HP Pascal/iX Reference Manual

HP 3000 MPE/iX Computer Systems

Edition 5

Manufacturing Part Number: 31502-90022
E0692

U.S.A. June 1992

Notice

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 or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing 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.

Restricted Rights Legend

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

Acknowledgments

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Preface

Printing History

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition or a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software code printed alongside the date indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

First Edition	March 1987	MPE XL: 31502A.01.01 HP-UX: 92431A.00.03
Update 1	August 1987	MPE XL: 31502A.01.03 HP-UX: 92431A.01.07
Second Edition	November 1987	MPE XL: 31502A.01.06 HP-UX: 92431A.01.09
Update 1	January 1988	MPE XL: 31205A.01.06 HP-UX: 92431A.01.12
Third Edition	October 1988	MPE XL: 31502A.01.21 HP-UX: 92431A.03.04
Fourth Edition	January 1991	MPE XL: 31502A.03.10 HP-UX:92431A.08.00
Fifth Edition	June 1992	MPE/iX: 31502A.04.05 HP-UX:92431A.09.00

Preface

MPE/iX, Multiprogramming Executive with Integrated POSIX, is the latest in a series of forward-compatible operating systems for the HP 3000 line of computers.

In HP documentation and in talking with HP 3000 users, you will encounter references to MPE XL, the direct predecessor of MPE/iX. MPE/iX is a superset of MPE XL. All programs written for MPE XL will run without change under MPE/iX. You can continue to use MPE XL system documentation, although it may not refer to features added to the operating system to support POSIX (for example, hierarchical directories).

Finally, you may encounter references to MPE V, which is the operating system for HP 3000s, not based on the PA-RISC architecture. MPE V software can be run on the PA-RISC (Series 900) HP 3000s in what is known as *compatibility mode*.

The *HP Pascal/iX Reference Manual* provides material about HP Pascal and its system programming extensions. It is intended for experienced Pascal programmers.

This manual is organized as follows:

Chapter 1 Introduces HP Pascal. A summary of extensions to ANSI/IEEE 770 X3.97-1983 and ISO 7185-1983 standard Pascal is included.

Chapter 2 Describes the language elements in HP Pascal.

Chapter 3 Describes HP Pascal's data types.

Chapter 4 Defines the expressions used in HP Pascal.

Chapter 5 Describes the parts of the declaration section in HP Pascal.

Chapter 6 Discusses the statements used in HP Pascal.

Chapter 7 Describes the program structure used in HP Pascal.

Chapter 8 Defines HP Pascal's procedures and functions.

Chapter 9 Defines the predefined routines used in HP Pascal.

Chapter 10 Explains input and output as used in HP Pascal.

Chapter 11 Defines the system programming extensions to HP Pascal.

Chapter 12 Explains every compiler option used in HP Pascal.

Appendix A Describes the error messages, notes, and warnings in HP Pascal.

Appendix B Defines the ASCII character set.

Appendix C Defines the compiler's limits and default values.

If you have suggestions for improving the *HP Pascal/iX Reference Manual*, please send us the Reader Comment Card, which is located at the front of this manual.

Additional Documentation

Additional information for the HP Pascal programmer can be found in the following documents:

* *IEEE Standard Pascal Computer Programming Language*, ANSI/IEEE 770 X3.97-1983, Library of Congress Catalog Number 82-84259. This book defines the ANSI standard Pascal that is the basis for HP Pascal.

* *HP Pascal/iX Programmer's Guide*, part number 31502-90002. This book explains HP Pascal topics in detail. It describes how statements interact with each other, if necessary. It does not explain every statement and feature of HP Pascal.

This manual also refers to the following manuals:

* *HP C Programmer's Guide* (92434-90002)

* *HP Link Editor/XL Reference Manual* (32650-90030)

* *ALLBASE/SQL Pascal Application Programming Guide* (36216-90007)

* *HP System Dictionary/XL General Reference Manual* (32256-90004)

* *HP TOOLSET/XL Reference Manual* (36044-90001)

* *Introduction to MPE XL for MPE V Programmers* (30367-90005)

* *MPE/iX Commands Reference Manual, Volumes 1 and 2* (32650-90003 and 32650-90364)

* *MPE/iX Intrinsics Reference Manual* (32650-90028)

* *MPE/iX Symbolic Debugger User's Guide* (31508-90003)

* *MPE/iX System Debug Reference Manual* (32650-90013)

* *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* (09740-90039)

* *Procedure Calling Conventions Reference Manual* (09740-90015)

* *TurboIMAGE/XL Reference Manual* (30391-90001)

* *Using VPLUS/V: Introduction to Forms Designs* (32209-90004)

Conventions

UPPERCASE — In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either upper or lowercase. For example:

```
COMMAND
```

can be entered as any of the following:

```
command       Command       COMMAND
```

It cannot, however, be entered as:

```
comm          com_mand      comamnd
```

italics — In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with the actual value. In the following example, you must replace *FileName* with the name of the file:

```
COMMAND FileName
```

punctuation — In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

```
(FileName ):(FileName )
```

{ } — In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either ON or OFF:

```
COMMAND {ON }
        {OFF}
```

[ ] — In a syntax statement, brackets enclose optional elements. In the following example, OPTION can be omitted:

```
COMMAND FileName  [OPTION]
```

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select OPTION or *Parameter* or neither. The elements cannot be repeated.

```
COMMAND FileName  [OPTION    ]
                  [Parameter ]
```

Conventions (continued)

`[...]` In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *Parameter* zero or more times. Each instance of *Parameter* must be preceded by a comma:

```
[,Parameter ][...]
```

In the example below, you only use the comma as a delimiter if *Parameter* is repeated; no comma is used before the first occurrence of *Parameter*:

```
[Parameter ][,...]
```

`|...|` In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select A, AB, BA, or B. The elements cannot be repeated.

```
{A} |...|
{B}
```

`...` In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.

triangle In a syntax statement, the space symbol triangle shows a required blank. In the following example, *Parameter* and *Parameter* must be separated with a blank:

```
(Parameter ) triangle (Parameter )
```

The symbol indicates a key on the keyboard. For example, RETURN represents the carriage return key.

base prefixes The prefixes %, #, and $ specify the numerical base of the value that follows:

%num	specifies an octal number.
#num	specifies a decimal number.
$*num*	specifies a hexadecimal number.

If no base is specified, decimal is assumed.

Pascal Specific Conventions

The conventions followed in this manual are summarized below:

For Text:

* The term PAC is used for the type PACKED ARRAY OF CHAR with the lower bound equal to 1.

* Reserved words and directives are in all uppercase letters.

  Examples: BEGIN, REPEAT, FORWARD

* Standard identifiers are in all lowercase letters.

  Examples: readln, maxint, text

* General information concerning an area of programming (topic) appears as a heading with initial capitalization. All headings that are not reserved words or standard identifiers appear with initial capitalization.

For Syntax Diagrams:

* Syntactic entities that are to be replaced by user-supplied entities are represented by sequences of lowercase letters and embedded underscore characters (_).

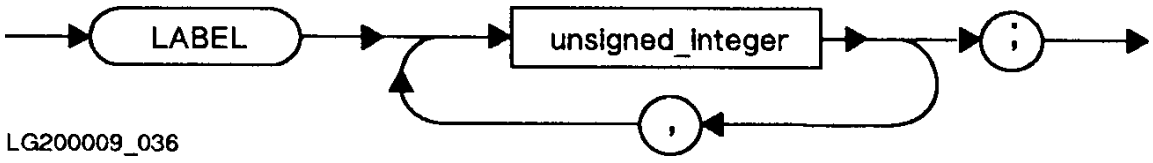
  Example: identifier

* Keywords, predefined symbolic names and special symbols that must be supplied exactly as given are shown in apostrophes. Usually, letters may be entered in uppercase or lowercase.

  Example: 'IMPORT', ','

* The diagrams are in the form of lines with directional arrows, known as "railroad tracks". Alternative paths are indicated by switches in the tracks.

  Example:



NOTE Some diagrams and tables have a number in the lower left or right corner, such as the number LG200009_036 in the diagram above. This number is not part of the diagram or table. It just identifies the artwork.

Chapter 1 Introduction

HP Pascal originates from the Pascal language developed by Nicklaus Wirth in 1968. Wirth's Pascal is based on the ALGOL 60 programming language. His objective was to introduce Computer Science students to "good programming practices." Since then, Pascal has undergone extensions, particularly in its input-output capabilities. This has helped it become a dominant language not only in the academic world, but also in major commercial software projects. Commercial attraction for Pascal stems from its structured nature that makes Pascal programs readable and self documenting. Because maintenance typically forms a large portion of software costs, the structuring is an attractive feature, particularly for large systems and subsystems.

Although Pascal differs from vendor to vendor, it is easy to program for portability by conforming to a reasonably large and effective subset of Pascal that is standard across several vendors. The standardization is achieved as a result of the ANSI/IEEE 770 X3.97-1983 and ISO 7185-1983 standards that exist for Pascal today. HP Pascal is a superset of these standards. It is based on HP's standard for the Pascal language.

The Pascal on the HP Precision Architecture Series of Computer Systems includes system programming extensions to the HP Pascal standard. These extensions have lead to widespread use of Pascal within HP for systems level applications. This trend is also expected to be observed by our customers. In addition to its past usages, Pascal may be used for applications traditionally written in Assembly or SPL. These applications will have a higher degree of portability across HP systems in the future.

This chapter is divided into several sections. The first section covers the conventions used in this manual. This is followed by a discussion about the HP Pascal Extensions to the ANSI/IEEE 770 X3.97-1983 and ISO 7185-1983 standards for Pascal.

Extensions to ANSI/IEEE and ISO Pascal

This section describes HP Pascal features that are extensions of ANSI/IEEE 770 X3.97-1983 and ISO 7185-1983 Pascal. For the full description of a feature, refer to the appropriate keyword or topic in this manual.

Type Compatibility

NOTE In the ISO 7185-1983 or ANSI/IEEE 770 X3.97-1983 standards for Pascal, the term "string" refers to any PACKED ARRAY of CHAR with a starting index of 1. HP Pascal, however, supports the standard type string. To avoid confusion, the term PAC is used for the type PACKED ARRAY [1..n] of CHAR with a starting index of 1.

Pascal defines a set of compatibility requirements for the operands of each operator, based both on the operator itself and the types of its operands, and a set of assignment compatibility rules. HP Pascal extends

the operator and assignment compatibility rules as follows:

* If T1 and T2 are PAC variables or string literals they are compatible. The shorter is padded with blanks for comparison.

* If T1 is a PAC variable and T2 is a string literal or PAC variable, then T2 is assignment compatible with T1 provided that T2 is shorter than or equal to T1. If T2 is shorter than T1, T2 is padded with blanks.

CASE Statement.

In a CASE statement, the reserved word OTHERWISE may precede a list of statements and the reserved word END. If the case selector evaluates to a value not specified in the case constant list, the system executes the statements between OTHERWISE and END. OTHERWISE must follow the last case constant. Also, subranges may appear as case constants.

Compiler Options.

Compiler options appear between dollar signs ($). HP Pascal has two categories of compiler options: system-independent and system-dependent compiler options. The system-independent category of compiler options are further distinguished by the following categories: HP Standard Options, HP Pascal Options, and System Programming Options. The system-dependent either work on only one operating system, or work differently on HP-UX and MPE/iX.

HP Pascal options are not required by the HP Standard, but are available in HP Pascal. An HP Pascal program containing HP Pascal options must be compiled by the HP Pascal compiler.

System-Independent Compiler Options:

HP Pascal Options

ALIAS
ALIGNMENT
ARG_RELOCATION
ASSERT_HALT
ASSUME
BUILDINT
CHECK_ACTUAL_PARM
CHECK_FORMAL_PARM
CODE
CODE_OFFSETS
COPYRIGHT
COPYRIGHT_DATE
ELSE
ENDIF
EXTERNAL
EXTNADDR
GLOBAL
HEAP_COMPACT
HEAP_DISPOSE
IF
INLINE
INTR_NAME
KEEPASMB
LIST_CODE
LISTINTR
LITERAL_ALIAS
LOCALITY
LONG_CALLS
MAPINFO
MLIBRARY
NOTES
OPTIMIZE
OS
OVFLCHECK
PAGEWIDTH
POP
PUSH
S300_EXTNAMES
SEARCH
SET
SKIP_TEXT
SPLINTR
STATEMENT_NUMBER
STDPASCAL_WARN
STRINGTEMPLIMIT
SUBPROGRAM
SYSINTR
SYSPROG
TABLES
TITLE
TYPE_COERCION
UPPERCASE
VERSION
VOLATILE
WARN
WIDTH
XREF

Table 1-0. (cont.)

HP Standard Options	System Programming Options
ANSI	EXTNADDR
LINES	TYPE_COERCION
LIST	
PAGE	
PARTIAL_EVAL	
RANGE	
STANDARD_LEVEL	

System-Dependent Compiler Options:

MPE/iX Only	MPE/iX and HP-UX	HP-UX Only
CALL_PRIVILEGE	INCLUDE	CONVERT_MPE_NAMES[REV BEG]
EXEC_PRIVILEGE	INCLUDE_SEARCH	GPROF
FONT	NLS_SOURCE	HP_DESTINATION
HP3000_16	SYMDEBUG	SHLIB_CODE[REV END]
HP3000_32		SHLIB_VERSION
RLFILE		
RLINIT		

Refer to Chapter 12 for details about these options.

Conformant Array Parameters.

The ISO Level 1 Conformant Array Parameter feature is implemented in HP Standard Pascal. This is the only feature in ISO Pascal that is not in ANSI/IEEE Pascal.

This feature allows the user to pass an array as a parameter, whose bounds are determined at run time and which conforms to the conformant array parameter specification. The specification includes the names of the array bounds. The values of the bounds of the actual array are given when it is passed.

Constant Expressions.

