size of an Integer Type or Subtype	4-8
F 4.2.4 Size of an Integer Type	4-10
F 4.2.5 Alignment of an Integer Type	4-12
F 4.3 Floating Point Types	4-13
F 4.3.1 Predefined Floating Point Types	4-13
F 4.3.2 Internal Codes of Floating Point Values	4-13
F 4.3.3 Minimum Size of a Floating Point Type or Subtype	4-16
F 4.3.4 Size of a Floating Point Type	4-16
F 4.3.5 Alignment of a Floating Point Type	4-16
F 4.4 Fixed Point Types	4-17
F 4.4.1 Predefined Fixed Point Types	4-17
F 4.4.2 Internal Codes of Fixed Point Values	4-18
F 4.4.3 Small of a Fixed Point Type	4-18
F 4.4.4 Minimum Size of a Fixed Point Type or Subtype	4-19
F 4.4.5 Size of a Fixed Point Type	4-21
F 4.4.6 Alignment of a Fixed Point Type	4-22
F 4.5 Access Types	4-23
F 4.5.1 Internal Codes of Access Values	4-23
F 4.5.2 Collection Size for Access Types	4-23
F 4.5.3 Minimum Size of an Access Type or Subtype	4-25
F 4.5.4 Size of an Access Type	4-25

F 4.5.5 Alignment of an Access Type	4-25
F 4.6 Task Types	4-26
F 4.6.1 Internal Codes of Task Values	4-26
F 4.6.2 Storage for a Task Activation	4-26
F 4.6.3 Minimum Size of a Task Stack	4-28
F 4.6.4 Limitation on Length Clause for Derived Task Types	4-28
F 4.6.5 Minimum Size of a Task Type or Subtype	4-28
F 4.6.6 Size of a Task Type	4-28
F 4.6.7 Alignment of a Task Type	4-28
F 4.7 Array Types	4-29
F 4.7.1 Layout of an Array	4-29
F 4.7.2 Array component size and pragma PACK	4-29
F 4.7.3 Array Gap Size and Pragma PACK	4-30
F 4.7.4 Size of an Array Type or Subtype	4-32
F 4.7.5 Alignment of an Array Type	4-33
F 4.8 Record Types	4-34
F 4.8.1 Layout of a Record	4-34
F 4.8.2 Bit Ordering in a Component Clause	4-36
F 4.8.3 Value used for SYSTEM.STORAGE_UNIT	4-37
F 4.8.4 Compiler-Chosen Record Layout	4-37
F 4.8.5 Change in Representation	4-38
F 4.8.6 Implicit Components	4-38
F 4.8.7 Indirect Components	4-40
F 4.8.8 Dynamic Components	4-41
F 4.8.9 Representation of the Offset of an Indirect Component	4-44
F 4.8.10 The Implicit Component RECORD_SIZE	4-45
F 4.8.11 The Implicit Component VARIANT_INDEX	4-45
F 4.8.12 The Implicit Component ARRAY_DESCRIPTOR	4-47
F 4.8.13 The Implicit Component RECORD_DESCRIPTOR	4-47
F 4.8.14 Suppression of Implicit Components	4-48
F 4.8.15 Size of a Record Type or Subtype	4-49
F 4.8.16 Size of an Object of a Record Type	4-50
F 4.8.17 Alignment of a Record Subtype	4-50
F 4.9 Data Allocation	4-51
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	4-52

F 4.9.2.3 Program Termination	4-53
F 4.9.3 Dynamic Memory Management	4-53
F 4.9.3 Collections of Objects	4-53
F 4.9.3.2 Global Dynamic Objects	4-54
F 4.9.3.3 Local Objects	4-55
F 4.9.3.4 Temporary Objects	4-55
F 4.9.3.5 Reclaiming Heap Storage	4-56
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 Characteristics	
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 Files	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	8-16
F 8.1.10 Ada I/O System Dependencies	8-17

F 8.2 The FORM Parameter	8-19
F 8.2.1 An Overview of FORM Attributes	8-19
F 8.2.2 The Format of FORM Parameters	8-19
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 - 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	8-28
F 8.2.8 The FORM Parameter - Terminal Input	8-29
F 8.2.9 The FORM Parameter Attribute - File Structuring	8-30
F 8.2.9.1 The Structure of TEXT_IO Files	8-30
F 8.2.9.2 The Structure of DIRECT_IO and SEQUENTIAL_IO Files	8-32
 9. F 9. The Ada/300 Development System and HP-UX Signals	
F 9.1 HP-UX Signals Reserved by the Ada/300 Runtime	9-2
F 9.2 Using HP-UX Signals in External Interfaced Subprograms	9-5
F 9.3 HP-UX Signals Used for Ada/300 Exception Handling	9-6
F 9.4 HP-UX Signals Used for Ada/300 Task Management	9-8
F 9.5 HP-UX Signals Used for Ada/300 Delay Timing	9-9
F 9.6 HP-UX Signals Used for Ada/300 Program Termination	9-10
F 9.7 HP-UX Signals Used for Ada/300 Interrupt Entries	9-12
F 9.8 Protecting Interfaced Code from Ada/300's Asynchronous Signals	9-13
F 9.9 Programming in Ada/300 With HP-UX Signals	9-13
 10. F 10. Limitations	
F 10.1 Compiler Limitations	10-1
F 10.2 Ada Development Environment Limitations	10-4
F 10.3 Limitations Affecting User-Written Ada Applications	10-5
F 10.3.1 Restrictions Affecting Opening or Creating Files	10-5
F 10.3.1.1 Restrictions on Path and Component Sizes	10-5
F 10.3.1.2 Conditions that Raise NAME_ERROR	10-5
F 10.3.2 Restrictions on TEXT_IO.FORM	10-6
F 10.3.3 Restrictions on the Small of a Fixed Point Type	10-6

(Delete section titled "Record Type Change of Representation")	10-6
F 10.3.4 Record Type Alignment Clause	10-6
F 10.3.5 Pragma INTERFACE on Library Level Subprograms	10-7
11. F 11. Calling External Subprograms From Ada	
F 11.1 General Considerations in Passing Ada Types	11-5
F 11.1.1 Scalar Types	11-5
F 11.1.1.1 Integer Types	11-6
F 11.1.1.2 Enumeration Types	11-6
F 11.1.1.3 Boolean Types	11-7
F 11.1.1.4 Character Types	11-7
F 11.1.1.5 Real Types	11-8
F 11.1.2 Access Types	11-9
F 11.1.3 Array Types	11-11
F 11.1.4 Record Types	11-12
F 11.1.5 Task Types	11-13
F 11.2 Calling Assembly Language Subprograms	11-14
F 11.2.1 Scalar Types and Assembly Language Subprograms	11-15
F 11.2.1.1 Integer Types and Assembly Language Subprograms	11-15
F 11.2.1.2 Enumeration Types and Assembly Language Subprograms	11-15
F 11.2.1.3 Boolean Types and Assembly Language Subprograms	11-15
F 11.2.1.4 Character Types and Assembly Language Subprograms	11-15
F 11.2.1.5 Real Types and Assembly Language Subprograms	11-15
F 11.2.2 Access Types and Assembly Language Subprograms	11-16
F 11.2.3 Array Types and Assembly Language Subprograms	11-16
F 11.2.4 Record Types and Assembly Language Subprograms	11-16
F 11.3 Calling HP C Subprograms	11-17
F 11.3.1 Scalar Types and HP C Subprograms	11-18
F 11.3.1.1 Integer Types and HP C Subprograms	11-19
F 11.3.1.2 Enumeration Types and HP C Subprograms	11-20
F 11.3.1.3 Boolean Types and HP C Subprograms	11-21
F 11.3.1.4 Character Types and HP C Subprograms	11-22
F 11.3.1.5 Real Types and HP C Subprograms	11-23

F 11.3.2 Access Types and HP C Subprograms	11-24
F 11.3.3 Array Types and HP C Subprograms	11-25
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	11-35
F 11.4.1.3 Boolean Types and HP FORTRAN 77 Subprograms	11-36
F 11.4.1.4 Character Types and HP FORTRAN 77 Subprograms	11-38
F 11.4.1.5 Real Types and HP FORTRAN 77 Subprograms	11-39
F 11.4.2 Access Types and HP FORTRAN 77 Subprograms	11-40
F 11.4.3 Array Types and HP FORTRAN 77 Subprograms	11-41
F 11.4.4 String Types and HP FORTRAN 77 Subprograms	11-44
F 11.4.5 Record Types and HP FORTRAN 77 Subprograms	11-48
F 11.4.6 Other FORTRAN Types	11-49
F 11.5 Calling HP Pascal Language Subprograms	11-50
F 11.5.1 Scalar Types and HP Pascal Subprograms	11-52
F 11.5.1.1 Integer Types and HP Pascal Subprograms	11-52
F 11.5.1.2 Enumeration Types and HP Pascal Subprograms	11-54
F 11.5.1.3 Boolean Types and HP Pascal Subprograms	11-54
F 11.5.1.4 Character Types and HP Pascal Subprograms	11-54
F 11.5.1.5 Real Types and HP Pascal Subprograms	11-55
F 11.5.2 Access Types and HP Pascal Subprograms	11-56
F 11.5.3 Array Types and HP Pascal Subprograms	11-56
F 11.5.4 String Types and HP Pascal Subprograms	11-58
F 11.5.5 Record Types and HP Pascal Subprograms	11-62
F 11.6 Summary	11-63
F 11.7 Potential Problems Using Interfaced Subprograms	11-66
F 11.8 Input-Output From Interfaced Subprograms	11-70
F 11.8.1 Files Opened by Ada and Interfaced Subprograms	11-70
F 11.8.2 Preconnected I/O and Interfaced Subprograms	11-70
F 11.8.3 Interactive I/O and Interfaced Subprograms	11-71

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	12-8
F 12.7 Disassociating an Ada Handler from an HP-UX Signal . .	12-10
F 12.8 Determining How Many Handlers are Installed	12-10
F 12.9 When Ada Signal Handlers Will Not Be Called	12-10
F 12.10 Address Clauses for Entries	12-11
F 12.11 Example of Interrupt Entries	12-11
F 12.12 Specification of the package INTERRUPT_MANAGER	12-12
F 12.13 Ada Runtime Routine Descriptions	12-17

Index

Figures

4-1. Layout of an Array	4-29
4-2. Record layout with an Indirect Component	4-40
4-3. Example of a Data Layout	4-43
11-1. Passing Access Types to Interfaced Subprograms	11-10

Tables

1-1. Ada/300 Pragmas	1-1
4-1. Methods to Control Layout and Size of Data Objects	4-1
4-2. Alignment and Pragma PACK	4-33
8-1. Standard Predefined I/O Packages	8-2
8-2. User Access Categories	8-21
8-3. File Access Rights	8-22
8-4. File Sharing Attribute Modes	8-25
8-5. Text File Terminators	8-31
8-6. Structuring Binary Files with the FORM Parameter	8-33
8-6. Structuring Binary Files with the FORM Parameter (Continued)	8-34
9-1. Ada/300 Signals	9-2
11-1. Ada Types and Parameter Passing Modes	11-2
11-2. Ada/300 versus HP C Integer Correspondence	11-19
11-3. Ada/300 versus HP FORTRAN 77 Integer Correspondence . .	11-33
11-4. Ada/300 versus HP Pascal Integer Correspondence	11-52
11-6. Modes for Passing Parameters to Interfaced Subprograms . .	11-63
11-7. Types Returned as External Function Subprogram Results . .	11-64
11-8. Parameter Passing in the Series 300 Implementation	11-65
12-1. Heap Management Routines	12-17
12-2. Collection Management (no STORAGE_SIZE representation clause)	12-17
12-3. Collection Management (collections with a STORAGE_SIZE representation clause)	12-18
12-4. Tasking Routines	12-19
12-4. Tasking Routines (Continued)	12-20
12-4. Tasking Routines (Continued)	12-21
12-5. Attributes Routines	12-22
12-6. Attributes for Tasks Routines	12-22
12-7. Support for Enumeration Representation Clauses Routines . .	12-23

F 1. Implementation Supported Pragmas

This section describes the predefined language pragmas and the Ada/300 implementation-specific pragmas. Table F-1 lists these pragmas.

Table 1-1. Ada/300 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 data objects to be referenced by a non-Ada external subprogram.	EXPORT EXTERNAL_NAME

Section F 1.6 lists predefined pragmas not implemented in Ada/300.

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/300 supports subprograms written in these languages:

- ■ HP 68K Assembly Language
 - HP C
 - HP Pascal
- ■ HP FORTRAN 77 for HP 9000 Series 300 computers

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

In addition, 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
<i>Language_name</i>	is one of ASSEMBLER, C, PASCAL, or FORTRAN.
<i>Ada_subprogram_name</i>	is the name used within the Ada program when referring to the interfaced external subprogram.