The value of a declared constant may be specified with a constant expression. A constant expression returns an ordinal or real value and can contain only declared constants, literals, calls to the functions ord, chr, pred, succ, hex, octal, binary, strlen, odd, and the operators +, -, *, DIV, and MOD. Note that a constant expression can appear anywhere that a constant can appear.

Constructors (Structured Constants).

The value of a declared constant can be specified with a constructor. A constructor establishes values for the components of a previously declared structured type. Constructors can only appear in a CONST section of a declaration part of a block. Set constructors can appear either in a CONST section or in expressions in executable statements.

Declaration Part.

In the declaration part of a block, CONST, TYPE, VAR, MODULE, and IMPORT sections can be repeated and intermixed.

Halt Procedure.

The *halt* procedure causes an abnormal termination of a program.

Heap Procedures.

The procedure *mark* saves the allocation state of the heap. The procedure *release* restores the allocation state of the heap to a state previously marked. This has the effect of deallocating all storage allocated by the procedure *new* since the time *mark* was called.

Identifiers.

The underscore character (_) can appear in identifiers, but not as the first character.

File Input/Output.

A file can be opened for direct access with the procedure *open*. Direct access files have a maximum number of components indicated by the function *maxpos* and have a current number of written components, indicated by the function *lastpos*. The procedure *seek* places the current position of a direct access file at a specified component. Data can be read from a direct access file or written to it with the procedures *readdir* or *writedir* that are combinations of *seek* and the standard procedures *read* or *write*. A textfile cannot be used as a direct access file.

A file can be opened in the "write-only" state without altering its contents by using the procedure *append*. The current position is set to the end of the file.

Any file can be explicitly closed with the procedure *close*.

To permit interactive input, the system defines the primitive file operation *get* as "deferred get." Refer to *get* in Chapter 10 for more information.

The procedure *read* accepts any ordinal type as input from text files. Therefore, it is possible to read a Boolean or enumerated value from a text file. It is also possible to read a value that is of type PAC or string.

The procedure *write* writes expressions to a text file. Any ordinal type can be a parameter. An enumerated constant can be written directly to a text file. *Write* also writes expressions of type string or PAC.

The function *position* returns the index of the current position for any file that is not a textfile.

The routines *page*, *overprint*, *prompt*, and *linepos* operate on textfiles. The following lists what each routine does:

* *Linepos* returns the integer number of characters that the program has read from or written to a textfile since the last end-of-line marker.
* *Page* causes a page eject when a text file is printed.
* *Overprint* causes the printer to perform a carriage return without a line feed, effectively overprinting a line.

* *Prompt* displays the output buffer without writing a line marker. This allows the cursor to remain on the same screen line when output is directed to a terminal.

The routine *associate* allows Pascal input/output operations on files that have been opened by the operating system. The routine *disassociate* disallows these operations.

Function Return.

A function can return any structured type, except those containing files. That is, a function may return an *array, record, set,* or *string*.

Longreal Numbers.

The type *longreal* is identical to the type real except that it provides greater precision. The letter "L" precedes the scale factor in a *longreal literal*.

Minint.

The standard constant *minint* is defined in HP Pascal. The value is implementation dependent. The type integer is defined as a subrange *minint...maxint*. *Minint* is less than or equal to *maxint.*

Formal Parameter Congruency.

Two formal parameter lists are congruent if they contain an equal number of parameters and each parameter in one list is equivalent to the parameter in the same position in the other list. The formal parameter lists do not need to be syntactically the same.

Record Variant Declaration.

The variant part of a record field list may have a subrange as a case constant and need not specify all the case constants for the tag type.

String or Character Literals.

HP Pascal permits the encoding of control characters or any other single ASCII character after the sharp symbol (#). For example, the string literal #G represents CTRL-G (or the bell). A character can also be encoded by specifying its ASCII ordinal value (0..255) after the sharp symbol. For example, #7 represents CTRL-G. These characters can be included in string literals by directly appending them in front of or behind a string literal.

String Type.

HP Pascal supports the predefined type string. A string type is a PACKED ARRAY of CHAR with a declared maximum length and an actual length that may vary at run time. All HP Pascal implementations have maximum lengths of at least 255 characters.

A variable of type string can be compared with a similar variable or a string literal or can be assigned to a variable of type string. A string literal can be assigned to a variable of type string.

The following standard procedures and functions manipulate strings:

* *Setstrlen* sets the current length of a string without changing its contents.
* *Str* returns a specified portion of a string, such as a substring.
* *Strappend* appends one string to another.
* *Strdelete* deletes a specified number of characters from a string.
* *Strinsert* inserts one string into another.
* *Strlen* returns the current length of a string.
* *Strltrim* and *strrtrim* trim leading and trailing blanks, respectively, from a string.
* *Strmax* returns the maximum length of a string.
* *Strmove* copies a substring from a source string to a destination string.
* *Strpos* returns the position of the first occurrence of a specified string within another string.
* *Strread* reads one or more values from a string.
* *Strrpt* returns a string composed of a designated string repeated a specified number of times.
* *Strwrite* writes one or more values to a string.

WITH Statement.

The record designator in a WITH statement can be a call to a function that returns a record as its result, or a structured constant.

Numeric Conversion Functions.

The functions *binary, octal,* and *hex* convert a parameter of type string or PAC, or a string literal, to an integer. These functions interpret the parameter the following ways:

* Binary interprets the parameter as a binary value.
* Octal interprets the parameter as an octal value.
* Hex interprets the parameter as a hexadecimal value.

Modules.

HP Pascal supports separately compiled program fragments called *modules*. Modules can be used to satisfy the unresolved references of another program or module. Typically, a module "exports" types, constants, variables, procedures, and functions. A program can then "import" a module to satisfy its own references.

This mechanism allows commonly used procedures and functions to be compiled separately and used by more than one program without having to include them in each program.

Chapter 2 Language Elements

A Pascal program is a sequence of statements that, when executed in a specified order, processes data to produce desired results. The elements of Pascal include basic *symbols, reserved words, identifiers, numbers, comments, separators,* and *literals*. The statements are made up of different elements depending on the needs of the program.

This chapter describes in detail the elements of statements in the HP Pascal language.

Basic Symbols

The *basic symbols* consist of letters, digits, and special symbols. The letters include A..Z and a..z. The digits are 0 through 9. Table 2-1 (*) lists the special symbols that are valid in HP Pascal.

Table 2-1. Special Symbols

Symbol	Description
+	Add, set union, concatenate strings, unary plus +.
-	Subtract, set difference, unary minus -.
*	Multiply, set intersection.
/	Divide (real results).
=	Equal to, type identifier.
<	Less than.
>	Greater than.
( )	Delimit a parameter list or a expression.
[ ]	Delimit an array or string index, set, or a constructor. May be replaced by the (. .) pair.
.	Select record field, decimal point.
,	Separate listed identifiers, values, or variables.
;	Separates statements and formal parameters.

Table 2-1. Special Symbols (cont.)

Symbol	Description
:	Denotes a statement label, list of case constants, or variable identifiers.
^	Define or dereference pointers, access file buffer. May be replaced by @.
<>	Not equal.
<=	Less than or equal, subset.
>=	Greater than or equal, superset.
:=	Assign value to a variable.
..	Delimit a subrange.
{ }	Delimit a comment. May be replaced by the (* *) pair.
#	Encode a control character.
$	Delimit a compiler option.
'	Delimit a string literal.
_	May appear within an identifier.

Reserved Words

Reserved words are symbols that have special meaning to the Pascal language. They are the names of statements, data types, or operators. A reserved word can be used in a program only in the context for which it is defined. A reserved word cannot be redefined for use as an identifier. It may, however, be used within comments or string literals.

A list of reserved words recognized by HP Pascal with a brief description of each is given in Table 2-2 . A more detailed description of some of the reserved words follows in this chapter. In some cases, a detailed description is presented elsewhere in this manual. Table 2-2 provides the location of these instances by word and chapter.

NOTE At the ANSI and ISO standard level, OTHERWISE, IMPORT, EXPORT, IMPLEMENT, and MODULE are not considered reserved words. The compiler option STANDARD_LEVEL controls whether these identifiers are recognized as reserved words. Refer to Chapter 12 for more information about STANDARD_LEVEL. If the system programming extensions are enabled, additional identifiers may be treated as reserved words.

Table 2-2. Reserved Words

Reserved Word(s)	Description	Chapter Reference
AND	Boolean conjunction operator.	4
ARRAY, OF	A structured type.	3
BEGIN...END	Delimit a compound statement or BLOCK.	6
CASE...OF...OTHERWISE...END	A conditional statement.	6
CONST	Begins constant definition section.	5
DIV	Integer division operator.	4
EXPORT	Begins module export section.	7
FILE...OF	Structured type.	3
FOR...TO...DOWNTO...DO	Repetitive statement.	6
FUNCTION	Begins a function declaration.	7
GOTO	Control transfer statement.	6
IF...THEN...ELSE	Conditional statement.	6
IMPLEMENT	Begins module implement section.	7

IMPORT	Begins module import section.	7
IN	Set inclusion operator.	4
LABEL	Begins label definition section.	5
MOD	Integer modulus operator.	4

Table 2-2. Reserved Words (cont.)

Reserved Word(s)	Description	Chapter Reference
MODULE	Begins a module declaration.	7
NIL	Special pointer value.	5
NOT	Boolean negation operator.	4
OR	Boolean disjunction operator.	4
PACKED	Controls allocation for structured type.	3
PROCEDURE	Begins a procedure declaration.	7
PROGRAM	Program heading.	7
RECORD...CASE...OF...END	Structured type.	3
REPEAT...UNTIL	Repetitive statement.	6
SET...OF	Structured type.	3
TYPE	Begins a type definition section.	5
VAR	Begins a variable declaration section.	5
WHILE...DO	Repetitive statement.	6

WITH...DO	Opens record scopes.	6	

Identifiers

An HP Pascal *identifier* consists of a letter preceding an optional character sequence of letters, digits, or the underscore character (_) up to a source line in length with all characters significant without respect to case.

Identifiers are used to denote declared constants, types, variables, procedures, functions, modules, and programs.

A letter may be any of the letters in the subranges A through Z or a through z. The compiler makes no distinction between upper and lower case in identifiers. A digit may be any of the digits 0 through 9. The underscore (_) is an HP Standard Pascal extension of ANSI/IEEE770X3.97 - 1983 Standard Pascal.

In general, an identifier must be defined before using it. Four exceptions are:

* Identifiers that define pointer types and are themselves defined later in the same declaration part.
* Identifiers that appear as program parameters and are declared subsequently as variables.
* Predefined identifiers such as integer and char.
* Forward procedures or functions.

An identifier does not need to be defined when it is a program, module, procedure, or function name, or one of the identifiers defining an enumerated type. Its initial appearance in a function, procedure, module, or program header is the "defining occurrence."

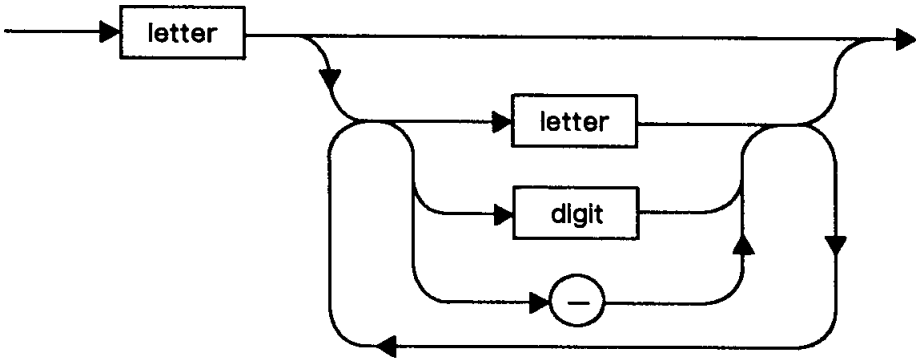
Finally, HP Pascal has a number of standard identifiers that may be redeclared. These standard identifiers include names of standard procedures and functions, standard file variables, standard types, standard constants, and procedure or function directives.

Reserved words are language defined symbols whose meaning can never change. Therefore, an identifier cannot be declared that has the same spelling as a reserved word.

For a list of reserved words recognized by HP Pascal, see Table 2-2 .

Syntax

Identifier:



Example

```
GOOD_TIME_9       { These identifiers   }
good_time_9       { are                 }
gOOd_TIme_9       { equivalent.         }

x2_GO
a_long_identifier
Boolean           { Standard identifier.}
```

Scope

The *scope* of an identifier is its domain of accessibility or the region of a program in which it may be used. In general, a user-defined identifier can appear anywhere in a block after its definition. Furthermore, the identifier can appear in a block nested within the block in which it is defined.

If an identifier is redefined in a nested block, however, this new definition takes precedence in the entire block. The object defined at the outer level is no longer accessible from the inner level. Once defined at a particular level, an identifier may not be redefined at the same level, except for field names.

Labels are not identifiers, and their scope is restricted. They cannot mark statements in blocks nested within the block where they are declared.

Identifiers defined at the main program level are *global*. Identifiers defined in a function or procedure block are *local* to the function or procedure. The definition of an identifier must precede its use, with the exception of pointer type identifiers, program parameters, predefined identifiers, and forward declared procedures or functions.

For a module, identifiers declared in the EXPORT section are valid for the entire module. Identifiers declared after the IMPLEMENT keyword are valid only within the IMPLEMENT part of the module.

When a module is imported, the identifiers in the EXPORT section of the imported module are placed in the *global scope* of the program. Because of this, the identifiers in the EXPORT section must be unique not only within the module, but also within the *global scope* of a program.

Example

```
PROGRAM show_scope (output);

CONST
   asterisk = '*';

VAR
   x: char;       {global variable}
```

```
PROCEDURE writeit;

CONST
   x = 'LOCAL AND GLOBAL IDENTIFIERS DO NOT CONFLICT';

BEGIN
   write (x)
END;     {writeit}

BEGIN { show_scope }
   x:= asterisk;
   write (x);
   writeit;
   write (x);
   writeln;
END.  { show_scope }

RESULTS:

     *LOCAL AND GLOBAL IDENTIFIERS DO NOT CONFLICT*
```

Numbers

HP Pascal recognizes three kinds of numeric literals: *integer, real,* and *longreal*.