It is not possible to supply a pragma INTERFACE to a library-level subprogram. Any subprogram that a pragma INTERFACE applies to must be contained within an Ada compilation unit, usually a package.

F 1.1.2 Pragma INTERFACE_NAME

- Ada/300 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
<i>Ada_subprogram_name</i>	is the name when referring to the interfaced external subprogram.
<i>External_subprogram_name</i>	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 also prefixes the external subprogram name with one underscore character (`_`) and truncates this name to 254 characters if necessary. This modification conforms to the naming conventions used by the HP linker (1d(1) - Link Editor) on the HP 9000 Series 300 Computer System.

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 declaration of the corresponding pragma `INTERFACE` and must be within the same declarative part, although it need *not* immediately follow that declaration.

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 will generate 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, which is named Do_Sample by both the HP C compiler and the HP Ada compiler.

The compiler prefixes the external subprogram name with one underscore character (_) and truncates this name to 254 characters if necessary. This modification conforms to the naming conventions used by the HP linker (ld(1)) - Link Editor) on the HP 9000 Series 300 Computer System.

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 two 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 (for binding and linking with the same command) or an object library (for binding and linking separately) 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 “_Ada” 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 searches these two libraries. If your library function is located in any of the other standard libraries, you must provide the `-lx` option to the binder to instruct the binder to search the library.

See section F 11 for additional information on using pragma `INTERFACE` and pragma `INTERFACE_NAME`.

F 1.1.5 Pragma EXPORT

The pragma EXPORT allows for a data object 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. The default name for the externally visible symbol is the name of the Ada object in all lowercase letters with an underscore character (_) attached to the beginning. The pragma EXTERNAL_NAME (described in section F 1.1.6) can be used to change this default.

Syntax

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

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

The pragma EXPORT must occur in a declarative part and applies only to objects declared in the same declarative part; that is, generic instantiated objects or renamed objects are excluded.

The pragma EXPORT can only be used for objects with direct allocation mode that are declared in a library package. Objects are allocated with indirect allocation mode if they are dynamic or have a significant size. For more details, see section F 4.9, "Data Allocation."

F 1.1.6 Pragma EXTERNAL_NAME

The pragma EXTERNAL_NAME is used to supply an alternative name for a global Ada data object 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.

Syntax

```
pragma EXTERNAL_NAME (Ada_object_name,  
                      "External_object_name");
```

Parameter	Description
<i>Ada_object_name</i>	is the name when referring to the interfaced external subprogram.
<i>External_object_name</i>	is the external name used outside the Ada program.

The compiler also prefixes the external object name with one underscore character (_) and truncates this name to 254 characters if necessary. This modification conforms to the naming conventions used by the HP linker (ld(1) - Link Editor) on the HP 9000 Series 300 Computer System.

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;

private

    pragma EXPORT(ASSEMBLER, MY_INT);
    pragma EXPORT(C, MY_CHAR);

    pragma EXTERNAL_NAME(MY_INT, "Int_from_Ada");
    pragma EXTERNAL_NAME(MY_CHAR, "Char_from_Ada");

end ADA_GLOBALS;
```

This example defines two Ada data objects that are known in Ada code as MY_INT and MY_CHAR. The externally visible symbols for the two data objects are _Int_from_Ada and _Char_from_Ada, respectively.

The compiler prefixes the external symbol name with one underscore and truncates the name to 254 characters if necessary. This modification conforms to the naming conventions used by the HP Linker (ld(1) - Link Editor) on the HP 9000 Series 300 Computer System.

By using the pragma EXTERNAL_NAME, the names of the external symbols are explicitly identified. If pragma EXTERNAL_NAME had not been used, the two external names would be _my_int and _my_char.

F 1.2 Using Text Processing Tools

The pragma `INDENT` is a formatting command that affects the HP supplied reformatter, `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/300 provides the implementation-defined pragma `INDENT` to assist in reformatting 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	<code>ada.format</code> does not modify the source lines after the pragma.
ON	<code>ada.format</code> 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/300. 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 dependancy 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/300, the range of the subtype `PRIORITY` is 1 to 127. This 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 kernel. 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.

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.

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".

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 subprogram will return different values depending upon the elaboration time of the subprogram. In particular, the value returned by the attribute 'ADDRESS changes after the elaboration of the subprogram body. This can be a problem when ADDRESS is applied to a subprogram in a package specification. Therefore, the attribute *subprogram* ADDRESS should not be used in a package specification and instead should be used only in the package body after the body of the subprogram.

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
<i>Language_name</i>	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.
<i>external_symbol_name</i>	Specifies the name of an external data object. This parameter is a static Ada string constant. The result is a value of the type <code>SYSTEM.ADDRESS</code> that can be used to denote this object in an address clause (see section F 6, "Address Clauses" for details.)

The following example shows how `SYSTEM.ADDRESS'IMPORT` is used in an address clause to provide access to a global object defined in C.

The C declaration:

```
extern int errno;
```

The Ada declaration:

```
package ERRNO_ACCESS is
```

```
    ERRNO: INTEGER;
```

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

```
end ERRNO_ACCESS;
```

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_300);

    SYSTEM_NAME  : constant NAME := HP9000_300;

    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 := 0.020;  -- 20 milliseconds

subtype PRIORITY is INTEGER range 1 .. 16;

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

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_SEGMENT (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
-----
-- This function returns true if the two addresses have the same
-- segment value. This is always true on the HP 9000/300.
-----

ADDRESS_ERROR : exception;

```

```

-----
-- This exception is raised by "<", "<=", ">", ">=", "-" if the
-- two addresses do not have the same segment value. This
-- exception is never raised on the HP 9000/300. The exception
-- CONSTRAINT_ERROR can be raised by "+" and "-".
-----

```

```

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 9000/300 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. The exception ADDRESS_ERROR is never raised
-- on the HP 9000/300.
-----

```

```

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.
-----
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.
-----

private

    --- private part of package SYSTEM

end SYSTEM;
```

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);
for BOOLEAN use (FALSE => 2#0000_0000#,
                 TRUE  => 2#1111_1111#);
```

```
-- 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  $-(2^{**7}) \dots (2^{**7})-1$ 
-- 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
-2#1.1111_1111_1111_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_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,

  ' ', '!', '"', '#', '$', '%', '&', ' ',

```

```

'(', ')', '*', '+', ',', '-', '.', '/',
'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;
  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;
```

F 4. Type Representation

This chapter explains how data objects are represented and allocated by the HP Ada compiler for the HP 9000 Series 300 Computer System 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/300 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 4.1.2 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  $(2^{**1})-1 \leq 2 \leq (2^{**2})-1$ 
```

```
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 300 Computer System 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/300 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; otherwise it is word aligned. Word alignment is 2-byte or 16-bit aligned.

F 4.2 Integer Types

F 4.2.1 Predefined Integer Types

The HP 9000 Series 300 Computer System 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
$m \geq 0$	$M \leq (2^{**}L) - 1$	Unsigned
$m < 0$	$-(2^{**}(L-1)) \leq m$ and $M \leq (2^{**}(L-1))-1$	Signed two's complement

Example

```
type 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_WORD is range 0 .. (2**16)-1;
for UNSIGNED_WORD'SIZE use 16;
-- Type UNSIGNED_WORD 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 that 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/300 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; otherwise, it is word aligned. Word alignment is 2-byte or 16-bit aligned.

F 4.3 Floating Point Types

F 4.3.1 Predefined Floating Point Types

The HP 9000 Series 300 Computer System 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 is always word aligned. Word alignment is 2-byte or 16-bit aligned.

F 4.4 Fixed Point Types

F 4.4.1 Predefined Fixed Point Types

To implement fixed point types, the HP 9000 Series 300 Computer System provides a set of three anonymous predefined fixed point types of this form:

```
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/300 compiler requires that the value assigned to `'SMALL` is always a power of two. Ada/300 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:

$$\begin{aligned} s &\leq m < M \leq S \\ m - T'BASE'SMALL &\leq s \\ M + T'BASE'SMALL &\geq S \\ M &= T'BASE'LARGE \\ i &= m / T'BASE'SMALL \\ I &= M / T'BASE'SMALL \end{aligned}$$

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/300 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; otherwise it is word-aligned. Word alignment is 2-byte or 16-bit aligned.

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; -- Subsitute 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'SORAGE_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 always word aligned. Word alignment is 2-byte or 16-bit 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.

F 4.6.2 Storage for a Task Activation

The value returned by the attribute `'STORAGE_SIZE` is determined as follows:

- 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 `-Wb, -t, nnn` to set the task storage size, the attribute `STORAGE_SIZE` returns $nnn \times 1024$.
- 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 a type (see *Ada RM*, section 13.2(10)). This space includes both the task stack space and a private data section of approximately 3600 bytes. The private data section contains the Task Control Block that 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 or else 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.

The following example 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 use 3600 + (4 * 1024); -- Allocates a 4K stack.
```

F 4.6.3 Minimum Size of a Task Stack

The task object will use 150 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 400 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

This 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 always word aligned. Word alignment is 2-byte or 16-bit 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.

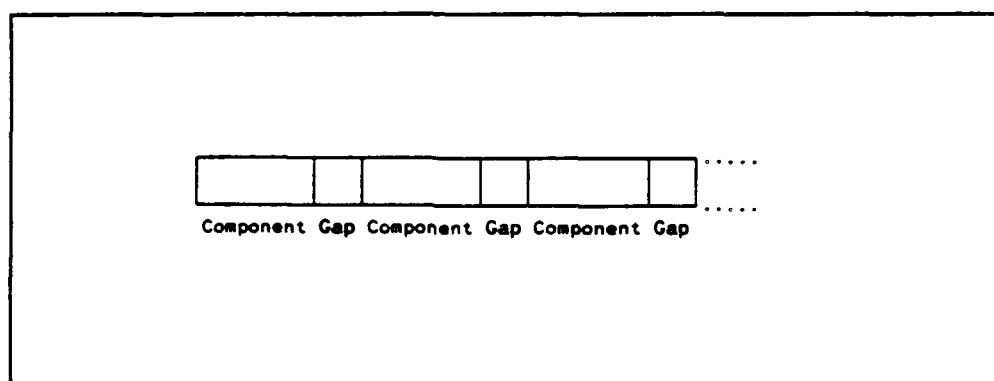


Figure 4-1. Layout of an Array

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 one byte is inserted after each array component in
-- order to respect the alignment of type R.
-- The size of array type A is 480 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/300 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 word aligned if the component type is word aligned. Otherwise, it is byte aligned.

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.

Table 4-2. Alignment and Pragma PACK

Component Alignment	Displacement is on an Even Byte	Displacement is on an Odd Byte	Displacement is on a Bit
word	word	byte	bit
byte	byte	byte	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;
```

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 "Implicit Components" in section 4.8. In the implementation for the HP 9000 Series 300 Computer System, 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 TRACE_KIND is (None, Change_of_Flow,
                    Any_Instruction, Reserved);

type SYSTEM_STATE is (User, Supervisor);

type SUPERVISOR_STATE is (Interrupt, Master);

type INTERRUPT_PRIORITY is range 0..7;

type CONDITION_CODE is new BOOLEAN;
for CONDITION_CODE'SIZE use 1;

SYSTEM_BYTE : constant := 0;
USER_BYTE   : constant := 1;

type STATUS_REGISTER is
  record
    T : TRACE_KIND;
    S : SYSTEM_STATE;
    M : SUPERVISOR_STATE;
    I : INTERRUPT_PRIORITY;
    X : CONDITION_CODE;
    N : CONDITION_CODE;
    Z : CONDITION_CODE;
    V : CONDITION_CODE;
    C : CONDITION_CODE;
  end record;

-- This type can be used to map the status register of
-- the MC68020 microprocessor.

for STATUS_REGISTER use
  record at mod 2;
    T at SYSTEM_BYTE range 0..1;
    S at SYSTEM_BYTE range 2..2;
    M at SYSTEM_BYTE range 3..3;
    I at SYSTEM_BYTE range 5..7;
```

```

        X at USER_BYTE    range 3..3;
        N at USER_BYTE    range 4..4;
        Z at USER_BYTE    range 5..5;
        V at USER_BYTE    range 6..6;
        C at USER_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 2-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 300 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. Notice that this ordering is different from the bit ordering used in the *Motorola MC68020 User's Guide*, which numbers the bits in the reverse order.

F 4.8.3 Value used for SYSTEM.STORAGE_UNIT

The smallest directly addressable unit on the HP 9000 Series 300 Computer System 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 300 Computer System 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.

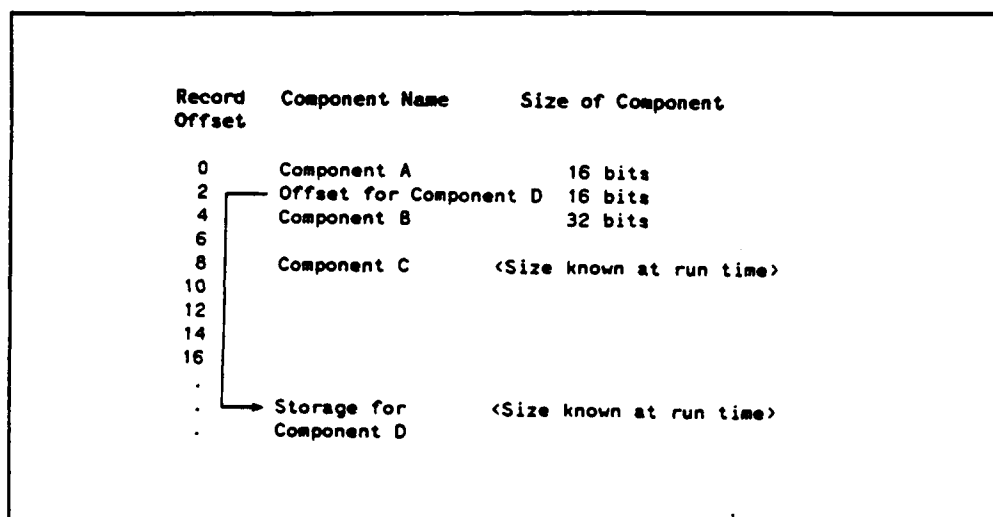


Figure 4-2. Record layout with an Indirect Component

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;

```

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.

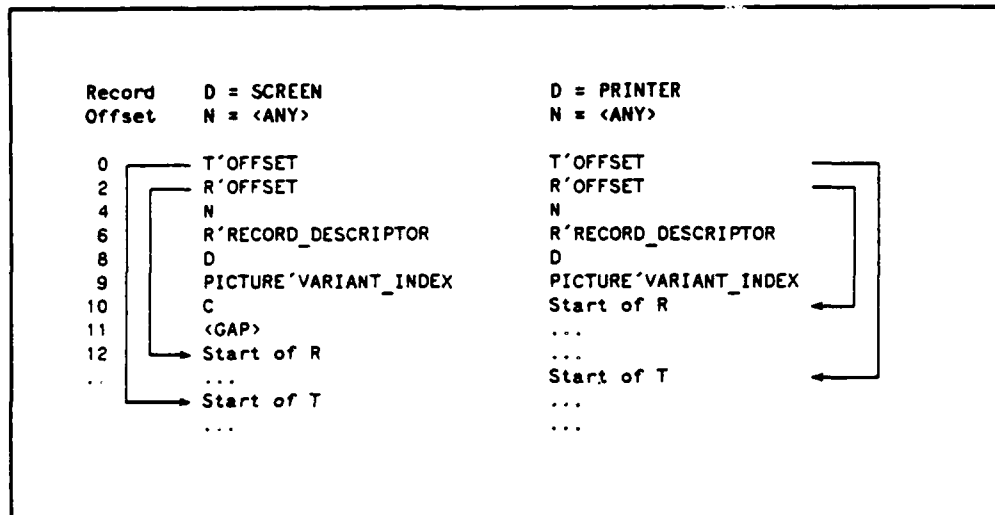


Figure 4-3. Example of a Data Layout

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 300 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.

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 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`.

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/300 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.

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 8192 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 8192 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 if it contains a component whose type is word-aligned and whose offset in the record is also word aligned. Otherwise the record type is byte aligned. 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 aligned or word aligned. Word alignment is 2-byte alignment, or 16-bit alignment. Ada/300 does not support alignments larger than a 2-byte alignment.

F 4.9 Data Allocation

Data objects are allocated into one of these:

- A stack frame.
- The global data area.
- The heap.
- No storage allocated.

A stack frame is used for objects declared in a subprogram or task body, or in 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 a library level packages, either in the specification or in the body. The global data area contains the data objects that can be 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 `.bss` segment of the UNIX object file (see *a.out_300(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 explicit deallocator operation is performed. The heap is allocated using the *sbrk(2)* system call and thus heap objects will reside in the `.data` segment of the UNIX object file (see *a.out_300(4)*).

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 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 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 from reclaiming the temporaries associated with any statements.

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 any and all objects of this access type has expired and thus their 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 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 6-byte descriptor if the collection is global (that is, is declared within a library level package) or a 14-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 6- or 14-byte descriptor.

When a collection size is specified using a length clause for an access type (that is, `T'SORAGE_SIZE use <nnn>`), the heap manager will allocate 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. When space in the collection is exhausted, the exception `STORAGE_ERROR` will be raised.

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 that is declared within a library package (specification or body including nested packages and blocks, but not subprograms or tasks.)

The compiler also will consider 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 will consider 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 above.

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, however, for an unconstrained record if the object discriminant or discriminants can be changed during the lifetime of the object, and this discriminant change has potentially a 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 section 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 happens only when the memory is not known at compiler time, or is big enough (more than 128 bytes) to be considered as 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 an access type's collection, all storage allocated is returned upon exiting the scope in which type access type was declared. Reclaiming occurs whether the exit is normal or abnormal (that is, due to exception propagation.)
- A task's storage (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.

F 5. Names for Predefined Library Units

The names listed below are used by the HP Ada Development System. Do not use any of these names for your library-level Ada units.

Not available for use:

ALSYS_ADA_RUNTIME
HIT

Names that are available, but should be avoided if you want access to packages ■
that are provided by Hewlett-Packard:

MATH_LIB
MATH_EXCEPTIONS ■
SYSTEM_ENVIRONMENT

The above three packages are documented in the *Ada 300 User's Guide*. ■

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 8192 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/300 compiler.

F 6.3 Constants

Address clauses for constants are not implemented in the current version of the Ada/300 compiler.

F 6.4 Packages

Address clauses for packages are not implemented in the current version of the Ada/300 compiler.

F 6.5 Tasks

Address clauses for tasks are not implemented in the current version of the Ada/300 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#;
```

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.)

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 discriminants 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.

Warning

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).

F 8. Implementation-Dependent Input-Output Characteristics

This chapter describes the I/O characteristics of Ada on the HP 9000 Series 300 computer. Ada handles I/O with packages, which are discussed in section F 8.1. File types are described in section F 8.1.3 and the FORM parameter is discussed in section F 8.2.

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/300. 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 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*, section 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 <i>Ada RM</i> Location
SEQUENTIAL_IO	A generic package for sequential files of a single element type. (<i>Ada RM</i> , section 14.2.3)
DIRECT_IO	A generic package for direct (random) access files of a single element type. (<i>Ada RM</i> , section 14.2.5)
TEXT_IO	A non-generic package for ASCII text files. (<i>Ada RM</i> , section 14.3.10)
IO_EXCEPTIONS	A package that defines the exceptions needed by the above three packages. (<i>Ada RM</i> , 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).

Warning

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/300 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/300 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 default of `FORM` is the null string.

The next three subsections describe the Ada/300 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 8.2.9.2 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 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 8.2.9.2 for more information on file structure.

If the file is created with the default `FORM` parameter attributes (see section F 8.2), 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 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 section F 8.2.9.1 in this manual 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 8.2.9.1 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 and F 8.2.8 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 on the `FORM` parameter 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/300 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/300. 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 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, 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

Note: Removed RFA information.

This section assumes knowledge of networks. It describes Ada program I/O involving Local Area Network (LAN) services available on the Series 300 computers: NFS † systems (remote file access using the NFS network services software.)

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 the remote system (*cezanne*) 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 300, 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/300 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 NFS systems. 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.

Warning



It is during the Ada/300 **OPEN** and **CREATE** routines that the name that uniquely identifies the external file is determined for later use by the **NAME** function. If it was 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 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.

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 composed from a list of attributes that specify the following:

- File protection
- File buffering
- File sharing
- Appending
- Blocking
- Terminal input
- File structuring
- Terminal Input

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: "inpt_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 => "inpt_file", -- file name is "inpt_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 zero, which means no buffering. 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 300.

An example of the use of 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;
```

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 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
<code>NOT_SHARED</code>	Indicates exclusive access. No other logical file can be associated with the external file.
<code>SHARED=> READERS</code>	Only logical files of mode <code>IN</code> can be associated with the external file.
<code>SHARED=> SINGLE_WRITER</code>	Only logical files of mode <code>IN</code> and at most one file with mode <code>OUT</code> can be associated with the external file.
<code>SHARED=> ANY</code>	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 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 - 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

F 8.2.7.1 Blocking

If the blocking attribute is set, the read or write operation will cause the HP-UX process to block until the read or write request is satisfied. This means that all Ada tasks are blocked from running until the data transfer is complete.

The default for this attribute depends on the actual program. The default is **BLOCKING** for programs without any task declarations and is **NON_BLOCKING** for a program containing tasks. This allows tasking programs to take advantage of their parallelism in the presence of certain I/O requests. There is no advantage in specifying **NON_BLOCKING** for a non-tasking program because *the program* must wait for the I/O request to complete before continuing its sequential execution.

F 8.2.7.2 Non-Blocking

The **NON_BLOCKING** attribute specifies that when a read request cannot be immediately satisfied, the Ada runtime should schedule another task to run and retry the read operation later. This attribute is currently only applicable for terminal devices and pipes. In the case of a pipe, a write request may also cause the current task to be rescheduled, and another task will run while the pipe buffer is full. This attribute sets the **O_NONDELAY** flag in the HP-UX file descriptor and allows the HP-UX process to continue running if there is no data available to be read from the terminal or pipe.

F 8.2.8 The FORM Parameter - Terminal Input

The terminal input attribute takes one of two alternative forms:

`TERMINAL_INPUT => LINES,`

`TERMINAL_INPUT => CHARACTERS,`

Terminal input is normally processed in units of one line at a time. A process attempting to read from the terminal as an external file is suspended until a complete line has been typed. At that time, the outstanding read call (and possibly also later calls) is satisfied.

The `LINES` option specifies a line-at-a-time data transfer, which is the default case.

The `CHARACTERS` option means that data transfers character by character, and so a complete line does not have to be entered before the read request can be satisfied. For this option, the `BUFFER_SIZE` must be zero.

When the `CHARACTERS` option is specified, the `ICANON` bit is cleared in the *c_lflag* component of the HP-UX *termio* structure. This bit changes the line discipline for the terminal device. Be aware that the line discipline statue is not maintained on a per file basis. Changing the line discipline for one terminal file does effect other terminal files that are actually associated with the same physical terminal device. See *termio*(7) for additional information.

The `TERMINAL_INPUT` attribute is only applicable to Ada files objects other than `STANDARD_INPUT`. The Ada runtime system uses the default `TERMINAL_INPUT` of `LINES` for the Ada file object `STANDARD_INPUT`. The file name `/dev/tty` can be used with the appropriate `FORM` parameter to achieve single character I/O on the program's controlling terminal (which is the same terminal that `STANDARD_INPUT` is associated with if `STANDARD_INPUT` has not been redirected.)

F 8.2.9 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 8.2.9.1 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.

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 for more information about terminators in text files.

F 8.2.9.2 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 <code>RECORD_UNIT</code> attribute is illegal.	If the <code>RECORD_SIZE</code> attribute is omitted, no <code>UNUSED_PART</code> is implemented. The default <code>RECORD_SIZE</code> is the object's size. If present, the <code>RECORD_SIZE</code> attribute must specify a record size greater than or equal to the object's size. Otherwise, the exception <code>USE_ERROR</code> is raised.

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 is:

$$((\text{OBJECT'SIZE} + \text{SYSTEM.STORAGE\_UNIT} - 1) / \text{SYSTEM.STORAGE\_UNIT}) + 8$$

F 9. The Ada/300 Development System and HP-UX Signals

The Ada/300 runtime on the HP 9000 Series 300 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/300 Runtime

Table 9-1 lists the HP-UX signals reserved and used by the Ada runtime.

Table 9-1. Ada/300 Signals

Signal	Description
SIGALRM	Used for <i>delay</i> and optionally for time-slicing.
SIGVTALRM	Optionally used for time-slicing (default time-slicing signal).
SIGPROF	Optionally used for time-slicing.
SIGILL	Causes the exceptions <code>CONSTRAINT_ERROR</code> or <code>NUMERIC_ERROR</code> .
SIGSEGV	Causes the exceptions <code>PROGRAM_ERROR</code> or <code>STORAGE_ERROR</code> .
SIGBUS	Used to implement <i>raise</i> statements. Also used to raise predefined exceptions when run time checking is enabled.
SIGFPE	Causes a <code>NUMERIC_ERROR</code> exception.