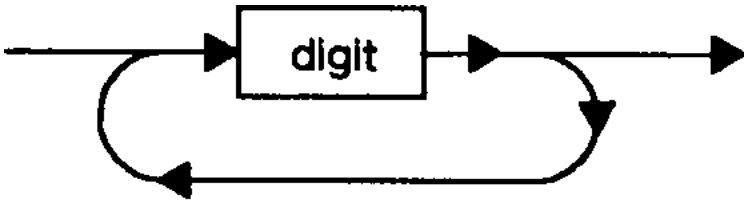
Integer Literals

An *integer literal* consists of a sequence of digits from the subrange 0 through 9. No spaces may separate the literal, and leading zeroes are not significant. The compiler interprets unsigned integer literals as positive values.

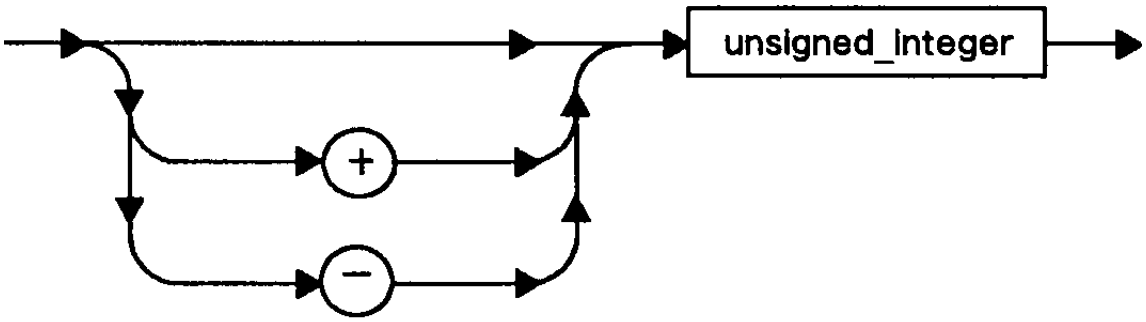
The maximum *unsigned integer literal* is equal in value to the standard constant *maxint*. The minimum *signed integer literal* is equal in value to the standard constant *minint*. The actual values of *minint* and *maxint* are implementation dependent; however, at least 9 decimal digits are allowed. Refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information.

Syntax

Unsigned Integer:



Signed Integer:



Example

```
 100                    { unsigned integer    }
-100                    { signed integer      }
```

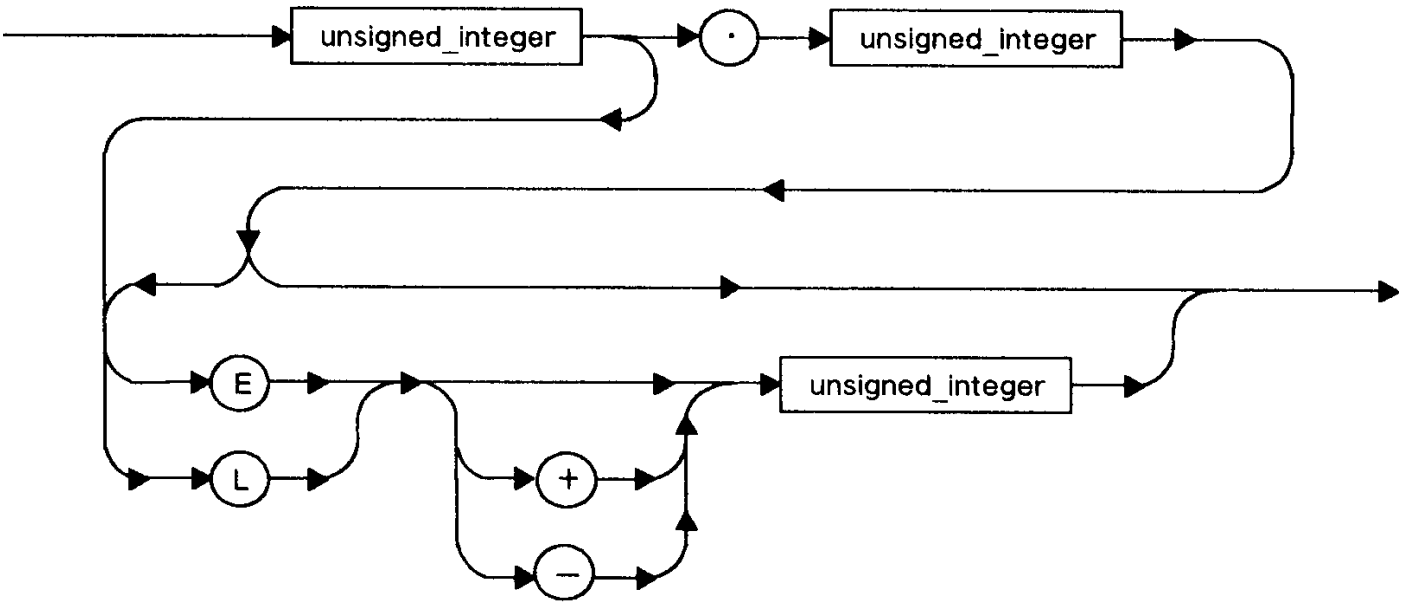
Real and Longreal Literals

A *real* or *longreal literal* consists of a coefficient and a scale factor. An E preceding the scale factor is read as times ten to the power of and specifies a *real literal*. An L preceding scale factor also means times ten to the power of, but specifies a *longreal literal*.

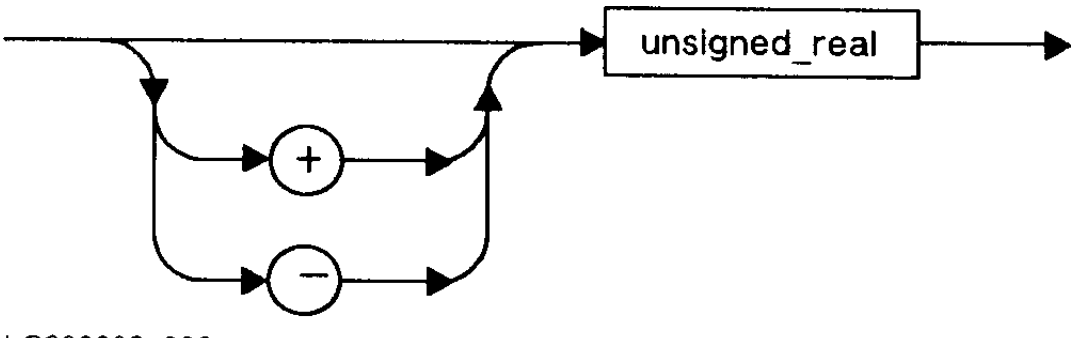
Lowercase e and l are legal. At least one digit must precede and follow a decimal point. A number containing a decimal point and no scale factor is considered a *real literal*.

Syntax

Unsigned Real:



Signed Real:



Example

```
0.1                 { Real with no scale factor.  }
5E-3                { Real with no decimal point. }
3.14159265358979L0  { Longreal.                   }
87.35e+8            { Real.                       }
```

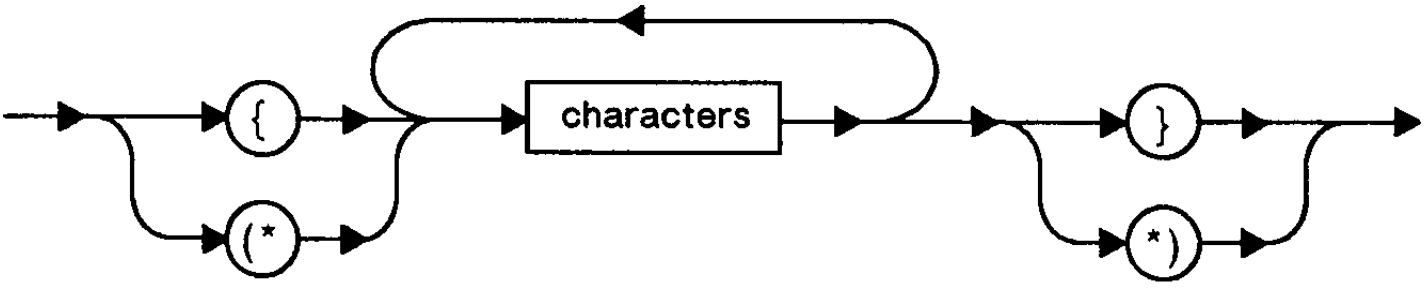
Comments

Comments consist of a sequence of characters that starts with either of the equivalent symbols { or (*, and end with either of the equivalent symbols } or *).

Comments are used to document a program. Since a comment is a *separator*, it may appear anywhere in a program where a *separator* may appear. However, nested comments are not legal. Note that comments do not have to be on lines by themselves and may cross line boundaries.

Syntax

Comment:



Example

```
{ comment }
(*comment*)
{ comment*)
{ { { { comment }
{ This comment
occupies more than one line. }
```

Separators

A *separator* is a space, a tab, an end-of-line marker, a compiler option, or a comment. Separators are used to separate reserved words, identifiers, numbers, strings, and special symbols. At least one separator must appear between any pair of consecutive identifiers, numbers, or reserved words. When one or both elements are special symbols, however, the separator is optional.

Separators may not appear within special symbols having more than one component (:=, for example). Certain special symbols have synonyms. In particular, (. and .) may replace the left and right brackets, [and]. The symbol @ may substitute for the up-arrow ^, also (* and *) may take the place of the left and right braces, { and }.

Example

```
IF EOF THEN GOTO 99          { Required separators. }
   x := x + 1                { Optional separators. }
   x:=x+1                    { No separators.       }
```

String Literals

String literals are sequences of characters, enclosed by single quote marks, that may not be longer than a single line of source code. String literals may consist of any combination of the following:

* A sequence of ASCII characters enclosed in single quote marks.

* A sharp symbol (#) followed by a single character.

* A sharp symbol (#) followed by up to three digits that represent the ASCII value of a character.

A letter or symbol after a sharp symbol is equivalent to a control character. For example, #G or #g encodes CTRL-G, the bell character. The compiler interprets the letter or symbol according to the expression chr(ord(letter) MOD 32). Therefore, the ordinal value of G is 71; modulus 32 of 71 is 7; and the ASCII value of 7 is the bell.

In a string literal, if a number is used after a sharp symbol, it may contain up to three digits, but must be in the range 0 through 255. It directly encodes any printing or nonprinting ASCII character. For example, the string literal #80#65#83#67#65# 76 is equivalent to the string literal PASCAL.

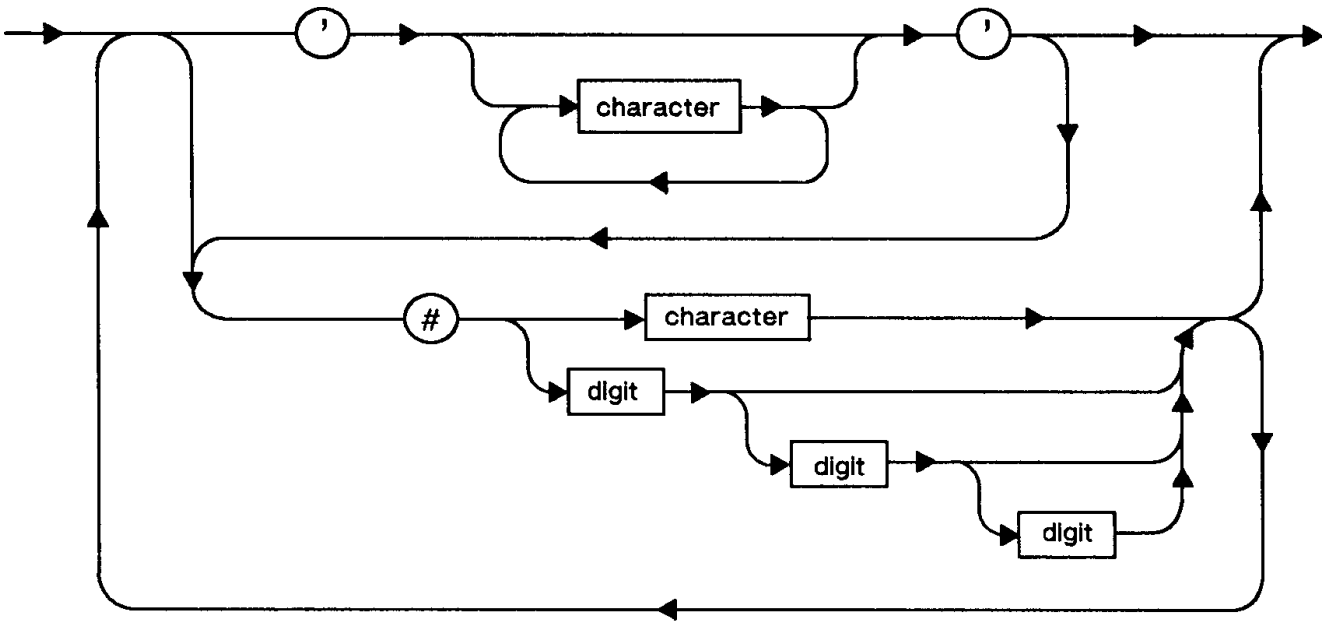
Any ASCII character can appear between quote marks. The sharp symbol # is provided to enable better documentation of nonprinting characters.

A string literal may be type char, PAC, or string. This is dependent on the context in which it is used. If a single quote is a character in a string literal, it must appear twice, consecutively.

Two consecutive quote marks ('') are used to specify the null or empty string literal. Assigning this value to a string variable sets the length of the variable to zero. Assigning this value to a PAC variable blank-fills the variable.