Note

The alarm signals `SIGALRM`, `SIGVTALRM`, and `SIGPROF` are not used or reserved in a non-tasking program. These signals are only sometimes used or reserved in a tasking 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.

In summary, if the Ada program contains no tasks, the following is true:

If the Ada program contains	
no delay statements	one or more delay statements
<code>SIGALRM</code> <i>not reserved</i>	<code>SIGALRM</code> <i>reserved</i>
<code>SIGVTALRM</code> <i>not reserved</i>	<code>SIGVTALRM</code> <i>not reserved</i>
<code>SIGPROF</code> <i>not reserved</i>	<code>SIGPROF</code> <i>not reserved</i>

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 <i>not reserved</i> SIGVTALRM <i>not reserved</i> SIGPROF <i>not reserved</i>	SIGALRM <i>reserved</i> SIGVTALRM <i>not reserved</i> SIGPROF <i>not reserved</i>
If time-slicing is enabled with SIGALRM timer with -W b, -S, a	SIGALRM <i>reserved</i> SIGVTALRM <i>not reserved</i> SIGPROF <i>not reserved</i>	SIGALRM <i>reserved</i> SIGVTALRM <i>not reserved</i> SIGPROF <i>not reserved</i>
If time-slicing is enabled with SIGVTALRM timer with -W b, -S, v	SIGALRM <i>not reserved</i> SIGVTALRM <i>reserved</i> SIGPROF <i>not reserved</i>	SIGALRM <i>reserved</i> SIGVTALRM <i>reserved</i> SIGPROF <i>not reserved</i>
If time-slicing is enabled with SIGPROF timer with -W b, -S, p	SIGALRM <i>not reserved</i> SIGVTALRM <i>not reserved</i> SIGPROF <i>reserved</i>	SIGALRM <i>reserved</i> SIGVTALRM <i>not reserved</i> SIGPROF <i>reserved</i>

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 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, or original signal handling action is 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/300 Exception Handling

The Ada/300 implementation used 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 signals used for Ada exception handling (SIGBUS, SIGILL, SIGFPE, or SIGSEGV) and your Ada program attempts to raise an exception, unpredictable program behavior will result.

The following signals have predefined meanings and are reserved for use by the Ada runtime for exception handling:

- SIGBUS
- SIGILL
- SIGFPE
- SIGSEGV

The signal SIGBUS is generated in your compiled Ada code whenever the corresponding Ada source code contains an explicit **raise** statement or when the compiler needs to raise a predefined exception. The compiled Ada code for an explicit **raise** statement will contain a branch to an odd address, which will cause the SIGBUS signal. The compiled Ada code for a run time check that raises a predefined exception will contain a conditional branch to an odd address, which will cause the SIGBUS signal. An unexpected SIGBUS signal that was generated outside of Ada code or was not generated by a branch to an odd address will cause the exception **PROGRAM_ERROR** to be raised.

The signal SIGILL is generated in your compiled Ada code when one of the predefined run time checks fails. The MC680x0 instructions CHK and TRAPV are occasionally used in compiled Ada code and generate the SIGILL signal. The CHK instruction will generate the exception CONSTRAINT_ERROR, while the TRAPV instruction will generate the exception NUMERIC_ERROR. An unexpected SIGILL that was generated outside of Ada code will cause the exception PROGRAM_ERROR to be raised.

The signal SIGFPE is generated in your compiled Ada code when a floating point division by zero occurs. Any other SIGFPE signal not caused by a divide by zero is an unexpected signal and will cause the exception PROGRAM_ERROR to be raised.

The signal SIGSEGV can be generated in your compiled Ada code when the stack of the main subprogram (the environment task stack space) grows too large. The Ada runtime will attempt to extend the size of this stack and continue program execution. If insufficient memory is available, the exception STORAGE_ERROR is raised. The Ada runtime system cannot dynamically extend the stack space for a child task stack. See section F 4.6 for more information on specifying child task stack sizes.

The signal SIGSEGV can also be generated when an object of an access type denotes an illegal address. Normally this will not be the case in an Ada program because an access type object is automatically initialized to null. However, using UNCHECKED_CONVERSION will greatly increase the possibility that a SIGSEGV signal can occur inside your Ada program. A SIGSEGV signal will cause the exception PROGRAM_ERROR to be raised in your Ada program.

F 9.4 HP-UX Signals Used for Ada/300 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 SIGALRM, SIGVTALRM (the default), or SIGPROF to control the time-slice interval. 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/300 Delay Timing

When an Ada program contains tasks and a delay statement is executed, the Ada runtime system uses SIGALRM to time the delay interval. The resolution of the SIGALRM timer is 1/50 of a second. Thus, all delay statements are implemented using actual delays that are integral multiples of 1/50 of a second. Non-zero delays for periods smaller than 1/50 of a second will delay for at least 1/50 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).

When an Ada program does not contain tasks, SIGALRM is not reserved. If an Ada program contains tasks but does not contain any delay statements, SIGALRM may or may not be reserved (see F 9.22, "Ada Timer Signals", 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/300 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 will see 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.

Warning

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/300 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.

Warning



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/300's Asynchronous Signals

The SIGALRM, SIGVTALRM, and SIGPROF signals (mentioned above) 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 300 User's Guide* on "Interfaced Subprograms and Ada's Use of Signals."

F 9.9 Programming in Ada/300 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/300 Interrupt Entries" and section F 12, "Interrupt Entries" for additional information.

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.
2047	Maximum number of actual compilation units in a library.
32767	Maximum number of enumeration elements in a single enumeration type (this limit is further constrained by the maximum number of characters for all enumeration literals of the type).
2047	Maximum number of "created" units in a single compilation.
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 <i>Ada User's Guide</i> , which discusses families of Ada libraries and the supported utilities (tools) to manage them).
-	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 <code>ada.mklib(1)</code> or <code>ada.umgr(1)</code> .
512	Maximum length of an input line for the tools <code>ada.fmgr(1)</code> and <code>ada.umgr(1)</code> .
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/300 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 accomodate 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 Conditions that Raise `NAME_ERROR`

When using `OPEN` and `CREATE`, the Ada exception `NAME_ERROR` will be raised if any path component exceeds 255 characters or if the entire path exceeds 1023 characters.

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.

(Delete section titled "Record Type Change of Representation")

F 10.3.4 Record Type Alignment Clause

A record type alignment clause can specify that a record type is either byte-aligned or word-aligned. Word alignment is 2-byte or 16-bit alignment. This release of the Ada/300 DS does not support 32-bit alignment. The largest alignment possible in this version of the Ada/300 DS is 16-bit alignment.

F 10.3.5 Pragma `INTERFACE` on Library Level Subprograms

In the current version of the compiler, it is not possible to supply a pragma `INTERFACE` to a library-level subprogram. Any subprogram that a pragma `INTERFACE` is applied to must be contained within an Ada compilation unit, usually a package.

F 11. Calling External Subprograms From Ada

In Ada/300, parameters of external interfaced subprograms are passed on the stack in reverse order of their declaration. The first parameter appears at the top of the stack. This same ordering is used by Hewlett-Packard for other language products on the HP 9000 Series 300 family of computers. The languages described in this section are HP implementations of 680X0 Assembler, C, FORTRAN 77, and Pascal on the HP-UX Series 300 systems.

In some cases, the interface requires not only the parameters but additional information to be pushed on the stack. This information can include a parameter count, a return result pointer, or other bookkeeping information.

When you specify the interfaced language name, that name is used to select the correct calling conventions for supported languages. Then, subprograms written in HP C, HP FORTRAN 77, and HP Pascal interface correctly with the Ada/300 subprogram caller. This section contains detailed information about calling subprograms. If the subprogram is written in a language from another vendor, you must follow the standard calling conventions.

In the Ada/300 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 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.

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. For example, in other vendors' implementation of Ada on the HP 9000 Series 300, the size of the predefined standard INTEGER type can be 16 bits, which is smaller than the Hewlett-Packard implementation size of 32 bits. Pay careful attention to the size of parameters because stacked values 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 HP 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/300, 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. For a call by reference, the value of the actual integer parameter is not copied or modified, but a 32-bit address pointer is pushed on the stack. For a call by value, a copy of the actual integer parameter value is pushed on the stack, with sign extensions as necessary to satisfy the requirements of the external subprogram. See sections F 11.2.1.1, F 11.3.1.1, F 11.4.1.1, and F 11.5.1.1 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") and are passed according to the convention given previously for integer values. See sections F 11.2.1.1, F 11.3.1.1, F 11.4.1.1, and F 11.5.1.1 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

The values of the predefined enumeration type `BOOLEAN` are represented in Ada/300 as 8-bit values. The Boolean value `FALSE` is represented by the 8-bit value zero and the Boolean value `TRUE` is represented by the 8-bit value `2#1111_1111#`. This is not the same as other enumeration types that occupy eight bits in Ada, where the position in the enumeration list at declaration defines the internal representation for the element.

When a Boolean is passed by reference, its value is not copied, but a 32-bit address pointer is pushed on the stack. When a Boolean is passed by value, a copy is pushed on the stack. The Ada Boolean value occupies one byte, and it is up to you to ensure that the Boolean is in a correct format for the external subprogram. Boolean types can be returned as function results from external interfaced subprograms, with caution as noted above.

See sections F 11.2.1.1, F 11.3.1.1, F 11.4.1.1, and F 11.5.1.1 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 11.2.1.1, F 11.3.1.1, F 11.4.1.1, and F 11.5.1.1 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 on the stack in a 32-bit container; this type is *never* extended to a 64-bit real. The Ada type LONG_FLOAT is a 64-bit real type and is passed on the stack in a 64-bit container. 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 on the stack, as indicated above.

See sections F 11.2.1.5, F 11.3.1.5, F 11.4.1.5, and F 11.5.1.5 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.

An object's address can be retrieved by applying the 'ADDRESS attribute to the object. In the case of an access type object, you may want either the address of the access type object or the address of the underlying object that it points to. The underlying object's address can be retrieved by applying the attribute 'ADDRESS in this way:

```
access_object.all'ADDRESS.
```

The use of .all implies that the 'ADDRESS operation applies to the contents of the access type object and not to the access type object itself.

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 pushed on the stack. If an access type object is passed by reference, however, the address of the access type object itself is pushed on the stack. This will effectively force references to the designated object to be double indirect references. See figure 11-1 for details.

Insert artwork here.

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 300 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 runtime 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 pushed on the stack 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 pushed on the stack. 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 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.

F 11.1.4 Record Types

Records (*Ada RM*, section 3.7) are always passed by reference in the HP implementation of Ada, pushing the 32-bit address of the first component of the record on the stack. 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."

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/300 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/300, 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/300. 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 MC680x0 microprocessor family (68K assembly language) upon which the Series 300 family is based.

When calling interfaced 68K assembly language subprograms, scalar and access parameters of mode in are passed by value; the value of the parameter object is copied and pushed on the stack. All other types of in parameters (arrays and records) and parameters of mode out and in out are passed by reference; the address of the parameter object is pushed on the stack.

When calling 68K assembly language subprograms, the processor scratch registers are considered to be A0, A1, D0, or D1. In external interfaced 68K assembly subprograms, processor registers D2 through D7 and A2 through A7 must be saved on entry and restored before returning to the Ada caller. The Ada compiler expects those registers to be unchanged across a call to an external interfaced subprogram. Only registers used in the called 68K assembly language subprogram must be saved and restored, but you are responsible for ensuring that all register contents (except for designated scratch registers) are unchanged. If any of the non-scratch registers are modified by the external subprogram, unpredictable program behavior might result.

The results returned by external function subprograms are expected to be in the register D0 if the result is scalar, or in register A0 if the result is an access value. **LONG_FLOAT** values that are represented as 64 bits are returned in two registers.

Only scalar types (integer, floating point, character, and enumeration types) and access types are allowed for the result returned by an external interfaced subprogram written in 68K assembly language.

For more information on 68K assembly language interfacing, see the *MC68020 32-Bit Microprocessor User's Manual* and the *MC68881 Floating-Point Coprocessor User's Manual*.

F 11.2.1 Scalar Types and Assembly Language Subprograms

See section F 11.1 for details.

F 11.2.1.1 Integer Types and Assembly Language Subprograms

See section F 11.1.1 for details.

F 11.2.1.2 Enumeration Types and Assembly Language Subprograms

See section F 11.1.1.2 for details.

F 11.2.1.3 Boolean Types and Assembly Language Subprograms

See section F 11.1.1.3 for details.

F 11.2.1.4 Character Types and Assembly Language Subprograms

See section F 11.1.1.4 for details.

F 11.2.1.5 Real Types and Assembly Language Subprograms

See section F 11.1.1.5 for details.

F 11.2.2 Access Types and Assembly Language Subprograms

See section F 11.1.2 for details.

F 11.2.3 Array Types and Assembly Language Subprograms

See section F 11.1.3 for details.

F 11.2.4 Record Types and Assembly Language Subprograms

See section F 11.1.4 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.

The compiler also prefixes the external subprogram name with one underscore character (`_`) and will truncate this name to 254 characters if necessary. The HP C compiler also performs this same transformation. This modification conforms to the naming conventions used by the HP linker (*ld(1)* - Link Editor) on the HP 9000 Series 300 Computer System.

In general, parameters passed to HP C subprograms from HP Ada programs are passed in 32-bit containers, except for 64-bit real quantities. The Ada/300 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` and `libm.a` are usually required. The Ada/300 binder will automatically provide the `-lm -lc` directives to the linker. You are not required to specify `-lm -lc` when binding the Ada program on the `ada(1)` command line.

For more information about C language interfacing, see the following manuals: *HP-UX Concepts and Tutorials: Programming Environment*, *HP-UX Concepts and Tutorials: Device I/O and User Interfacing*, and *HP-UX Portability Guide*. For general information about passing Ada types, see section F 11.1.

F 11.3.1 Scalar Types and HP C Subprograms

See section F 11.1.1 for details.

F 11.3.1.1 Integer Types and HP C Subprograms

See section F 11.1.1.1 for details.

When passed by value to an HP C subprogram, all integer values are sign-extended to 32 bits and pushed onto the stack. This method conforms to the HP C parameter passing convention for signed integers that are passed by *value*.

When passed by reference, a 32-bit address pointer to the integer object is pushed on the stack. The integer object is not sign-extended to 32 bits. This method conforms to the HP C parameter passing convention for signed integers that are passed by *reference*.

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/300 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 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 16 bits. When HP C passes enumeration types as *value* parameters, the values are extended to 32 bits; the high order 16 bits are disregarded. Because Ada also performs the extension to 32 bits for enumeration type values, they will be in the correct form for HP C subprograms.

HP C enumeration types are considered to be 16 bits. Therefore, Ada enumeration types that use the 32-bit representation *cannot* be passed to HP C as an enumeration type. For these 32-bit enumeration types, you will have to pass them as integers.

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.

When passed by reference, the original values are not extended. Therefore, Ada's 8-bit enumeration objects have no direct representation in HP C and *cannot* be used by an HP C subprogram. You may want to use a size specification clause to force the size of the enumeration type to 16 bits. Alternatively, an Ada 8-bit enumeration value passed by reference to HP C can be accessed as a character in C using the type `char *`.

Enumeration types are allowed for the result returned by an external interfaced subprogram written in HP C.

F 11.3.1.3 Boolean Types and HP C Subprograms

The type Boolean is not defined in HP C and the Ada/300 representation of Booleans does not correspond to any type in HP C. Ada Booleans can be converted to integers and then passed to external interfaced subprograms written in HP C.

In Ada, `BOOLEAN'POS(TRUE)` has a result of one and `BOOLEAN'POS(FALSE)` has a result of zero. Thus, the Ada attribute `'POS` can be used to convert an Ada Boolean type to an integer type. The integer type can then be passed to an HP C external subprogram.

Boolean types are represented internally as an 8-bit enumeration type of the predefined type (`FALSE` or `TRUE`). Although a predefined enumeration type, Ada Boolean values are represented in eight bits in a manner that is different from other enumeration types in the HP implementation of Ada. The internal representation of `FALSE` corresponds to `2#0000_0000#` and `TRUE` corresponds to `2#1111_1111#` (that is, all zeros or all ones in eight bits, respectively). Note the value of `BOOLEAN'POS(TRUE)` and the value of `BOOLEAN'POS(FALSE)`. To ensure the portability of your code, Ada Booleans should be passed using the result of the `'POS` attribute.

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 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.

Character types are represented as 8-bit enumeration values in the HP implementation of Ada. When passed by value to an HP C subprogram, all character values are zero-extended to 32 bits and pushed onto the stack. This method conforms to the HP C parameter passing convention for unsigned characters that are passed by *value*.

When passed by reference, a 32-bit address pointer to the character object is pushed on the stack. The character object is not sign-extended to 32 bits. This method conforms to the HP C parameter passing convention for unsigned characters that are passed by *reference*.

The Ada predefined type `CHARACTER` is compatible with the HP C `char` type (except for the smaller range of the Ada type.) The type `CHARACTER` can be passed by *value* or by *reference* using the HP C types `char` and `char *` respectively.

Character types are allowed for the result returned by an external interfaced subprogram written in HP C.

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

See section F 11.1.1.5 for details.

The HP C 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 on the stack. The HP C calling convention for functions returning values of real types also requires that a 64-bit double precision real be returned (in registers D0 and D1) and converted to 32 bits afterwards if the function was to return a 32-bit real value.

The Ada type `FLOAT` is a 32-bit real type and is passed on the stack in a 32-bit container; it is *not* extended to a 64-bit real. The Ada type `FLOAT` should *not* be used as a parameter type when passing reals by *value* to external HP C subprograms. The Ada type `LONG_FLOAT` is a 64-bit real type and is passed on the stack in a 64-bit container. The Ada type `LONG_FLOAT` should be used as a parameter type when passing reals by *value* to external HP C subprograms. The same kind of restrictions make the Ada type `FLOAT` incompatible with HP C when used as a function result type. The Ada type `LONG_FLOAT` *must* be used whenever you want to interface with an HP C function that returns a floating point type.

The above limitation on 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 composite type, such as an array or a 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.

When passing floating point values by *reference* to an interfaced subprogram written in HP C, the 32-bit address of the object is pushed on the stack.

F 11.3.2 Access Types and HP C Subprograms

Ada access types are pointers to Ada objects. In the implementation of Ada/300 for the Series 300 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.

Ada access types of mode `in` are passed by *value*. 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 pushed on the stack.

Ada access types of mode `out` or `in out` are passed by *reference*. When an access type object is passed by reference, the address of the access type object itself is pushed on the stack. This will effectively force references to the designated object to be double indirect references. See figure F-1 in section F 11.1.2 for details.

Ada access types may be returned as function results from external interfaced subprograms written in HP C.

An object designated by an Ada access type can be passed to an HP C external subprogram subject to rules applicable to the type of the underlying object.

F 11.3.3 Array Types and HP C Subprograms

See section F 11.1.3 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/300, 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, 300. 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 will be 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  
--   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 (ELEMENT_TYPE => 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, integer) operator
  LEN  : NATURAL := 0;
  START : SYSTEM.ADDRESS;
  CUR   : CHARACTER;
begin
  START := SRC.all'ADDRESS;
  loop
    CUR := FETCH_CHAR (FROM => START + INTEGER (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, integer) 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 + INTEGER (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 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.

No special handling of leading underscores is required because this is handled by the compiler to conform to standard calling conventions. The pragma ensures that the underscore required in front of the HP FORTRAN 77 subroutine name is correctly inserted by the compiler.

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," 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. Boolean and access type results are not supported.

The FORTRAN libraries `libI77.a` and `libF77.a` are required when linking Ada programs that contain external interfaced subprograms written in

FORTTRAN on the HP 9000 Series 300 implementation of Ada. The binder will specify that the linker search the HP FORTRAN 77 libraries if any pragma INTERFACE to FORTRAN are in the closure at bind time. For more information, see the *FORTTRAN/9000 Reference Manual*.

For general information about passing types to interfaced subprograms, see section F 11.1.

F 11.4.1 Scalar Types and HP FORTRAN 77 Subprograms

FORTTRAN 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," 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

Table 11-3 summarizes the correspondence between integer types in Ada/300 and HP FORTRAN 77.

Table 11-3.
Ada/300 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);
```

Because this has the effect of obscuring the types of the actual parameters, it is suggested that such declarations be encapsulated within an inlined Ada body so that the parameter types are made visible. An example follows:

```
-- specification of function to encapsulate
function FTNFUNC (PARM1 : INTEGER;
                  PARM2 : FLOAT    ) return INTEGER;
pragma INLINE (FTNFUNC);

with SYSTEM;
-- body of function to encapsulate
function FTNFUNC (PARM1 : INTEGER;
                  PARM2 : FLOAT    ) return INTEGER is
  function FORTFUNC (P1, P2 : SYSTEM.ADDRESS) return INTEGER;
  pragma INTERFACE (FORTRAN, FORTFUNC);

begin -- function FTNFUNC
  return FORTFUNC (PARM1'ADDRESS, PARM2'ADDRESS);
end FTNFUNC;
```

In the previous example, the name of the interfaced external function subprogram (written in HP FORTRAN 77) is `FORTFUNC`. This name is declared in the following way:

```
INTEGER*4 FUNCTION fortfunc (i, x)
INTEGER*4 i
REAL*4 x
...
END
```

F 11.4.1.2 Enumeration Types and HP FORTRAN 77 Subprograms

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

In Ada/300, the type `BOOLEAN` is represented in eight bits and *cannot* be used as a logical or short logical (that is, `LOGICAL*2`) in HP FORTRAN 77. However, an `INTEGER` may be passed instead of the Boolean value (which can be mapped to a `LOGICAL*2`).

Although a predefined enumeration type, Ada Boolean values are represented in eight bits in a manner different from other enumeration types in the HP implementation of Ada. The internal representation of `FALSE` corresponds to `2#0000_0000#` and `TRUE` corresponds to `2#1111_1111#` (that is, all zeros or all ones in eight bits, respectively). Note that the value of `BOOLEAN'POS(TRUE)` is one and the value of `BOOLEAN'POS(FALSE)` is zero.

The Ada attribute `'POS` can be used to convert an Ada Boolean to an integer type, which can then be passed to an HP FORTRAN 77 external subprogram. Note that in Ada, `BOOLEAN'POS(TRUE)` has a result of one, and that `BOOLEAN'POS(FALSE)` has a result of zero. This result can then be converted to an integer type of 16 bits and subsequently can be treated as a `LOGICAL*2` in HP FORTRAN 77 and passed as a parameter.

Likewise, function results of type `LOGICAL` from an HP FORTRAN 77 function subprogram may be considered in Ada as having a `SHORT_INTEGER` type representing the position (attribute `'POS`) of the equivalent Boolean value.

For example, an HP FORTRAN 77 subroutine of the form

```
SUBROUTINE logical_sub (logical_parm)
  LOGICAL*2 logical_parm
  ...
  RETURN
```

can be called after suitable declarations in Ada as follows:

```

subtype FORTRAN_LOGICAL is SHORT_INTEGER; -- 16 bits
BVAR : BOOLEAN;          -- will be input Boolean value

```

```

-- define a place holder for FORTRAN logical sized integer
BTEMP : FORTRAN_LOGICAL;

```

```

procedure BOOL_PROC(B_PARM : in out FORTRAN_LOGICAL);
pragma INTERFACE (FORTRAN, BOOL_PROC);
pragma INTERFACE_NAME ( BOOL_PROC, "logical_sub");

```

The above subroutine is called from the Ada program as:

```

BTEMP := FORTRAN_LOGICAL(BOOLEAN'POS(B_VAR)); -- get equivalent
BOOL_PROC( BTEMP ); -- call the FORTRAN subroutine "logical_sub"

```

Note that the expression

```

BOOLEAN'POS(B_VAR)

```

evaluates to a value of one if B_VAR is TRUE or a value of zero if B_VAR is FALSE, while the expression of the form

```

BTEMP := FORTRAN_LOGICAL(...);

```

is an explicit type conversion to a 16-bit integer type.

Note that the call from the Ada program in the preceding example involved an assignment rather than an embedded expression. In Ada, an actual parameter passed as an in out parameter *cannot* be an expression; it must denote an object. Ada does, however, allow an explicit type conversion to be performed on the parameter before it is passed as a parameter if the two types are of the same size.

Therefore, the following example is not legal Ada:

```

BOOL_PROC(FORTRAN_LOGICAL(BOOLEAN'POS(B_VAR))); -- not legal

```

But this example is legal Ada:

```

BOOL_PROC(FORTRAN_LOGICAL(SOME_16_BIT_INTEGER_OBJECT)); -- legal

```


F 11.4.1.4 Character Types and HP FORTRAN 77 Subprograms

See section F 11.1.1.4 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. 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/300 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 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/300 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/300 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/300 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/300 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

Parameters of type FLOAT in Ada correspond to the default REAL (REAL*4) format in HP FORTRAN 77. The Ada type LONG_FLOAT is equivalent to the HP FORTRAN 77 type DOUBLE PRECISION (or REAL*8). HP FORTRAN 77 follows the standard IEEE floating point conventions for both 32-bit and 64-bit floating point numbers. Both 32-bit real values and 64-bit real values can be passed as parameters.

A REAL value from an HP FORTRAN 77 external function subprogram may be returned as a function result of type FLOAT in an Ada program.

A DOUBLE PRECISION (or REAL*8) value from an HP FORTRAN 77 external function subprogram may be returned as a function result of type LONG_FLOAT in an Ada program.

When passed as parameters to an interfaced subprogram written in HP FORTRAN 77, the original real values are not altered and a 32-bit address of the real object is pushed on the stack.

F 11.4.2 Access Types and HP FORTRAN 77 Subprograms

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/300, 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 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 F00(3,5,10)
```

or

```
REAL*4 F00(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
                    F   : in out FLOAT;    -- must be IN OUT
                    S2 : STRING;           -- passed by reference
                    LS1 : INTEGER ;        -- len of string S1,
                                           -- must be IN
                    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, FLOAT_VAL, SB, LENGTH_SA, LENGTH_SB);
end FTNSTR;
```

Note

Note that the order of the string lengths is in the same order of their appearance in the corresponding parameter and that they appear after all other parameters (at the end of the parameter list).