Syntax

```
String_literal:
```



Example

```
'Please don''t!'                    { Single quote character. }
'A'
''                                  { Null string.           }
#F
#243#H
#27'that was an ESC char, and so is this'#[
'this string has five bells'#G#g#g#7#7' in it'
```

Chapter 3 Data Types

One of the most important contributions and fundamental ideas of Pascal is the formalization of the concept of a *data type*. A data type is a collection of elements that belong together because they are all formed in the same way and are treated uniformly.

There are three categories of data types in HP Pascal. They are:

* Simple
* Structured
* Pointer

These data types are used to determine a set of attributes that include:

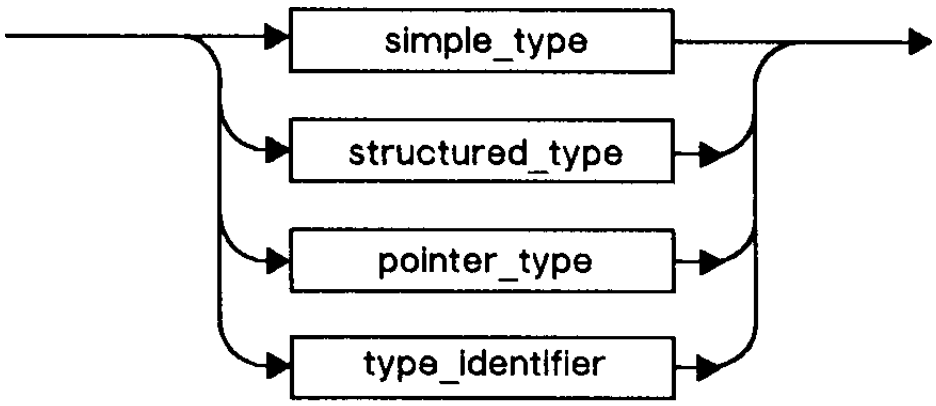
* The set of permissible values that an object of a specified type may assume.
* The set of permissible operations that may be performed on an object of a specified type.
* The amount of storage that variables of a specified type require.

Figure 3-1 summarizes the various data types in HP Pascal. A detailed discussion of the data types in each category follows in this chapter. When appropriate, permissible operators, standard procedures, standard functions, and examples are given.

NOTE The system programming extensions, if enabled, define additional data types. See Chapter 11 for more information.

Syntax

Type:



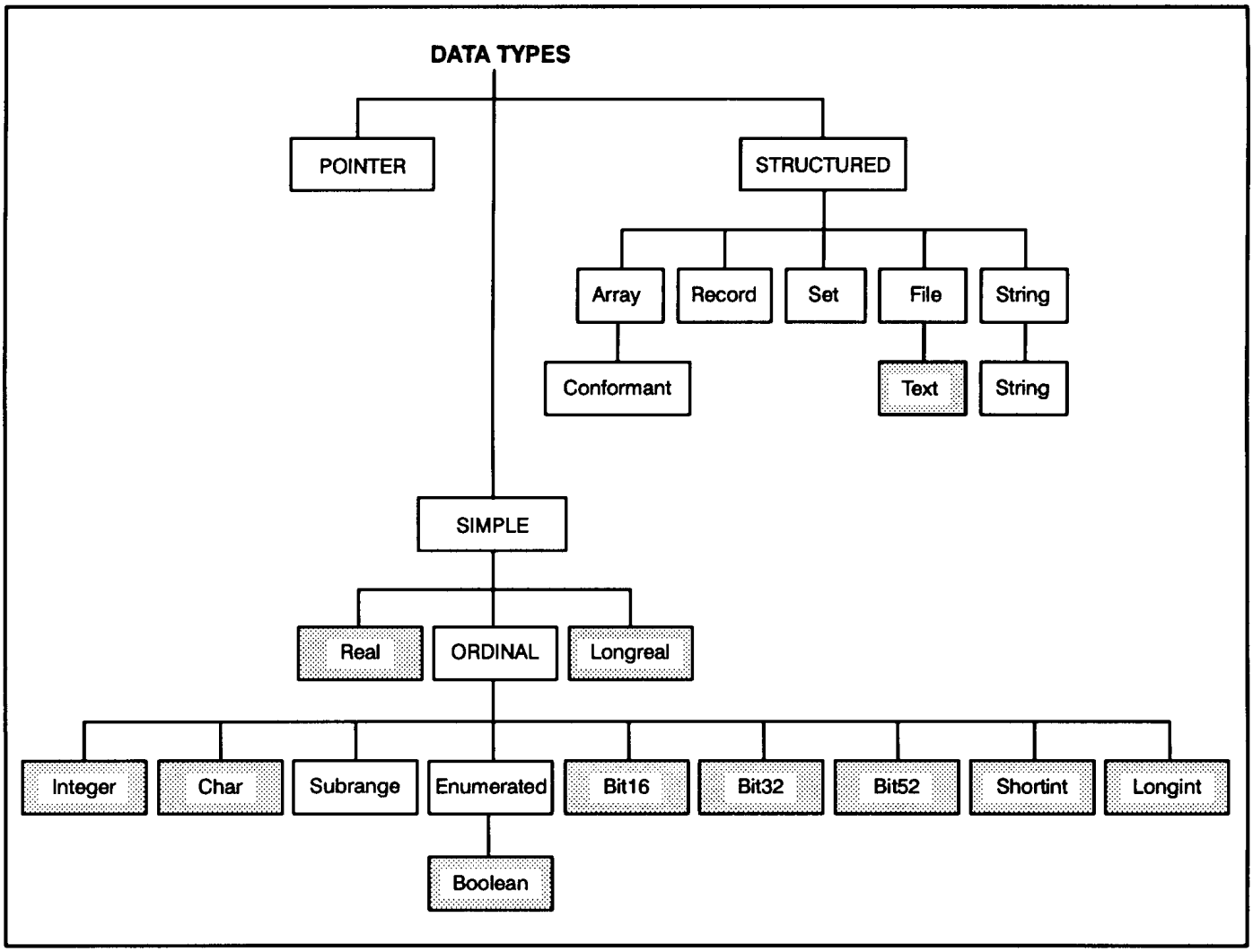


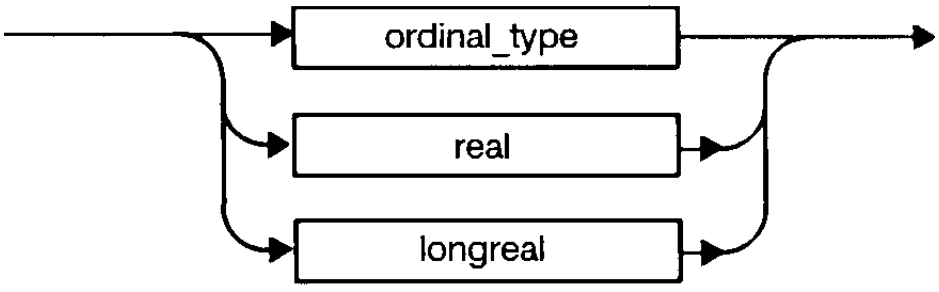
Figure 3-1. HP Pascal Data Types

Simple Types

The *simple data types* are made up of ordinal, real, and longreal types. Ordinal types include the standard types *integer*, *char*, and *Boolean* as well as *enumerated*, *subrange*, *shortint*, *longint*, *bit16*, *bit32*, and *bit52* types.

Syntax

Simple_type:



Ordinal

Ordinal types are types that have a one-to-one correspondence with a subset of natural numbers. These values are ordered so that each has a unique ordinal value that indicates its position in a list of all the values of the type.

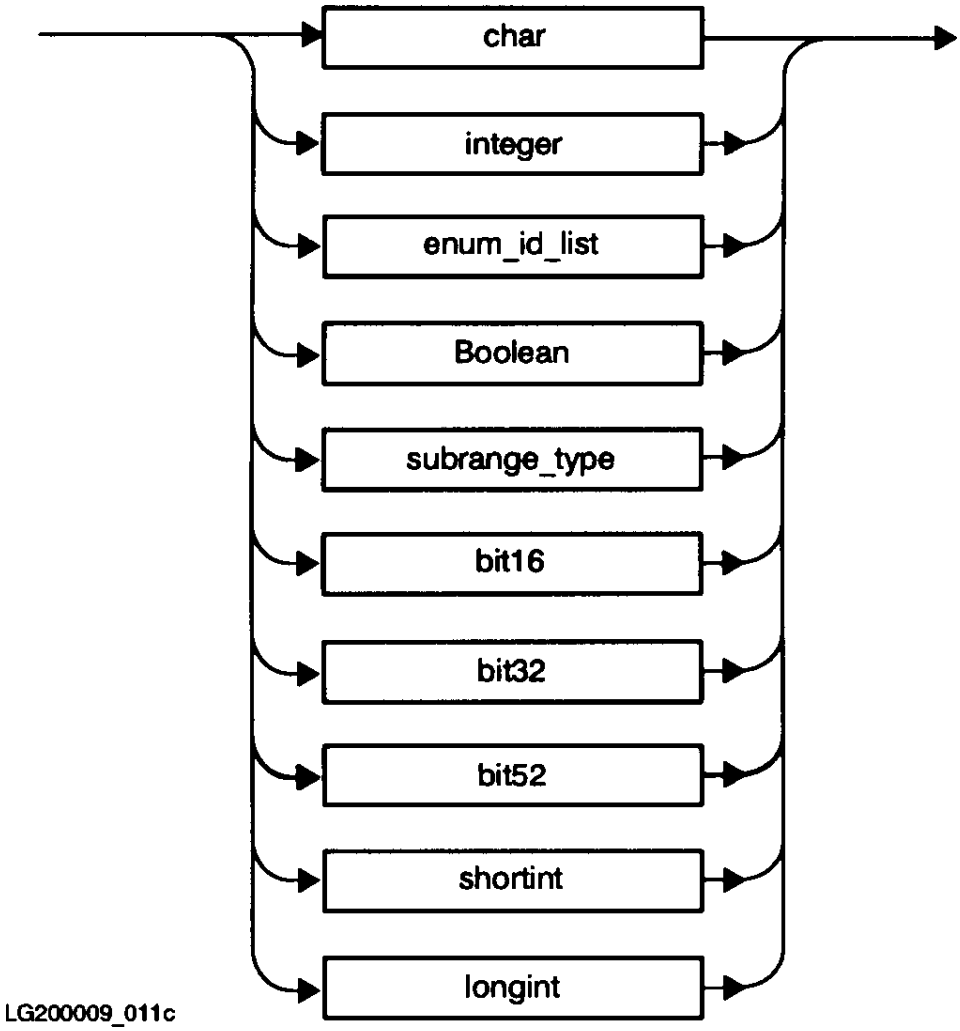
Ordinal types include *bit16*, *bit32*, *bit52*, *Boolean*, *char*, *enumerated*, *integer*, *shortint*, *longint*, and *subrange*. Enumerated types are declared by enumerating all the possible values of the type. Subrange types are declared by specifying the minimum and maximum values of the subrange.

Integral-types include *bit16*, *bit32*, *bit52*, *integer*, *shortint*, *longint*, and subrange of *integer*.

Sub-integer includes *bit16*, *shortint*, and subrange of *integer*; super-integer includes *bit52* and *longint*. *bit32* is a sub-integer when used with a *real* operand or used in a *real* function such as sin; otherwise, *bit32* is a super-integer.

Syntax

Ordinal_type:



LG200009_011c

NOTE For relational tests, the two operands must be compatible types. When membership tests are performed, the left-operand type must be a single ordinal value, while the right-operand is of a SET type.

Bit16.

The predefined data type *bit16* is a subrange, 0..65535, that is stored in 16 bits. *bit16* is a unique HP Pascal type because arithmetic operations on *bit16* data are truncated to modulo 65536 when stored.

Permissible Operators

assignment	:=
relational	<, <=, =, <>, >=, >, IN
arithmetic	+, -, *, /, DIV, MOD

Standard Functions

bit16 argument -	abs	ln	sin
	arctan	odd	sqr
	chr	ord	sqrt
	cos	pred	succ
	exp		
bit16 return -	pred		
	succ		

Standard Procedures

prompt	strread
read	strwrite
readdir	writedir
readln	writeln

Example

```
program bits1 (output);
var q:bit16;
begin
q:=hex('ffff');
q:=q + 1;                 { q is now 0 }
writeln('wrapped around value = ',q:1);
end.
```

Output:

```
wrapped around value = 0
```

Bit32.

The predefined data type *bit32* is a subrange, 0..232-1, that is stored in 32 bits. *bit32* is a unique HP Pascal type because arithmetic operations on *bit32* data are performed with unsigned 32 bit integers.

Permissible Operators

assignment	:=

relational	<, <=, =, <>, >=, >, IN
arithmetic	+, -, *, /, DIV, MOD

Standard Functions

bit32 argument -	abs	ln	sin
	arctan	odd	sqr
	chr	ord	sqrt
	cos	pred	succ
	exp		
bit32 return -	pred		
	sqr		
	succ		

Standard Procedures

prompt	strread
read	strwrite
readdir	writedir
readln	writeln

NOTE The multiply operator (*) may cause overflow traps. See "OVFLCHECK" .

Example

```
$standard_level 'hp_modcal'$
program bits2(output);
var q,r:bit32;
begin
{ one way to get bit32 constants >= 2 ** 31 }
$push; type_coercion 'conversion'; range off$
q:=bit32(hex('ffffffff')) + 1;             { q is now 0 }
r:=bit32(hex('7fffffff')) + 1;             { r is now > maxint }
$pop$
writeln('wrapped around value = ',q:1);
writeln('past maxint value    = ',r:1);
end.
```

Output:

```
wrapped around value = 0
past maxint value    = 2147483648
```

Bit52.

The predefined data type *bit52* is a subrange, 0..252-1, that is stored in 64 bits. *bit52* is a unique HP Pascal type because arithmetic operations on *bit52* data are performed with unsigned 64 bit integers.