The HP FORTRAN 77 external subprogram is the following:

```
SUBROUTINE Fextr (s1, r, s2)
CHARACTER *(*) s1, s2
REAL*4  r
...
END
```

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.

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 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.

The name of the Pascal external subprogram viewed externally from Ada is the single identifier modp_p_subr, which is formed by the module name modp and the procedure name p_subr. A pragma INTERFACE_NAME will often be used for HP Pascal because the naming convention in HP Pascal always appends the module name to the procedure name when generating external names for the ld(2) command.

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.

The *HP Pascal Language Reference Manual* states that calling external interfaced Pascal language subprograms from Ada programs should use the two assembly language routines `asm_initproc` and `asm_wrapup`. However, these routines interfere with the Ada runtime and *cannot* be used by an Ada program. The routine `asm_initproc` performs three functions: it associates the file descriptors `stdin` and `stdout` with the standard Pascal files `INPUT` and `OUTPUT`, it initializes the Pascal heap manager, and it sets a Pascal error handler for all HP-UX signals. The Ada runtime requires many of these signals to be mapped to the Ada runtime. Your Ada program will not handle exceptions correctly after a call to `asm_initproc`.

Because `asm_initproc` is not called by an Ada program before executing HP Pascal subprograms, the following limitations are imposed upon the HP Pascal subprograms. First, no access to the standard Pascal files `INPUT` and `OUTPUT` is allowed. Second, the subprogram cannot dynamically allocate any objects in HP Pascal using the *new* call. These restrictions apply to all HP Pascal functions and procedures that are reachable from the interfaced subprogram.

For more information on Pascal interfacing, see the *HP Pascal Language Reference Manual* and the section, "HP-UX Implementation, Pascal and other Languages". 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 pushed on the stack. All other types of `in` parameters (arrays and records) and parameters of mode `out` and `in out` are passed by reference: The address of the object is pushed on the stack. 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`.

For Pascal external interfaced subprograms called from an Ada program, all parameters are passed in containers of size 32 bits, except for Ada `LONG_FLOAT` parameters which are passed in 64 bits.

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.

F 11.5.1 Scalar Types and HP Pascal Subprograms

See section F 11.1.1 for details.

F 11.5.1.1 Integer Types and HP Pascal Subprograms

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/300 versus HP Pascal Integer Correspondence

Ada	HP Pascal	Bit Length
predefined type INTEGER	predefined type integer	32
predefined type SHORT_INTEGER user-defined types type I16=-32768 ... 32767;	16	
predefined type SHORT_SHORT_INTEGER	user-defined type type I8 = -128 ... 127;	8

When passed by *value* to an HP Pascal language subprogram, all integer values are extended to 32 bits to conform to the HP Pascal parameter passing conventions. Ada INTEGER (32 bits) types passed by value are pushed onto the stack without alteration. Values of type SHORT_SHORT_INTEGER (8 bits) and type SHORT_INTEGER (16 bits) are sign extended to 32 bits and pushed on the stack. The automatic sign extension helps in passing different sized values that are within an acceptable range.

When passed by *reference*, the integer values are not changed; the original values are not sign extended to 32 bits. A 32-bit address pointer to the integer object is pushed on the stack.

In HP Pascal, user-defined INTEGER types can occupy either 32 bits or 16 bits. When integer types are passed by reference, the two integer types must be exactly the same size. In table 11-4 above, the HP Pascal user-defined integer type I16 can be used when values of the Ada type SHORT_INTEGER are passed by reference.

When passing integers by reference, note that the predefined Ada type SHORT_SHORT_INTEGER (stored in eight bits) has no equivalent integer representations in HP Pascal. This means that parameters of the type SHORT_SHORT_INTEGER *cannot* be passed by *reference* to 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

Ada and HP Pascal have slightly different implementations of enumeration types. In Ada, enumeration types can have a size of 8, 16, or 32 bits. In HP Pascal, all enumeration types are 16-bit quantities. The parameter passing conventions for enumeration types in both languages is to pass enumeration values in 32-bit stack containers. This allows the two implementations to treat enumeration types as equivalent. However, an Ada enumeration type that contains a representation clause that includes 32-bit values *cannot* be used as an HP Pascal enumeration value.

When passed by *reference*, the original value is not altered by sign extension and a 32-bit address pointer to the enumeration object is pushed on the stack.

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 for details.

F 11.5.1.4 Character Types and HP Pascal Subprograms

See section F 11.1.1.4 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

HP Pascal uses the standard IEEE floating point conventions for both 32- and 64-bit floating point numbers. Ada `FLOAT` values correspond to HP Pascal real values. Ada `LONG_FLOAT` values correspond to HP Pascal longreal values.

When passed by *value* to interfaced subprograms written in HP Pascal, values of type `FLOAT` and `LONG_FLOAT` are copied and pushed on the stack. Both 32- and 64-bit real types can be returned as results from an external interfaced function subprogram written in HP Pascal.

When passed by *reference* to an interfaced subprogram written in HP Pascal, the original values are not altered and a 32-bit address of the object is pushed on the stack.

F 11.5.2 Access Types and HP Pascal Subprograms

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.

An object designated by an Ada access type can be passed to an HP Pascal external subprogram, subject to rules applicable to the type of the underlying object.

Ada access types of mode *in* are passed by value. When an access type is passed by value, a copy of the 32-bit object address is pushed on the stack. If the type is passed by reference, however, a 32-bit address pointer to the object address location is pushed on the stack. This is effectively a double indirect address to the underlying object. (See figure 11-1 in section F 11.1.2.)

Ada access types may be returned as function results from external interfaced subprograms written in HP Pascal.

F 11.5.3 Array Types and HP Pascal Subprograms

See section F 11.1.3 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.

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. These parameters are the bounds of the array and the size of the array elements in bytes; the parameters must be declared in the above order and they must immediately precede 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 and element size parameters is needed for the two formal parameters A and B in the following example. The actual parameters passed to the formal A and B parameters must have the same index bounds.

Note that two sets of implicit bound and element size parameters are needed for the two-dimensional conformant array formal parameter C (one set for each dimension). If there were additional two-dimensional conformant array parameters, and they were all declared in the same comma-separated parameter list in Pascal (with the C parameter), only the two sets of implicit bound and element size parameters would be required for the entire list. For example, if the Pascal function heading is

```
FUNCTION Vector1 (VAR a,b: ARRAY [i..j: INTEGER] of INTEGER;  
                  VAR c:  ARRAY [p..q] of ARRAY [r..s] of INTEGER)  
                  : INTEGER;
```

The Ada declaration to call such a Pascal function would be as 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 (I, J: INTEGER; ELSIZE1: INTEGER;
                          A, B: VECTOR;
                          P, Q: INTEGER; ELSIZE2: INTEGER;
                          R, S: INTEGER; ELSIZE3: INTEGER;
                          C: VECTOR2) return INTEGER;

    pragma INTERFACE (PASCAL, VECTORTHING);

    INTEGER_SIZE : constant INTEGER
                  := INTEGER'SIZE / SYSTEM.STORAGE_UNIT;
    VECTOR_SIZE  : constant INTEGER
                  := VECTOR'SIZE  / SYSTEM.STORAGE_UNIT;

    W, V: VECTOR;
    X   : VECTOR2;
    I   : INTEGER;

begin -- PASCAL_CONFORM_FUNC
    I := VECTORTHING (VECTOR'FIRST, VECTOR'LAST, INTEGER_SIZE, W, V,
                     VECTOR2'FIRST, VECTOR2'LAST, VECTOR_SIZE,
                     VECTOR'FIRST, VECTOR'LAST, INTEGER_SIZE, X);
end PASCAL_CONFORM_FUNC;
```

Array types *cannot* be returned as function results from external interfaced subprograms written in HP Pascal.

F 11.5.4 String Types and HP Pascal Subprograms

Passing variable length strings between Ada and HP Pascal is supported with some restrictions. The parameters must be passed by reference only. HP Pascal programs must declare *var* parameters and the Ada program must declare the parameters to be of mode *in out* or *out* to ensure passing by reference.

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 to a packed array of characters in Pascal. 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..80 );

    -- 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, "p_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: an 8-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
```

```
    type PASCAL_LENGTH is range 0 .. (2**8)-1;
```

```
-- The size specification clause forces the type  
-- PASCAL_LENGTH to be an 8-bit unsigned integer.
```

```
for PASCAL_LENGTH'SIZE use 8;  
-- Define an Ada record that will correspond exactly  
-- with the HP Pascal type: string[80]
```

```
type PASCAL_STRING80 is  
    record  
        LEN : PASCAL_LENGTH  
        S   : STRING ( 1..80 );  
    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 .. 7;  
        S   at 1 range 0 .. 80*8;  
    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);
```

```
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 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.

Table 11-7.
Types Returned as External Function Subprogram Results

Ada Type	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	not allowed (1)	not 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	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).

Table 11-8 summarizes information presented in sections F 11.2 through F 11.5.

Table 11-8. Parameter Passing in the Series 300 Implementation

Ada Type	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 (3)
FLOAT	allowed	allowed	allowed	allowed
FIXED POINT	not allowed	not allowed	not allowed	not allowed
ACCESS	allowed	allowed	not allowed	allowed
ARRAY (4)	not allowed	allowed	allowed (5)	allowed
STRING	not allowed	allowed (6)	allowed (7)	not allowed (8)
RECORD	not allowed	allowed	not 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) Passed by value only.
- (4) Using only arrays of compatible component types.
- (5) See warning on layout of elements in section for each language.
- (6) Special handling of null terminator character is required.
- (7) Requires that the length also be passed.
- (8) Ada strings can be passed to a Pascal PAC (Packed Array of Characters.)

F 11.7 Potential Problems Using Interfaced Subprograms

- The Ada runtime on the HP 9000 Series 300 computer 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 interruptibility 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 runtime 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 interfaced 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 */

    sigsetmask (mask); /* restore old mask so Ada runtime can function */

    ...
}
```

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 runtime 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 300 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 11.8 Input-Output From Interfaced Subprograms

Using I/O from interfaced subprograms written in other languages requires caution. Some areas in which problems can arise are discussed in this section.

F 11.8.1 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.

F 11.8.2 Preconnected I/O and Interfaced Subprograms

The standard HP-UX files `stdin` and `stdout` are preconnected by Ada I/O. If non-blocking interactive I/O is used, additional file descriptors will be used for interactive devices connected to `stdin` or `stdout`. Ada does not preconnect `stderr`, which is used for run-time 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`. For more details on Ada I/O, see the *Ada RM*, section 14 and the section on using the Ada Development System in the *Ada 300 User's Guide*.

F 11.8.3 Interactive I/O and Interfaced Subprograms

The default I/O system behavior is **NON-BLOCKING** for Ada programs with tasking and **BLOCKING** for sequential (non-tasking) Ada programs. HP's implementation of Ada/300 will set non-blocking I/O by default for interactive files and pipes if the program contains tasks. If the Ada program contains no task structures (that is, it is a sequential program), blocking I/O is set for interactive files and pipes. You can override the defaults with binder options.

The binder option **-W b, -b** sets up blocking I/O and the binder option **-W b, -B** sets up non-blocking I/O. In non-blocking I/O, a task (or Ada main program) will not block when attempting interactive input if data is not available. If the I/O request cannot be immediately satisfied, the Ada runtime will place the task that requested I/O on a suspend queue and will awaken the task when the I/O operation is complete. This arrangement allows other tasks to continue execution; the task requesting I/O will be suspended until the I/O operation is completed by the Ada runtime.