Permissible Operators

assignment	:=
relational	<, <=, =, <>, >=, >, IN
arithmetic	+, -, *, /, DIV, MOD

Standard Functions

bit52 argument -	abs	exp	pred	succ
	arctan	ln	sin	
	chr	odd	sqr	
	cos	ord	sqrt	
bit52 return -	pred			
	sqr			
	succ			

Standard Procedures

prompt	strread
read	strwrite
readdir	writedir
readln	writeln

Example

```
$standard_level 'hp_modcal'$
program bits3(output);
var q:bit52;
begin
{ one way to get bit52 constants >= 2 ** 31 }
$push; type_coercion 'conversion'$
q:=bit52(123456) * 1000000000 + 789012345;
$pop$
writeln(q);
end.
```

Output:

```
123456789012345
```

Boolean.

The *Boolean* type is a predefined enumerated type that indicates logical values. The elements of this data type are two constant identifiers, *true* and *false*, where *false* is less than *true*. HP Pascal defines the type Boolean in the following way:

```
TYPE
   Boolean = (false, true);
```

Permissible Operators

assignment	:=
Boolean	AND, OR, NOT
relational	<, <=, =, <>, >=, >, IN

Standard Functions

Boolean argument -	ord
	pred
	succ
Boolean return -	eof
	eoln
	odd
	pred
	succ

Standard Procedures

```
prompt          strread
read            strwrite
readdir         writedir
readln          writeln
```

Example

```
VAR
   left_handed: Boolean;

BEGIN
   left_handed := false;

END;
```

Char.

The *char* type is a predefined ordinal type that is used to represent individual characters in the 8-bit ASCII character set. A *char literal* is either a single character surrounded by single quote marks, or a sharp (#) followed by a number or letter.

Permissible Operators

```
assignment          :=

relational          <, <=, =, <>, >=, >, IN
```

Standard Functions

```
char argument -     ord
                    pred
                    succ

char return -       chr
                    pred
                    succ
```

Standard Procedures

```
prompt       strread
read         strwrite
readdir      writedir
readln       writeln
```

Example

```
VAR
   do_you:  char;

BEGIN
   do_you := 'Y';
   do_you := #G; { BELL character }
END;
```

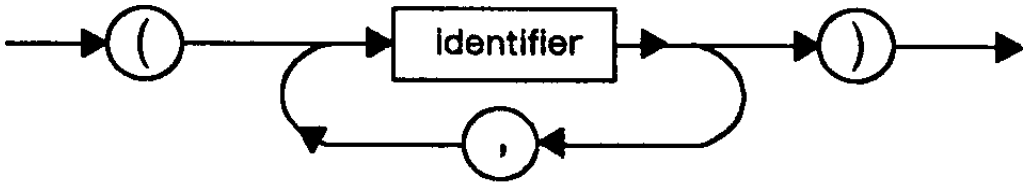
Enumerated.

An *enumerated* type is a user-defined, ordinal type that defines an ordered set of values by the enumeration of identifiers in parentheses. The sequence in which the identifiers appear determines the ordering. The enumerated identifiers are defined as constants. The ORD of the first has the value zero, and the ORD of the others have successive integer values in order of their specification. The limit on the maximum number of identifiers in an enumerated type is implementation dependent. Refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more

information.

Syntax

Enumerated_id_list:



Permissible Operators

assignment	:=
relational	<, <=, =, <>, >=, >, IN

Standard Functions

enumerated argument -	ord pred succ
enumerated return -	pred succ

Standard Procedures

prompt	strread
read	strwrite
readdir	writedir
readln	writeln

Example

```
TYPE
   days = (monday, tuesday, wednesday, thursday, friday, saturday, sunday);
   color = (red, green, blue, yellow, cyan, magenta, white, black);
```

Integer.

The *integer* type is a predefined, ordinal type whose possible values are determined by a subrange of the negative and positive integers. The lower bound of the subrange is the predefined constant *minint*, and the upper bound is the predefined constant *maxint*. The integer type represents a signed number of at least nine digits.

Permissible Operators

assignment	:=
relational	<, <=, =, <>, >, >=, IN
arithmetic	+, -, *, /, DIV, MOD

Standard Functions

integer argument -	abs	exp	pred
	arctan	ln	sin

	chr	odd	sqr	
	cos	ord	succ	
integer return -	abs	maxpos	round	strmax
	binary	octal	sqr	strpos
	hex	ord	sqrt	succ
	lastpos	position	strlen	trunc
	linepos	pred		

Standard Procedures

halt	strread
prompt	strwrite
read	writedir
readdir	writeln
readln	

Example

```
VAR
   wholenum:  integer;
   i,j,k,l :  integer;
```

Longint.

The predefined data type *longint* is an integer in the range -263..263-1 that is stored in 64 bits.

Permissible Operators

assignment	:=
relational	<, <=, =, <>, >=, >, IN
arithmetic	+, -, *, /, DIV, MOD

Standard Functions

longint argument -	abs	ln	sin	
	arctan	odd	sqr	
	chr	ord	sqrt	
	cos	pred	succ	
	exp			
longint return -	abs pred sqr succ			

Standard Procedures

prompt	strread
read	strwrite
readdir	writedir
readln	writeln

Example

```
     $standard_level 'hp_modcal'$
     program prog(output);
     var q:longint;
     begin
     { one way to get longint constants >= 2 ** 31 or < - 2 ** 31 }
     $push; type_coercion 'conversion'$
     q:=longint(123456) * 1000000000 + 789012345;
     $pop$
     writeln(q);
     end.
```

Output:

```
     123456789012345
```

Shortint.

The predefined data type *shortint* is an integer in the range -32768..32767 that is stored in 16 bits. (In contrast, if you declare a variable to be in that range, it is stored in 32 bits.)

Permissible Operators

assignment	:=
relational	<, <=, =, <>, >=, >, IN
arithmetic	+, -, *, /, DIV, MOD

Standard Functions

shortint argument -	abs	ln	sin
	arctan	odd	sqr
	chr	ord	sqrt
	cos	pred	succ
	exp		
shortint return -	pred		
	succ		

Standard Procedures

prompt	strread
read	strwrite
readdir	writedir
readln	writeln

Example

```
     program short(output);
     var q:shortint;
     begin
     q:=-1;
     writeln('size of shortint = ',sizeof(q):1);
     end.
```

Output:

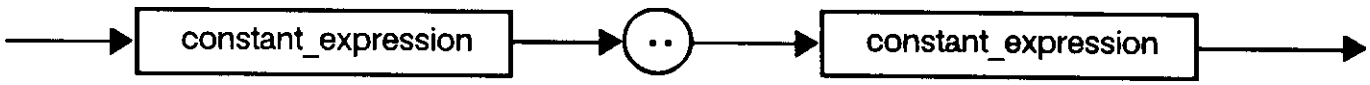
```
     size of shortint = 2
```

Subrange.

A *subrange* type is a user-defined, ordinal type that is a sequential subset of a predefined or user-defined, ordinal base type. It consists of a lower bound and an upper bound separated by the special symbol "..". The upper and lower bounds must be constant values or constant expressions of the same ordinal type. The lower bound cannot be greater than the upper bound.

Syntax

Subrange_type:



NOTE A variable of a subrange type possesses all the attributes of the base type of the subrange, but its values are restricted to the specified closed range. It has the same set of permissible operators and standard functions as its base type.

Standard Procedures

```
prompt    strread
read      strwrite
readdir   writedir
readln    writeln
```

Example

```
CONST
   maxsize = 10;

TYPE
   day_of_year = 1..366;
   lowercase   = 'a'..'z';          { Base type is char.  }
   days        = (Monday, Tuesday, Wednesday,
                  Thursday,Friday,Saturday,Sunday);
   weekdays    = Monday..Friday;
   weekend     = Saturday..Sunday;
   e_type      = 1..maxsize - 1;    { Upper bound is con- }
                                    { stant expression.   }
                                    { Maxsize is declared }
                                    { constant.           }
```

Real

The *real* type is a predefined, simple type that represents a subset of the real numbers. For the range covered by the subset, see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

Permissible Operators

assignment	:=
relational	<, <=, =, <>, >=, >
arithmetic	-, +, *, /

Standard Functions

real argument -	abs	ln	sqr
	arctan	round	sqrt
	cos	sin	trunc
	exp		
real return -	abs	exp	sqr
	arctan	ln	sqrt
	cos	sin	

Standard Procedures

prompt	strread
read	strwrite
readdir	writedir
readln	writeln

Example

```
PROGRAM show_realnum(output);

VAR
   realnum: real;
BEGIN
   realnum := 6.023E+23;
   writeln(realnum);
END.
```

Output:

```
        6.02300E+23
```

Longreal

The *longreal* type is a predefined, simple type that represents a subset of the real numbers. This type may have more precision and a larger range than the type *real*. The range the subset covers is implementation dependent in HP Pascal. For more details see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

Permissible Operators

assignment	:=
relational	<, <=, =, <>, >=, >
arithmetic	-, +, *, /

Standard Functions

longreal argument -	abs	round
	arctan	sin
	cos	sqr
	exp	sqrt
	ln	trunc
longreal return -	abs	ln
	arctan	sin

```
                    cos          sqr

                    exp          sqrt
```

Standard Procedures

```
prompt       strread
read         strwrite
readdir      writedir
readln       writeln
```

Example

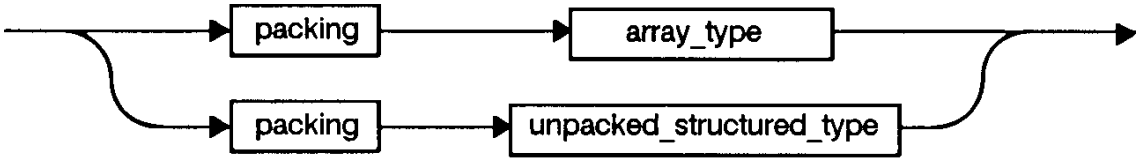
```
VAR
   precisenum: longreal;
BEGIN
   precisenum:= 1.1234567891L+04;
   .
```

Structured Types

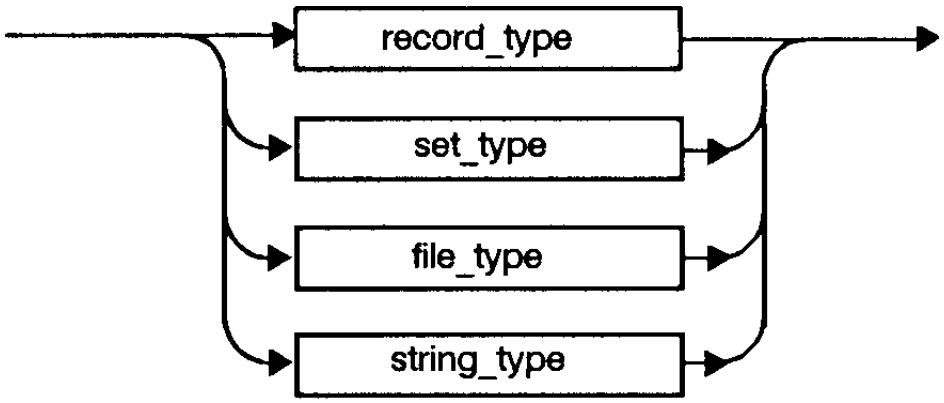
Structured data types are the *array, file, record, set,* and *string* types. These data types can be preceded by a packing modifier. The effect of the packing modifier is implementation-defined. Refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information.

Syntax

Structured_type:



Unp_Struc_type:

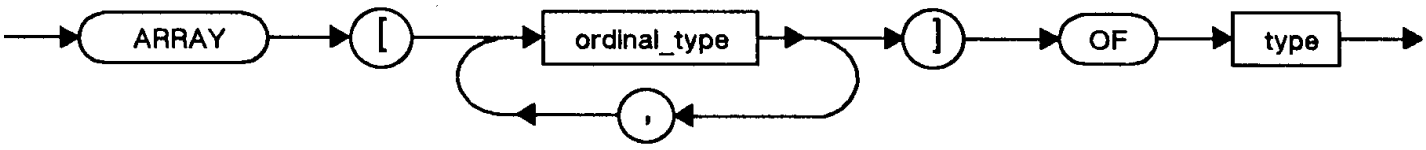


ARRAY

An *array* is a structured type consisting of a fixed number of components that are all of the same type. The maximum number of components is implementation dependent. Depending on your implementation, refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide* for more information.

Syntax

Array_type:



Array Declarations

An array type definition consists of the reserved word ARRAY, an index type in square brackets, the reserved word OF, and the component type. The reserved word PACKED may precede ARRAY. It instructs the compiler to optimize storage space for the array components, possibly at the expense of execution time.

An index that must be an ordinal type specifies the number of component of an array. The *component* type may be any *simple, structured,* or *pointer* type, including a *file* type. The symbols (. and .) may replace the left and right square brackets, respectively. The component of an array may be accessed using the index of the component in a selector.

In the ANSI/IEEE770X3.97 - 1983 Standard Pascal, the term *string* designates a packed array of char with a starting index of 1 and an ending index >1. HP Pascal uses the term PAC to designate a packed array of char with a starting index of 1. HP Pascal also defines a standard type string that is similar to a packed array with a declared maximum length, whose actual length may vary at run time.

Permissible Operators

assignment	:=
relational (PAC only)	<, <=, =, <>, >=, >

Standard Functions

strlen**
hex**
octal**
binary**

Standard Procedures

array parameters -	pack	strread**
	prompt**	strwrite**
	read*	unpack
	readdir*	write*
	readln**	writedir*
	strmove**	writeln**

NOTE One asterisk (*) after a routine name indicates that this routine can be used on all arrays, whereas two asterisks (**) indicates that this routine should be used with PAC arrays only.

Example

```
TYPE
   name    = PACKED ARRAY [1..30] OF char; { PAC type }
   list    = ARRAY [1..100] OF integer;
   strange = ARRAY [Boolean] OF char;
   flag    = ARRAY [(red, white, blue)] OF 1..50;
   files   = ARRAY [1..10] OF text;
```

Multi-Dimensioned Arrays

If an array definition specifies more than one index type or if the components of an array are themselves arrays, then the array is said to be *multi-dimensioned*. The maximum number of array dimensions is implementation dependent.

Example

```
TYPE
   { equivalent definitions of truth }
   truth  = ARRAY [1..20] OF
               ARRAY [1..5] OF
                  ARRAY [1..10] OF Boolean;
   truth  = ARRAY [1..20] OF
               ARRAY [1..5, 1..10] OF Boolean;
   truth  = ARRAY [1..20, 1..5] OF
               ARRAY [1..10] OF Boolean;
   truth  = ARRAY [1..20, 1..5, 1..10] OF Boolean;
```

FILE

This reserved word designates a declared data structure that consists of a sequence of components all of the same type. *Files* are usually associated with peripheral storage devices, and their length is not specified in the program. A *file_type* consists of the reserved words FILE OF and a component type that may be predefined or user-defined. The type *text* is a special type of FILE OF CHAR that has additional attributes. For further information about textfiles, refer to the section "Standard Textfiles" in this chapter.

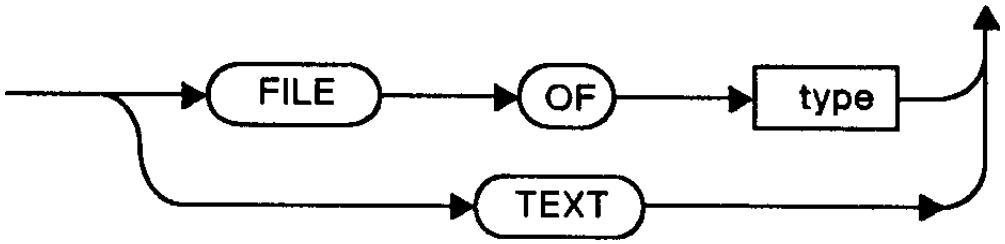
A *logical* file is a file variable declared in an HP Pascal program. A *physical* file is a file that exists in the environment outside the program and is controlled by the operating system. During program execution, logical files are associated with physical files, allowing any operation performed on the logical file to be performed on the physical file. Thus, a program is allowed to manipulate data in the external environment.

A logical file may be any type except a file type or a structured type with a file type component. The number of components is not fixed by the file type definition. File components may be accessed sequentially or directly using a variety of HP Pascal standard procedures and functions.

It is legal to declare a *packed* file. The effect on the storage of the file is implementation dependent.

Syntax

File_type:



Example

```
TYPE
   person      = RECORD
                   name: PACKED ARRAY [1..30] OF char;
                   age:  1..100;
                 END;
   person_file = FILE OF person;

   bit_vector  = PACKED ARRAY [1..100] OF Boolean;
   vector_file = FILE OF bit_vector;

   data_file   = FILE OF integer;
   doc_file    = text;
```

Standard Textfiles

text

Text type variables are called *textfiles*. The standard file type *text* permits ordinary input and output that is oriented to characters and lines. Text type files have two important features:

* The components are type *char*.

* The file is subdivided into lines by special *end-of-line markers*.

Textfiles cannot be opened for direct access with the procedure *open*. Textfiles can be sequentially accessed, however, with the procedures *reset, rewrite,* or *append*. All standard procedures that are legal for sequentially-accessed files are also legal for textfiles.

Certain standard procedures and functions, on the other hand, are only legal for textfiles. These procedures are:

* eoln
* linepos
* overprint
* page
* prompt
* readln
* writeln

Textfiles permit conversion from the internal form of certain types to an ASCII character representation and vice versa.

Example

```
VAR
   myfile: text;
        i: integer;
        r: real;
```

```
BEGIN
  rewrite(myfile);
  writeln(myfile,'integer',i);
  writeln(myfile,'real',r);
END.
```

input

When the standard textfile *input* appears as a program parameter, there are several important consequences:

* *Input* may not be declared in the global declaration of the source code.

* The system automatically associates *input* with an implementation-dependent physical file.

* The system automatically resets *input*.

* If certain file operations omit the logical file name parameter, *input* is the default file. For example, the call read(x) where x is some variable, reads a value from *input* into x. Consider:

```
PROGRAM mute (input);
VAR  answer : string[255];
BEGIN
   readln(answer);
END.
```

The program waits for input. Output need not appear.

If an imported module uses *input*, it must appear as a program parameter for the importing program, and the module must import the predefined module *stdinput*.

output

When the standard textfile *output* appears as a program parameter, there are several important consequences:

* *Output* may not be declared in the global declaration part of the source code.

* The system automatically associates *output* with an implementation dependent, physical file. Depending on your implementation, refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide* for more information.

* The system automatically rewrites *output*.

* If certain file operations omit the logical file name parameter, *output* is the default file. For example, the call write(x), where x is some variable, writes the value of x onto *output*. Consider:

```
PROGRAM sample (output);
BEGIN
   writeln('I like Pascal!');
END.
```

The program displays the string literal on the default output device. *output* must appear as a program parameter; *input* need not appear if the program does not use it.

If an imported module uses *output*, it must appear as a program parameter for the importing program, and the module must import the predefined module *stdoutput*.

Record

A *record* is a structured type consisting of a collection of components that are not necessarily of the same type. Each component is termed a *field* of the record and has its own identifier. A field of a record is accessed by using the appropriate *field selector*.

A record type consists of the reserved word RECORD, a field list, and the reserved word END. The reserved word PACKED may precede the reserved word RECORD. If PACKED is used, it instructs the compiler to optimize storage of the record fields.

Syntax

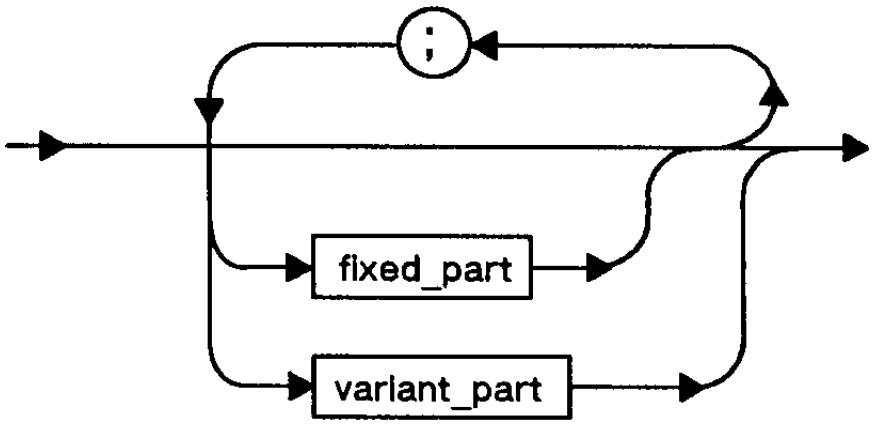
Record_type:

Field List

The *field list* has an optional *fixed* part and an optional *variant* part. The field list may have any number of fields, and each field is given a unique name called a *field identifier*.

Syntax

Field_list:

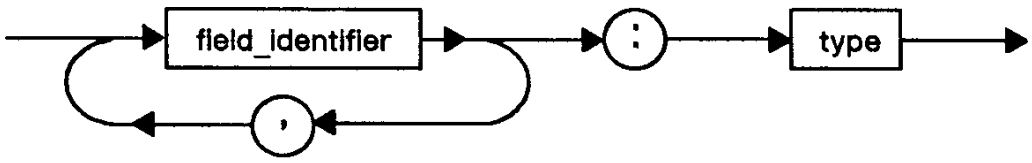


Fixed Part

In the *fixed part* of the field list, a field definition consists of an *identifier*, a *colon* (:), and a *type*. Any simple, structured, or pointer type is legal. Several fields of the same type may be defined by listing the identifiers separated by commas.

Syntax

Fixed_part:



Variant Part

In the *variant part*, the reserved word CASE introduces an optional *tag field* identifier and a required *ordinal type identifier*. The reserved word OF precedes a list of case constants and alternative field lists.

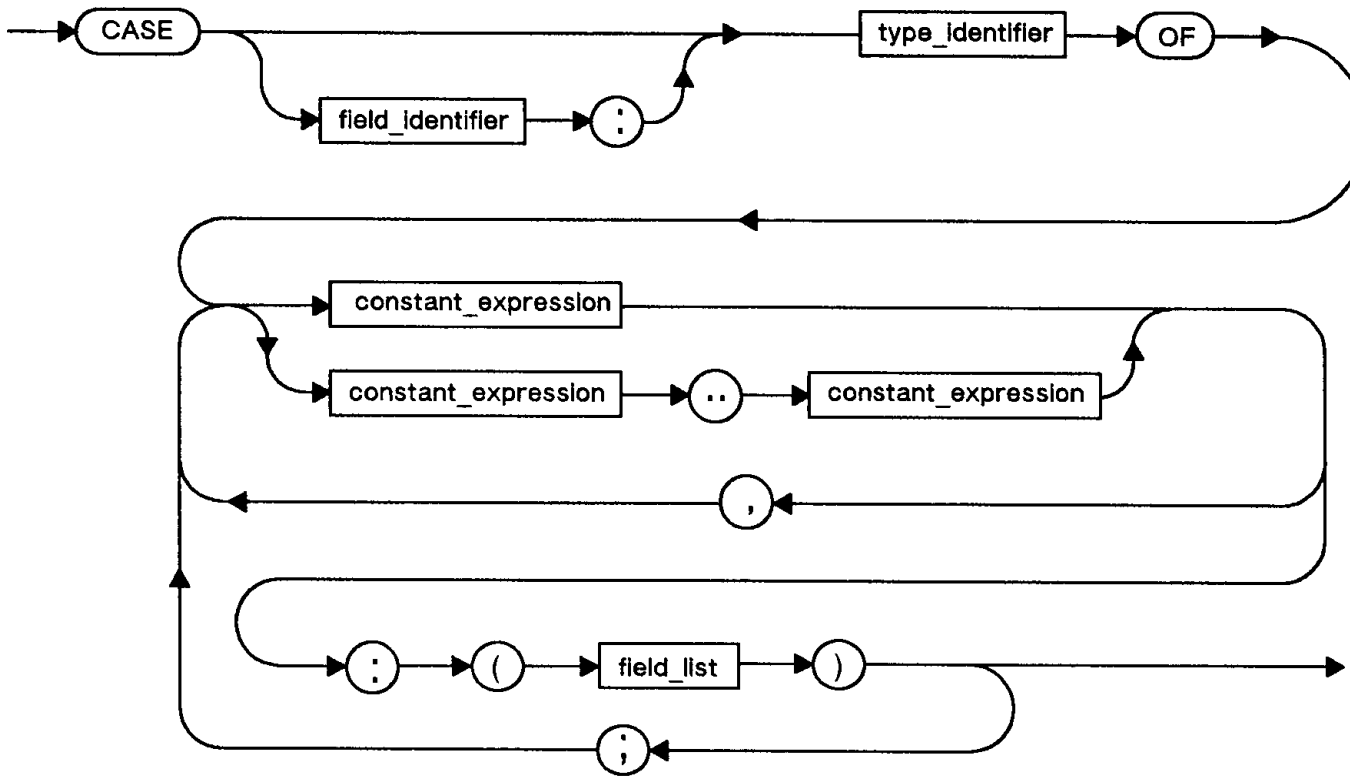
Case constants must be compatible with the *tag*. See "Type Compatibility" in this chapter for more information. Several case constants may be associated with a single field list. The various constants appear separated by commas. Subranges are also legal case constants in HP Pascal. The empty field list may be used to indicate that a variant doesn't exist. This is illustrated in the example in this section. HP Pascal does not require that all possible tag values be specified.

The OTHERWISE construction may not be used in the variant part of the field list. OTHERWISE is only legal in CASE statements.

Variant parts allow variables of the same record type to exhibit structures that differ in the number and type of their component parts. If a record has *multiple variants*, when a value is assigned to the tag field, any fields associated with a previous variant cease to exist, and the new variant's fields become active with undefined values. If there is no tag field when a value is assigned to a field of any particular variant, any fields associated with another variant cease to exist, and the new variant fields become active with undefined values. An error results when a reference is made to a field of a variant other than the current variant. A field of a record is accessed using the appropriate field selector.

Syntax

```
Variant_part:
```



Permissible Operators

assignment (entire record)	:=
field selection	.

Standard Procedures

```
read
readdir
write
writedir
```

Example

```
TYPE
   word_type = (int, ch);
   word      = RECORD             { variant part only with tag }
                 CASE word_tag: word_type OF
                   int: (number: integer);
                   ch : (chars : PACKED ARRAY [1..2] of char);
               END;

   polys   = (circle, square, rectangle, triangle);
   polygon = RECORD      { fixed part and tagless variant part }
               poly_color: (red, yellow, blue);
               CASE polys OF
                 circle:    (radius: integer);
                 square:    (side: integer);
                 rectangle: (length, width: integer);
                 triangle:  (base, height: integer);
             END;
```

```
date_info    = PACKED RECORD           { fixed part only }
                 mo: (jan, feb, mar, apr, may, jun,
                      jul, aug, sep, oct, nov, dec);
                 da: 1..31;
                 yr: 1900..2001;
               END;
marital_status = (married, separated, divorced, single);
name_string    = PACKED ARRAY [1..30] of CHAR;
person_info    = RECORD                { nested variant parts }
                   name: name_string;
                   born: date_info;
                   CASE status: marital_status of
                     married..divorced:
                               (when:   date_info;
                                CASE has_kids: Boolean OF
                                  true: (how_many: 1..50);
                                  false: (); { Empty variant }
                                )
                     single: ();
                   END;
```

Set

A *set* is a user-defined, structured type that is the power set consisting of the set of all subsets of a *base type*. A set type consists of the reserved words SET OF and a *base type*. The *base type* may be any ordinal type. The maximum number of elements is implementation defined, but must be at least 256 elements. It is legal to declare a packed set. However, whether this affects the storage is implementation dependent. HP Pascal defines "SET OF integer" (or any other integral-type) as "SET OF 0..255".