F 12. Interrupt Entries

This chapter describes interrupt 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 a generalization of the language-defined mechanism that includes 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.

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 below). The pool of free buffers is allocated once at the program startup by calling `INIT_INTERRUPT_MANAGER` with the number of buffers specified (see below).

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; runtime 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, the 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 runtime calls and must be compiled with checks off (using the `-Roption`), 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, 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.

There are several techniques that can be used to calculate an appropriate value for the `MAX_PARAM_AREA_SIZE` parameter. This 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 runtime 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).

■ Techniques for determining parameter block sizes follows:

1. Refer to section F 4, "Type Representation", to determine the representations of the various types of parameters. All scalar types and those record types that occupy less than or exactly 64 bits (eight storage units) have their sizes rounded up to the next multiple of 16 bits (two storage units) within a parameter block. All record types that occupy more than 64 bits (eight storage units), and all other non-scalar types, are passed by reference as a 32-bit pointer occupying four storage units in the parameter block.
2. Use the 'SIZE attribute on each of the parameter types. See the method above for parameter types whose sizes may be other than expected.
3. Use the -S option when compiling the units that contain the interrupt entry calls (the Ada signal handlers) and examine the generated code for the call to determine the size of the parameter block. The parameter block is built by the generated code just prior to the call to the Ada runtime routine ENTRY_CALL.

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 a 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 12.9, "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 runtime routines update critical runtime 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.

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 if your Ada handler makes any Ada runtime system routine calls, you can compile the Ada handler with the `-S` option to obtain an assembly language listing. Then, look at the listing of the generated code for your Ada handler procedure. If calls to any of the following Ada runtime system routines appear, your Ada handler is "unsafe"

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).

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 runtime routines is listed in section F 12.13, "Ada Runtime Routine Descriptions".

Warning

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

To be provided???????????

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.
```

```
-----
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.

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
<i>Fixed element size</i>	
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.
<i>Variable element size</i>	
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.

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 <code>accept</code> 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 <code>select</code> 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 <code>delay</code> 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	Implementation of a simple entry call. <i>Note:</i> a call to this routine 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.
ENTRY_CALL	Implementation of a simple entry call. <i>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.</i>
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 runtime 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'VALUE, 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 ??? 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 32767 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 32767 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, 8-33
 - sequential, 8-33
- access protection, 8-11
- access rights, 8-3, 8-7
- access types
 - alignment, 4-25
 - as function results, 11-10, 11-24, 11-56
 - bit representation, 11-9, 11-56
 - caution, 11-10
 - collection size specification, 4-23, 4-24
 - FORTRAN, 11-40
 - general considerations, 11-9
 - HP C subprograms, 11-24
 - illustration of passing methods, 11-9
 - internal representation, 4-23
 - I/O operations, 8-13
 - minimum size, 4-25
 - not returned as function results, 11-40
 - passing to external subprograms, 11-9, 11-56
 - size, 4-25
 - value of 'STORAGE_SIZE, 4-23
- address
 - pointer, 11-19, 11-24, 11-40, 11-53, 11-56
 - scalar parameter, 11-33
- ADDRESS, 2-2
- 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, 4-33
 - enumeration types, 4-6
 - integer types, 4-12
 - record types, 4-36, 4-50
- APPEND, 8-27
- 'ARRAY_DESCRIPTOR, 4-47
- ARRAY_DESCRIPTOR, 2-1
- array objects, 11-25
- arrays
 - as function results, 11-41
 - caution, 11-41
 - unconstrained, 7-1
- array types
 - alignment, 4-30
 - as function results, 11-11, 11-57
 - C language, 11-25
 - correspondence with HP Pascal types, 11-56
 - default size, 4-29
 - gaps between components, 4-29
 - general considerations, 11-11
 - layout, 1-12, 4-29
 - minimum size, 4-29
 - Pascal, 11-56
 - passing to external subprograms, 11-11, 11-41, 11-56

- pragma PACK, 4-29
- size of dynamic arrays, 4-32
- string types, 11-11
- ASCII, 3-14, 8-10
- assembly language
 - access types, 11-16
 - array types, 11-16
 - Boolean types, 11-15
 - character types, 11-15
 - enumeration types, 11-15
 - floating point types, 11-15
 - integer types, 11-15
 - processor registers, 11-14
 - record types, 11-16
 - scalar types, 11-15
 - subprograms, 11-14
- asynchronous signals, 9-13
- attribute
 - 'ADDRESS, 2-2
 - implementation-dependent, 2-1
 - 'POS, 11-36

B

- binary files, 8-32
- binder
 - and pragma INTERFACE, 1-6
- bit ordering
 - component clause, 4-36
- blocked signal, 11-67, 11-69
- BOOLEAN, 4-3, 11-36
- BOOLEAN'POS(FALSE), 11-36, 11-54
- BOOLEAN'POS(TRUE), 11-36, 11-54
- Boolean types
 - as function results, 11-7
 - bit representation, 11-7, 11-36
- C language, 11-21
 - converting to integer types, 11-21
- FORTTRAN, 11-36
- general considerations, 11-7
- Pascal, 11-54

- passing to external subprograms,
 - 11-7, 11-36
- predefined, 3-4
- returned as function results, 11-21
- buffer, 8-8, 8-10, 8-23
- BUFFER_SIZE, 8-23

C

- calling conventions, 1-3, 11-1
- catenation operators, 3-13
- char, 11-20, 11-22, 11-25, 11-54
- CHARACTER, 3-12, 11-19, 11-22,
 - 11-41, 11-54
- character types
 - as function results, 11-7
 - bit representation, 11-7, 11-22, 11-54
 - calling FORTRAN, 11-32
 - calling Pascal, 11-54
 - correspondence with HP C types,
 - 11-22
 - general considerations, 11-7
 - passing to external subprograms,
 - 11-22
- checks, 11-3, 11-11
- C language
 - access types, 11-24
 - access types as function results, 11-18
 - array types, 11-25
 - bit representation of parameters passed
 - to, 11-18
 - Boolean types, 11-21
 - calling, 11-17
 - character types, 11-22
 - enumeration types, 11-20
 - integer correspondence, 11-19
 - integer types as function results,
 - 11-19
 - real types, 11-18
 - record types, 11-31
 - scalar types, 11-18
 - scalar types as function results, 11-18

- types returned as function results, 11-18
- compiler
 - errors, 11-4
 - limitations, 10-1
- COMPLEX, 11-49
- components of a record, organization of, 11-4
- composite types, 11-2
- consistency checks, 11-3
- CONSTRAINT_ERROR, 3-13
- CONTROLLED, 1-18
- COUNT_LAST, 8-3
- CREATE, 8-2, 8-3, 8-7, 8-8, 8-19

D

- default access protection, 8-11
- delay statement, 9-9, 11-67
- device, 8-3, 8-10, 8-12, 8-23
- dimensions
 - declared in reverse order, 11-41
 - illustration of reversed order, 11-43
- direct access, 8-33
- direct files
 - elements, 8-8
 - index, 8-8
- DIRECT_IO files, 8-8, 8-32
- disks, 8-12
- double, 11-23
- DOUBLE_COMPLEX, 11-49
- DOUBLE_PRECISION, 11-39
- DURATION, 3-12
- dynamic components, 4-41, 11-13

E

- ELABORATE, 1-1, 1-13
- elements, 8-8
- embedded expression, 11-37
- END_OF_FILE, 8-10
- END_OF_PAGE, 8-10
- enumeration

- alignment, 4-6
- enumeration types
 - as function results, 11-6, 11-54
 - bit representation, 11-6, 11-54
 - default size, 4-6
 - FORTRAN, 11-35
 - general considerations, 11-6
 - internal representation, 4-3, 4-5, 4-6
 - minimum size, 4-5, 4-6
 - Pascal, 11-54
 - passing to external subprograms, 11-6, 11-35, 11-54
 - returned as function results, 11-20
 - syntax, 4-2
 - unsigned representation, 4-6
- errors
 - compiler, 11-4
 - USE_ERROR exception, 8-25
- exceptions
 - handling, 9-5, 9-6
 - predefined, 3-13
- execution of delay statements, 11-67
- EXPORT, 1-1
- expression, 11-37
- extension, 11-6, 11-20
- external files
 - access rights, 8-7
 - appending to, 8-27
 - associating Ada file objects with, 8-2, 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, 1-1
- external subprograms

- delay statements, 9-9
- general aspects of calling, 11-1
- I/O on files opened by Ada, 11-70
- parameter values, 11-3
- passing access types to, 11-9, 11-24, 11-56
- passing array types to, 11-11, 11-41, 11-56
- passing Boolean types to, 11-7, 11-36
- passing character types to, 11-22
- passing enumeration types to, 11-20, 11-35, 11-54
- passing floating point types, 11-55
- passing floating point types to, 11-8, 11-23, 11-39
- passing integer types to, 11-6, 11-33, 11-53
- passing record types to, 11-62
- passing string types to, 11-11, 11-25, 11-44
- passing task types to, 11-13
- potential problems using, 11-66
- pragma INTERFACE, 1-4
- protecting code with a critical section, 11-69
- signals, 9-9
- types not passed as parameters to, 11-2
- external subroutines names, 1-6

F

FIELD'LAST, 8-3

files

- Ada definition, 8-1
- appending, 8-27
- associate NAME with file object, 8-2
- associating external with file object, 8-1
- binary, 8-32
- blocking, 8-28
- buffering, 8-23

- characteristics, 8-1
- correspondence of Ada with external, 8-2
- direct, 8-8, 8-11
- DIRECT_IO, 8-32
- disk, 8-23
- external, 8-3, 11-3
- HP-UX pathname, 8-1
- I/O, 8-1
- length of elements, 8-8
- name of external, 8-3
- object, 1-6, 8-2
- protecting, 8-21
- protection flags, 8-21
- sequential, 8-7, 8-11, 8-12, 8-27
- SEQUENTIAL_IO, 8-32
- shared, 8-11, 8-12
- tasking, 8-11
- terminal input, 8-19
- terminator, 8-10
- text, 8-10, 8-11, 8-27, 8-30
- fixed point
 - predefined, 4-17
 - types, 4-17
- fixed point types
 - alignment, 4-22
 - as function results, 11-8, 11-23, 11-55
 - default size, 4-21
 - external subprograms, 11-2
 - general considerations, 11-8
 - internal representation, 4-18
 - minimum size, 4-19, 4-22
 - parameters, 11-23
 - unsigned representation, 4-19, 4-22
 - value of 'SMALL, 4-18
- FLOAT, 3-10, 11-23, 11-41, 11-55
- floating point types
 - alignment, 4-16
 - as function results, 11-8, 11-39
 - bit representation, 11-8, 11-23
 - calling FORTRAN, 11-32

- default size, 4-16
- FORTTRAN, 11-39
- general considerations, 11-8
- internal representation, 4-13
- minimum size, 4-16
- Pascal, 11-55
- passing to external subprograms,
 - 11-8, 11-39, 11-55
- passing to HP C, 11-23
- predefined, 4-13
- FORM, 8-2, 8-7
- FORM parameter
 - appending to a file, 8-27
 - attributes, 8-19
 - blocking, 8-28
 - file buffering, 8-23
 - files protection flags, 8-21
 - file structuring, 8-30
 - format, 8-19
 - shared files, 8-25
 - terminal input, 8-29
- FORTTRAN
 - access types, 11-40
 - array types, 11-41
 - Boolean types, 11-36
 - calling subprogram, 11-32
 - enumeration types, 11-35
 - equivalence of types, 11-49
 - fixed point types, 11-39
 - floating point types, 11-39
 - integer correspondence, 11-33
 - integer types, 11-33
 - libraries, 11-32
 - other types, 11-49
 - record types, 11-48
 - scalar types, 11-33
 - string types, 11-44
- function results
 - access types, 11-24
 - access types as, 11-40, 11-56
 - array types as, 11-41, 11-57

- Boolean types as, 11-21
- enumeration types as, 11-20, 11-54
- fixed point types as, 11-55
- floating point types as, 11-23, 11-39
- integer types as, 11-33
- record types as, 11-31, 11-48, 11-62
- types returned as, 11-18
- types returned from HP assembly
 - language, 11-14
- types returned from HP C, 11-19
- types returned from HP FORTRAN
 - 77, 11-32
- types returned from HP Pascal, 11-51

G

- gap sizes, 4-30
- generating code, 1-13
- GROUP, 8-21

H

- HP-UX
 - library function, 1-6
 - pragma EXPORT, 1-7
 - pragma EXTERNAL_NAME, 1-8
 - pragma INTERFACE, 1-6
 - signals, 9-1
 - system call, 1-6
 - utilities and routines, 11-70

I

- implementation-dependent
 - attributes, 2-1
 - characteristics of external files, 8-7
- implicit components, 4-38
- implicit parameters, 11-32, 11-44
- IMPROVE, 1-1, 1-12
- INDENT, 1-1, 1-10
- index
 - direct files, 8-8
 - set when file opened, 8-27
- indirect components, 4-40

- INLINE, 1-1, 1-14
- int, 11-19
- INTEGER, 11-4, 11-6, 11-19, 11-41, 11-53
- integer type
 - alignment, 4-12
 - size, 4-10
- integer types
 - as function results, 11-6, 11-33
 - bit representation, 11-53
 - calling FORTRAN, 11-32
 - compatibility with HP Pascal types, 11-52
 - correspondence between Ada and HP C, 11-19
 - correspondence with FORTRAN integers, 11-33
 - default size, 4-10
 - general considerations, 11-6
 - internal representation, 4-7
 - minimum size, 4-8, 4-11
 - passing to external subprograms, 11-6, 11-33, 11-53
 - predefined, 4-7
 - unsigned representation, 4-8, 4-11
- interactive devices, 8-12, 11-67
- INTERFACE
 - limitation, 10-7
 - pragma, 1-1, 1-2, 11-17, 11-32, 11-50
- INTERFACE_NAME, 1-1, 1-2
- interrupt, 11-66, 11-67
- interrupt entries, 12-1
- I/O
 - access types, 8-13
 - calls to interactive devices, 11-67
 - considered as external files, 8-3
 - implementation-generated
 - characteristics, 8-1
 - intermixing HP-UX utilities and Ada routines, 11-70
 - 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, 8-17

K

- keyboards, 8-12

L

- LAN, 8-13
- length clauses
 - collection size specification, 4-23, 4-24
 - size specification, 4-1, 4-6, 4-10, 4-11, 4-16, 4-21, 4-22, 4-29, 4-30, 4-32, 4-33, 4-34, 4-49
 - 'SMALL of a fixed point type, 4-18
 - storage, 4-26
 - task activation size specification, 4-26
- length clauses\size specification, 4-3
- libraries
 - HP C, 11-18
 - HP FORTRAN 77, 11-32
- library units, 5-1
- limitations
 - Ada development environment, 10-4
 - compiler, 10-1
 - on pragma PACK, 4-32
 - on record representation clauses, 4-38
 - on the value of 'SMALL, 4-18
 - opening or creating files, 10-5
 - path and component sizes, 10-5
 - user-written applications, 10-5
- line terminator, 8-10
- link editor, 1-6
- LIST, 1-1, 1-11
- local area networks, 8-13
- LOGICAL, 11-36
- long, 11-19

LONG_FLOAT, 3-11, 11-23, 11-39,
11-41, 11-55
long int, 11-19
LONGREAL, 11-55

M

MASK, 8-11
MEMORY_SIZE, 1-18

N

NAME, 8-2
names
 for predefined library units, 5-1
networks, 8-13
NFS, 8-13, 8-14
nonblocking, 8-28
NON_BLOCKING attribute, 8-28
NOT_SHARED, 8-25
NUMERIC_ERROR, 3-13

O

object address location, 11-56
object descriptors, 11-24
object file, 1-6
object library, 1-6
offset, 11-12
OFFSET, 2-1
OPEN, 8-2, 8-3, 8-7, 8-8, 8-19, 8-27
operating system, 8-1, 11-66, 11-67
OPTIMIZE, 1-18
order
 reversed, 11-41
 row major, 11-42
other types, 11-49
OWNER, 8-21

P

PACK, 1-1, 1-12
package
 ASCII, 3-14
 I/O, 8-2, 8-3

STANDARD, 3-4, 3-5, 3-7, 3-8, 3-10,
3-12, 3-14, 3-16
SYSTEM, 3-1, 3-3

PAGE, 1-1, 1-11

page terminator, 8-10

parameter

 count, 11-1
 modification of values, 11-3
 passing conventions, 11-17, 11-32,
 11-33, 11-50, 11-53
 passing modes, 11-1, 11-3, 11-32,
 11-33
 values and bit representations, 11-4
parameter passing conventions, 11-44

Pascal

 access types, 11-56
 array types, 11-56
 Boolean types, 11-54
 character types, 11-54
 enumeration types, 11-54
 fixed point types, 11-55
 floating point types, 11-55
 integer types, 11-52
 record types, 11-62
 scalar types, 11-52
 string types, 11-58
 subprograms, 11-50
 user-defined types corresponding to
 Ada types, 11-53

passing conventions, 1-2

pathname, 8-1, 8-3

pointer, 11-1, 11-6, 11-7, 11-9, 11-19,
11-24

pointer-to-char, 11-20, 11-22

portability, 11-24

POS, 11-36

POSITIVE_COUNT, 8-8, 8-10

pragma

 CONTROLLED, 1-18
 ELABORATE, 1-1, 1-13
 EXPORT, 1-1

EXTERNAL_NAME, 1-1
 implementation-specific, 1-1
 IMPROVE, 1-1, 1-12, 4-1, 4-38, 4-48
 INDENT, 1-1, 1-10
 INLINE, 1-1, 1-14
 INTERFACE, 1-1, 1-2, 11-17, 11-32, 11-50
 INTERFACE_NAME, 1-1, 1-2
 LIST, 1-1, 1-11
 MEMORY_SIZE, 1-18
 OPTIMIZE, 1-18
 PACK, 1-1, 1-12, 4-1, 4-6, 4-11, 4-22, 4-29, 4-30, 4-32, 4-33, 4-34
 PAGE, 1-1, 1-11
 predefined, 1-1
 PRIORITY, 1-1, 1-16
 SHARED, 1-1, 1-17
 STORAGE_UNIT, 1-18
 SUPPRESS, 1-1, 1-15
 SYSTEM.ADDRESS'IMPORT, 2-3
 SYSTEM_NAME, 1-18
 tasking programs, 1-16
 unimplemented, 1-18
 predefined BOOLEAN type, 4-3
 predefined integer types, 4-7
 predefined library units, 5-1
 PRIORITY, 1-1, 1-16
 PROGRAM_ERROR, 3-13
 protection
 external files, 8-11
 parameter, 11-3
 protection of interfaced code from signals, 9-13, 11-69
 PUT, 8-10

R
 random access devices, 8-12
 ranges, 11-25, 11-52
 REAL, 11-39, 11-67
 real types
 bit representation, 11-8

fixed point types, 11-8, 11-23, 11-39, 11-55
 floating point types, 11-8, 11-23, 11-39, 11-55
 general considerations, 11-8
 'RECORD_DESCRIPTOR, 4-47
 RECORD_DESCRIPTOR, 2-1
 record representation clauses, 11-12
 'RECORD_SIZE, 4-45
 RECORD_SIZE, 2-1, 8-32
 record size, 11-13
 record types
 as function results, 11-13, 11-31, 11-48, 11-62
 assignment to a discriminant, 11-13
 bit representation, 11-12
 C language, 11-31
 compiler adds implicit components, 11-13
 compiler errors, 11-4
 components,default size, 4-34, 4-37
 components,minimum size, 4-34
 components reordered by compiler, 11-12
 composite components, 11-13
 direct components, 4-40
 dynamic components, 11-13
 FORTRAN, 11-48
 general considerations, 11-12
 implicit component
 'ARRAY_DESCRIPTOR, 4-47
 implicit component 'OFFSET, 4-44
 implicit component
 'RECORD_DESCRIPTOR, 4-47
 implicit component 'RECORD_SIZE, 4-45
 implicit components, 4-38
 implicit component
 'VARIANT_INDEX, 4-45
 indirect components, 4-40

Index-8

- layout, 1-12, 4-34, 4-37, 11-12
- offset, 11-12
- parameter passing modes, 11-3
- Pascal, 11-62
- passing to external subprograms, 11-12, 11-62
- record representation clauses, 11-12
- size of record, 11-13
- syntax, 4-34
- unconstrained, 7-1
- RECORD_UNIT, 8-32
- reformatter, 1-10
- representation clauses
 - data objects, 4-1
 - enumeration, 4-1, 4-3
 - enumeration types, 11-6
 - pragma PACK, 1-12
 - record, 4-1, 4-6, 4-11, 4-22, 4-34, 4-36, 4-37, 4-44, 4-45, 4-47, 4-48, 4-50
- reserved signal, 11-69
- restrictions, 7-1, 8-3, 10-1, 11-23, 11-37
- RESUME_ADA_TASKING, 11-69
- return result pointer, 11-1
- reverse order, 11-41
- row major order, 11-42
- runtime, 11-66, 11-69
- run-time checks, 11-3
- run-time system, 1-6

S

- scalar types
 - Boolean types, 11-7, 11-36, 11-54
 - calling FORTRAN, 11-32
 - calling Pascal, 11-54
 - character types, 11-7
 - enumeration types, 11-6, 11-54
 - FORTRAN, 11-33
 - general considerations, 11-5
 - integer types, 11-6, 11-33, 11-52
 - Pascal, 11-52
 - real types, 11-8, 11-39, 11-55
 - string types, 11-44, 11-46
- scheduling, 11-67
- sequential access, 8-33
- SEQUENTIAL_IO files, 8-7, 8-32
- SHARED, 1-1, 1-17, 8-25
- short, 11-19
- short int, 11-19
- SHORT_INTEGER, 3-8, 4-11, 11-6, 11-19, 11-36, 11-41, 11-53
- SHORT_SHORT_INTEGER, 3-7, 11-6, 11-19, 11-25, 11-53
- SIGALRM, 9-13, 11-67, 11-69
- SIGBUS signal, 9-6
- SIGFPE signal, 9-7
- SIGHUP, 9-10
- SIGILL signal, 9-7
- SIGINT, 9-10
- signals
 - asynchronous, 9-13
 - blocked, 11-69
 - HP-UX, 9-1
 - interruption, 11-67
 - subprograms, 11-66
- SIGPIPE, 9-10
- SIGQUIT, 9-10
- SIGSEGV signal, 9-7
- SIGTERM, 9-10
- sigvector(2), 11-66, 11-67
- SIGVTALRM, 9-13, 11-67, 11-69
- size of record, 11-13
- slow HP-UX routines, 11-67
- stack, 11-18, 11-22, 11-24
- STANDARD, 3-4, 3-5, 3-7, 3-8, 3-10, 3-12, 3-14, 3-16
- STANDARD_INPUT, 8-12
- STANDARD_OUTPUT, 8-12
- stderr, 8-12, 11-70
- stdin, 8-12, 11-70
- stdout, 8-12, 11-70
- STORAGE_ERROR, 3-13
- STORAGE_UNIT

- pragma, 1-18
- record representation clause, 4-37
- STRING, 3-13
- string types
 - Pascal, 11-58
 - passing to external subprograms, 11-11, 11-44
 - scalar types, 11-46
 - special case of arrays, 11-25
- subprograms
 - predefined, 8-10
- SUPPRESS, 1-1, 1-15
- SUSPEND_ADA_TASKING, 11-69
- symbolic links, 8-16
- SYSTEM, 3-1, 3-3, 11-33
- system dependencies, 8-17
- SYSTEM_NAME, 1-18

T

- tapes, 8-12
- task
 - management, 9-1, 9-8
 - minimum size of stack, 4-28
 - rescheduling, 11-67
 - scheduling, 11-67
 - time slice amount, 11-67
- TASKING_ERROR, 3-13
- task types
 - alignment, 4-28
 - as function results, 11-13
 - external subprograms, 11-2
 - general considerations, 11-13
 - internal representation, 4-26
 - minimum size, 4-28
 - passing to external subprograms, 11-13
- terminal input, 8-29
- terminator
 - character, 11-25

- file, 8-10
- line, 8-10
- page, 8-10
- representation, 8-30
- text files, 8-30
- TEXT_IO files, 8-10
- text processing tools, 1-10
- time slicing, 11-67
- type conversions, unchecked, 7-1
- type representation, 4-1

U

- UNCHECKED_CONVERSION
 - access types, 4-23
 - enumeration types, 4-3
 - interfacing with external data structures, 4-23
 - limitations, 7-1
- UNCHECKED_DEALLOCATION, 8-13
- unchecked type conversions, 7-1
- unconstrained record types, 7-1
- unconstrained strings, 11-25
- underlying object, 11-56
- universal_fixed, 3-7
- universal_integer, 3-5
- universal types, 3-5
- USE_ERROR, 8-3, 8-8, 8-20, 8-25
- user-written applications, 10-5

V

- VAR, 11-56, 11-62
- 'VARIANT_INDEX, 4-45
- VARIANT_INDEX, 2-1
- variant index, 11-13

W

- WORLD, 8-21