Syntax

Set_type:

Permissible Operators

assignment	:=
union	+
intersection	*
difference	-
subset	<=
superset	>=
equality	=, <>
inclusion	IN

Example

```
TYPE
   charset   = SET OF char;
   fruit     = (apple, banana, cherry, peach, pear, pineapple);
```

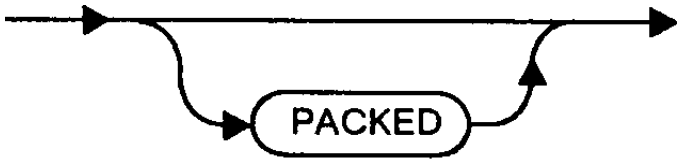
```
somefruit = SET OF apple..cherry;
poets     = SET OF (Blake, Frost, Brecht);
some_set  = SET OF 1..200;
```

PACKED

This reserved word indicates that the compiler should minimize data storage even if the access time may be increased. The reserved word PACKED may appear with an ARRAY, RECORD, SET, or FILE. By declaring a PACKED structured data type, the amount of memory needed to store an item is generally reduced. The decision to *pack* a particular data type depends on many factors including available memory size, processor speed, required response time, and volume of data. Therefore, a choice that is valid for one environment may be quite inappropriate for another. It is illegal to pass a component of a packed structure by reference.

Syntax

Packing:



Example

```
CONST
   wordsize = 20;

VAR
   buffer: ARRAY [1..wordsize] OF char;
   word:   PACKED ARRAY [1..wordsize] OF char;
```

String

In HP Pascal a *string* type consists of the standard identifier *string* and an integer constant expression in square brackets that specifies the maximum length. *Integer constant expressions* are constant expressions that return an integer value, an unsigned integer being the simple case. The limit for the maximum length is implementation defined, but must be at least 255. The symbols **(.** and **.)** may replace the left and right brackets, respectively.

Characters enclosed in single quotes are *string literals*. The compiler interprets a string literal as type *PAC, string,* or *char,* depending on the context.

When a formal reference parameter is type *string*, the maximum length need not be specified. This allows actual string parameters to have various maximum lengths.

A single component of a string can be accessed by using an integer expression in square brackets as a selector. The numbering of the characters in the string begins at one. Strings are initialized by performing an operation that sets the current length, making an assignment to the entire string or by calling *setstrlen*.

Syntax

String_type:

A string expression may consist of any of the following:

* A string literal.
* A string variable.
* A string constant.
* A function result that is a string.
* An expression formed with the concatenation operator.

NOTE Variables of type string, as well as other Pascal variables, are not initialized. The current string length contains meaningless information until the string is initialized.

Permissible Operators

assignment	:=
concatenation	+
relational	=, <>, <=, >=, >, <

Standard Functions

string argument -	str	strpos
	strlen	strrpt
	strltrim	strrtrim
	strmax	
string return -	str	
	strltrim	
	strrpt	
	strrtrim	

Standard Procedures

string parameter -	prompt	strinsert
	read	strmove
	readdir	strread
	readln	strwrite
	setstrlen	write

```
                  strappend    writedir

                  strdelete    writeln
```

Example

```
     CONST
        maxlength = 100;

     TYPE
        name   = string[30];
        remark = string[maxlength * 2];

     PROCEDURE proc1 (VAR s: string); EXTERNAL; { Maximum length }
                                                { not required.  }
```

Pointer Types

Pointers

A *pointer* is a data type that may reference any type, including type FILE. A pointer references a dynamically allocated variable on the heap. The *pointer type* consists of the caret (^) and a type identifier. The @ symbol may replace the caret.

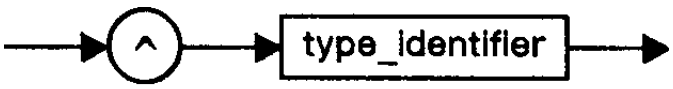
The type appearing after the caret need not be previously defined. This is an exception to the general rule that HP Pascal identifiers are first defined and then used. However, the identifier after the caret must be defined within the same declaration part.

A *type identifier* used in a pointer type declaration in an EXPORT section need not be defined until the IMPLEMENT section. A pointer declared in this manner cannot be dereferenced in modules that IMPORT the pointer type.

The pointer value NIL belongs to every pointer type. NIL points to no variable on the heap. For more information, refer to the section on NIL in this chapter.

Syntax

Pointer_type:



Permissible Operators

```
 assignment      :=

 equality        =, <>
```

Standard Procedures

```
 pointer parameters -    new
                         dispose
                         mark
                         release
```

Example

```
     TYPE
```

```
        ptr1   = ^rec1;
        ptr2   = ^rec2;
        rec1   = RECORD
                    f1, f2: integer;
                    link:   ptr2;
                 END;
        rec2   = RECORD
                    f1, f2: real;
                    link:   ptr1;
                 END;
```

Type Compatibility

Relative to each other, two HP Pascal types can be *identical*, *type compatible*, or *incompatible*. The guidelines that determine type compatibility are listed below.

Identical Types

Two types are identical if either of the following is true:

* Their types have the same type identifier.

* If A and B are two type identifiers, and they were made equivalent by a definition of the form:

```
TYPE A = B
```

Compatible Types

Two types, T1 and T2, are type compatible if any of the following is true:.

* T1 and T2 are identical types.

* T1 and T2 are subranges of identical base types, T1 is a subrange of T2, or T2 is a subrange of T1.

* T1 and T2 are set types with compatible base types and both T1 and T2 or neither are packed or crunched.

* T1 and T2 are PAC types.

* T1 and T2 are both string types.

* T1 and T2 are both real types.

Incompatible Types

Two types are *incompatible* if they are *not identical* or *type compatible,* or *assignment compatible*. In the following example all of the variables are type compatible, but v4, v5, and v6 have identical types. The variables v2 and v3 also have identical types.

Example

```
     TYPE
        interval = 0..10;
        range = interval;

     VAR
        v1 : 0..10;
        v2, v3: 0..10;
        v4 : interval;
        v5 : interval;
        v6 : range;
```

Note that two types that are structurally the same are not necessarily

compatible. In the following example, types T1 and T2 are not compatible. Variables v3 and v4 are also not compatible.

```
PROGRAM t(input,output);

TYPE
   T1 = record
           a:  integer;
           b:  char;
        end;

   T2 = record
           c:  integer;
           d:  char;
        end;
VAR
   v1:  T1;
   v2:  T2;
   v3:  ^T1;
   v4:  ^T2;

BEGIN
   v1:=  v2;  { This generates a compile-time error }
   v3:=  v4;  { This generates a compile-time error }
END.
```

Assignment Compatibility

A value of type T2 may only be assigned to a variable or function result of type T1 if T2 is *assignment compatible* with T1. For T2 to be *assignment compatible* with T1, any of the following conditions must be true:

* T1 and T2 are type compatible types that are neither files nor structures that contain files.

* T1 is real or longreal, and T2 is integer or an integer subrange. The compiler converts T2 to real or longreal prior to assignment.

* T1 is longreal and T2 is real. The compiler converts T2 to longreal prior to assignment.

* T1 is real and T2 is longreal. The compiler rounds T2 to the precision of T1 prior to assignment.

Furthermore, a *run-time* or *compile-time* error occurs if the following restrictions are not observed:

* If T1 and T2 are type compatible ordinal types, the value of type T2 must be in the closed interval specified by T1.

* If T1 and T2 are type compatible set types, all the members of the value of type T2 must be in the closed interval specified by the base type of T1.

* A special set of restrictions applies to assignment of string literals or variables of type string, PAC, or char.

NOTE The pointer constant NIL is both type compatible and assignment compatible with any pointer type. Likewise, the empty set ([ ]) is both type compatible and assignment compatible with any set type.

String Assignment Compatibility

Certain restrictions apply to the assignment of string literals or variables of the type string, PAC, or char. These restrictions are listed below.

* If T1 is a string variable, T2 must be a string variable or a string literal whose length is equal to or less than the maximum length of T1. T2 cannot be a PAC or char variable. Assignment sets the current length of T1.

* If T1 is a PAC variable, T2 must be a PAC or a string literal whose current length is less than or equal to the length of T1. T1 is blank filled if T2 is a string literal or PAC that is shorter than T1. T2 cannot be a string or a char variable.

* If T1 is a char variable, T2 may be a char variable or a string literal with a single character. T2 cannot be a string or PAC variable.

Table 3-1 summarizes these rules. The standard function *strmax(s)* returns the maximum length of the *string* **s**. The standard function *strlen(s)* returns the current length of the *string* s or the number of characters in the PACs.

String constants are considered string literals when they appear on the right side of an assignment statement. Any string operation on two string literals, such as the concatenation of two string literals, results in a string of type string.

Table 3-1. String, PAC, and String Literal Assignment

T1:=T2	string	PAC	char	String Literal
string	Only if strmax(T1)>= strlen(T2)	Not allowed	Not allowed	Only if strmax(T1)>= strlen(T2)
PAC	Not allowed	Only if strlen (T1) >= strlen (T2) T1 is padded with blanks if necessary	Not allowed	Only if strlen (T1) >= strlen(T2) T1 is padded with blanks if necessary
char	Not allowed	Not allowed	Yes	Only if strlen(T2)=1

Chapter 4 Expressions

An *expression* is a construct composed of operators and operands that represent the computation of a result of a particular type. In the simplest case, an expression consists of a single operand with no operator.

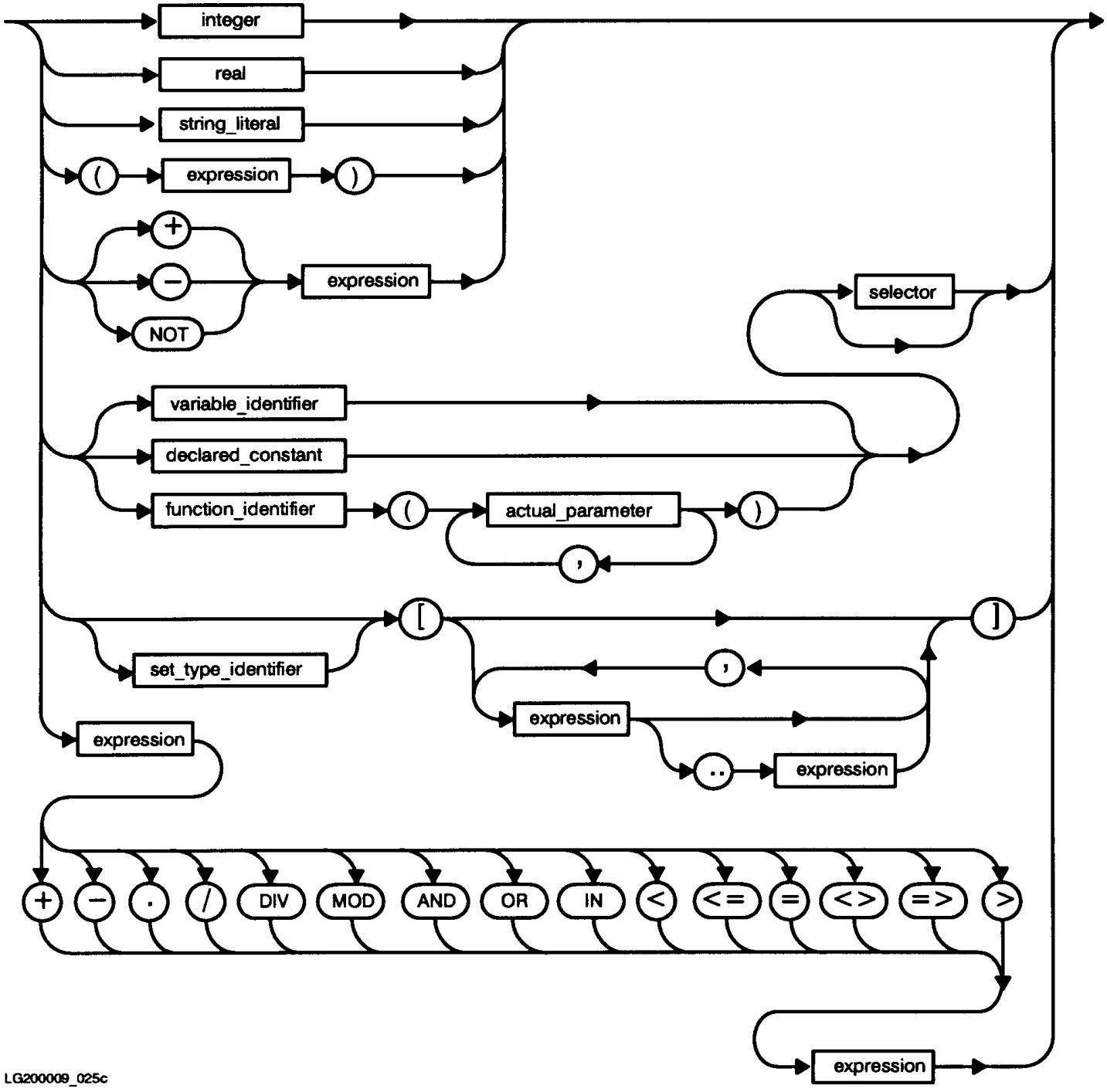
The type of an expression is known when the expression is written, and never changes. The actual value, however, may not be known until the system evaluates the expression at run time. It may differ for each evaluation.

Constant expressions are a restricted class of HP Pascal expressions. They must return a value that is computable at compile time. Consequently, operands in constant expressions must be integers, reals, longreals, or declared constants. The operators used with constant expressions must be +, -, *, DIV, or MOD. All other operators are excluded. Furthermore, only calls to the following standard functions are legal:

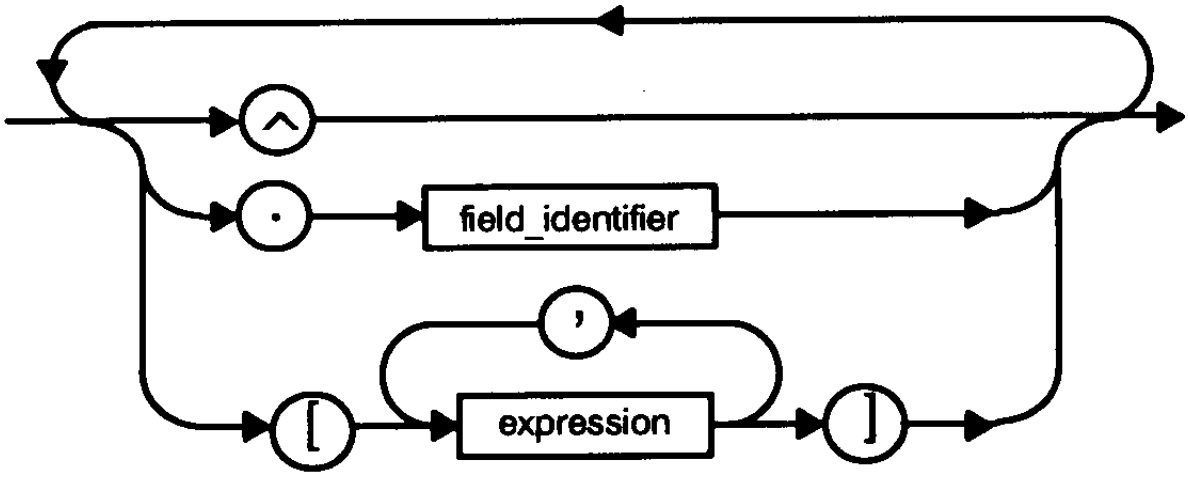
* abs
* binary
* chr
* hex
* octal
* odd
* ord
* pred
* strlen
* succ

Syntax

Expression:



Selector:



Example

```
x:= 19;                 { Simplest case. "19" is the expression   }
                        { in the statement: "x := 19".            }

100 + x;                { Arithmetic operator with literal and    }
                        { variable operands.                      }

(A OR B) AND (C OR D)   { Boolean operator with Boolean operands. }

x > y                   { Relational operator with variable       }
                        { operands.                               }

setA * setB;            { Set operator with variable operands.    }

'ice'+'cream'           { Concatenation operator with string      }
                        { literal operands.                       }

x:= func1(B);           { Function call                           }
```

Operands

An *operand* denotes the object that operators use in obtaining a value. An operand may be a *literal*, *a declared constant*, a *variable access (variable)*, a *set constructor*, a *dereferenced pointer*, or the *value* of another expression. *Function calls* are also operands in the sense that they return a result that an operator can use to compute another value.

An operand may be acted upon by an operator. Performing an operation on operands of different types is called mixing data types. In all cases except one, you cannot mix data types. You can, however, mix *reals* and *integers* with an operator that allows two real operators. Table 4-1 provides a list of operands and tells where they are described in the manual.

Table 4-1. HP Pascal Operands

Operand	Chapter
literal	4
constant	4
variable	5
set constructor	4
function call	4,8
dereferenced pointer	4
array selector	4
record selector	4
file buffer selector	4

Operators

An *operator* defines an action to be performed on one or more operands and produces a value. An operator may be classified as *arithmetic, Boolean, relational, set,* or *concatenation*. A particular symbol may occur in more than one class of operators. For example, the symbol **+** is an *arithmetic, set,* and *concatenation* operator representing numeric addition, set union, and string concatenation, respectively. The class of the operator is determined by the type of the operands.

Precedence ranking determines the order in which the compiler evaluates a sequence of operators. For more information about precedence, refer to the section on Operator Precedence in this chapter.

The value resulting from the action of an operator may in turn serve as an operand for another operator. Table 4-2 lists each HP Pascal operator together with its actions, permissible operands, and type of results. In the table, the term *real* indicates both real and longreal types and *integer* indicates any integral-type.

Table 4-2. HP Pascal Operators

Operator	Actions	Type of Operands	Type of Results
+	addition set union concatenation	real, integer any set type string, string literal	real, integer set string
-	subtraction set difference	real, integer any set type	real, integer set
*	multiplication set intersection	real, integer any set type	real, integer set
/	division	real, integer	real
DIV	division with truncation	integer	integer
MOD	modulus	integer	integer
AND	logical 'and'	Boolean	Boolean
OR	logical 'or'	Boolean	Boolean
NOT	logical negation	Boolean	Boolean
<	less than	any simple type string or PAC	Boolean Boolean
>	greater than	any simple type string or PAC	Boolean Boolean

Table 4-2. HP Pascal Operators (cont.)

Operator	Actions	Type of Operands	Type of Results
<=	less than or equal, set subset	any simple type string or PAC any set type	Boolean Boolean Boolean
>=	greater than or equal, set superset	any simple type string or PAC any set type	Boolean Boolean Boolean
=	equal to	any simple type string or PAC any set type pointer	Boolean Boolean Boolean Boolean
<>	not equal to	any simple type string or PAC any set type pointer	Boolean Boolean Boolean Boolean
IN	set membership	left operand: any ordinal type T right operand: set of T	Boolean

Operator Precedence

The *precedence* ranking of an HP Pascal operator determines the order of its evaluation in an unparenthesized sequence of operators. The four levels of ranking are:

PRECEDENCE	OPERATORS
highest	NOT
.	*, /, DIV, MOD, AND
.	+, -, OR
lowest	<, <=, <>, =, >=, >, IN

The compiler evaluates higher precedence operators first. For example, since * ranks above +, it evaluates these expressions identically:

```
(x + y * z)    and   (x + (y * z))
```

When a sequence of operators has equal precedence, the order of evaluation is implementation dependent. If an operator is commutative, for example, *, the compiler may evaluate the operands in any order. Note that within a parenthesized expression the compiler evaluates the operators and operands without regard for any operators outside the parentheses.

Arithmetic Operators

Arithmetic operators perform *integer* and *real* arithmetic by taking numeric *operands* and producing a *numeric result*. These operators are +, -, *, /, DIV, and MOD.

Most arithmetic operators permit *real*, *longreal*, or *integral-type* operands. However, DIV and MOD only accept integral-type operands. The type of the result of a unary operator is the same as the type of its operand. However, if the operand is bit16, the result is an integer

type, and if the operand is bit32 or bit52, the result is a longint. The type of the result of a binary operator is the same as the data types of its operands, provided that both operands are of the same type. Special rules apply for division and in cases where operands have different data types.

Implicit Type Conversion of Operands

The operators +, -, *, and / permit operands with different numeric types. For example, it is possible to add an *integer* and a *real* number. The compiler converts the integer to a real number, and the result of the addition is real.

This *implicit conversion* of operands relies on a ranking of numeric types. This is defined as follows:

RANK	TYPE
highest	longreal
.	real, longint, bit52
.	integer, bit32
lowest	sub-integer

The rank of the value the result of an operation is the same as the highest rank of all the operands. Operands having types whose ranks are less than the rank of the type of the result are converted prior to the operation, so that they have a type with a rank equal to that of the result type.

For example, if i is an integer and x is a real in the expression (x + i), then i is converted to real before the addition. In short, the two operands to an arithmetic operator must be compatible. For more details, refer to the section "Type Compatibility" .

Table 4-3. Type Comparisons and Results

Operand A Type	Operand B Type	Results
sub-integer	sub-integer	sub-integer
sub-integer	integer	integer
sub-integer	real	real
integral-type	longreal	longreal
integer	real	real
integer	super-integer	longint
integral-type	longint	longint
longint	real	longreal
real	super-integer	longreal
real	longreal	longreal
bit16	bit32	bit32
bit16	bit52	bit52
bit32	bit52	bit52

Real division (/) is an exception to this restriction. If both operands are integer or sub-integer, the compiler changes both to real numbers prior to the division and the result is real. If both operands are super-integers, the result is longreal because both[REV BEG] operands are converted to longreal.[REV END]

Example

```
EXPRESSION      RESULT

-(+10)          -10      { Unary -.                                  }
5 + 2           7        { Addition with integer operands.           }
5 - 2.0         3.0      { Subtraction with implicit conversion.     }
5 * 2           10       { Multiplication with integer operands.     }
5.0 / 2.0       2.5      { Division with real operands.              }
5 / 2           2.5      { Division with integer operands, real      }
                         { result.                                   }
5.0 / 2      2.5         { Division with implicit conversion.        }

5 DIV 2          2       { Division with truncation.                 }
5 DIV (-2)      -2
-5 DIV 2        -2
-5 DIV (-2)      2

5 MOD 3          2       { Modulus.                                  }
5 MOD (-2)     error     { Right operand must be positive.           }
(-5) MOD 3     1         { Result is positive regardless of          }
                         { sign of left operand, which is            }
                         { parenthesized since MOD has higher        }
                         { precedence than -.                        }
                         { See Operator Precedence.                  }
```

DIV

This operator returns the integer portion of the quotient of the *dividend* and the *divisor*. The dividend must be an integral-type with no range restriction. The divisor must also be an integral-type; the divisor cannot be 0.

Example

```
INPUT                          RESULT

 413 DIV 6                      68
-413 DIV 6                     -68
```

MOD

This operator returns the *modulus* of two integers. The dividend must be an integral-type. The divisor must also be an integral-type. If the divisor is less than or equal to 0, an error will occur. The result is always positive, regardless of the sign of the left operand. The left operand must be parenthesized if it is a negative literal. MOD is defined as:

(i - k * j) *for some integer* k

such that

0 <= i MOD j < j, j > 0

Example

```
INPUT                        RESULT

4 MOD 3                     1
7 MOD 5                     2
(-7) MOD 5                  3
```

Boolean Operators

Boolean operators perform logical functions on Boolean type operands and produce Boolean results. The Boolean operators are NOT, AND, and OR. When both operands are Boolean, = denotes *equivalence*, <= *implication*, and <> *exclusive or*.

The compiler can be directed to perform or not perform partial evaluation of Boolean operators used in statements. For example:

```
IF right_time AND right_place THEN ...
```

By specifying the $PARTIAL_EVAL ON$ compiler directive, if right_time is *false*, the remaining operators are not evaluated since execution of the statement depends on the logical AND of both operators. Both operators have to be *true* for the logical AND of the operators to be *true*.

Similarly, the logical OR of two operators are *true* even if only one of the operators is *true*. Partial evaluation allows expressions like (Ptr <> NIL) AND (Ptr^.F1) to execute without an error when Ptr is NIL.

Example

```
IF NOT possible THEN forget_it;

WHILE time AND money DO your_thing;

REPEAT...UNTIL tired OR bored;

IF has_rope THEN skip;

IF pain <= heartache THEN try_it;

FUNCTION NAND (A, B : BOOLEAN) : BOOLEAN;
   BEGIN
   NAND := NOT(A AND B);  { NOT AND }
END;

FUNCTION XOR (A, B : BOOLEAN) : BOOLEAN;
   BEGIN
   XOR := NOT(A AND B) AND (A OR B);  { EXCLUSIVE OR }
END;

FUNCTION XOR (A, B : BOOLEAN) : BOOLEAN;
   BEGIN
   XOR := A <> B;
END;
```

AND

This Boolean operator is used to perform the logical operation on two Boolean operands. The result is of type Boolean. The following truth table illustrates the operator AND along with its results.

OPERATOR	RESULT
AND	The evaluation of two Boolean operands produces a Boolean result, such that:

(logical and)

a	**b**	**a AND b**
false	false	false
false	true	false
true	false	false
true	true	true

Example

```
VAR
   bit6, bit7 : Boolean;
   counter    : integer;

BEGIN
   ...
   IF bit6 AND bit7 THEN counter := 0;
   ...
   IF bit6 AND (counter = 0) THEN bit7 := true;
   bit7 := bit6 AND (counter = 0);
END
```

NOT

This Boolean operator complements the value of the Boolean expression following the NOT operator. The result is of type Boolean. The truth table for NOT is given below.

OPERATOR	RESULT
NOT (logical negation)	The logical negation of a single Boolean operand, such that:

a	**NOT a**
false	true
true	false

Example

```
PROGRAM show_not(input,output);

VAR
   time, money : Boolean;
   line        : string[255];
   test_file   : text;

BEGIN
   .
   .
   IF NOT (time AND money) THEN wait;
   .
   .
   WHILE NOT eof(test_file) DO
      BEGIN
         readln(test_file,line);
         writeln(line);
      END;
   .
   .
END.
```

OR

This Boolean operator is used to perform the logical inclusive OR operation on two Boolean operands. The result is the logical OR of its two factors. The OR operator is shown below in terms of its truth table.

OPERATOR	RESULT
OR (inclusive or)	The evaluation of two Boolean operands produces a Boolean result, such that:

a	b	a OR b
false	false	false
false	true	true
true	false	true
true	true	true

Example

```
PROGRAM show_or(input,output);

VAR
   ch     : char;
   time   : Boolean;
   energy : Boolean;

BEGIN
   .
   .
   IF time OR energy THEN do_it;
   .
   .
   IF (ch = 'Y') OR (ch = 'y') THEN ch := 'Y';
   .
   .
END.
```

Relational Operators

Relational operators compare two operands and return a Boolean result. The relational operators are <, <=, =, <>, >=, >, and IN. The following lists the relational operators with their associated meanings:

OPERATOR	MEANING
<	less than
<=	less than or equal to
=	equal
<>	not equal
>=	greater than or equal
>	greater than
IN	set membership

Depending on the type of its operands, a relational operator may be classified as *simple, set, pointer,* or *string*. For a description of simple, set, pointer, or string relational operators, refer to the appropriate section in this chapter.

Simple Relational Operators

A *simple relational* operator has operands of any simple type such as integer, Boolean, char, real, longreal, enumerated, or subrange. All the operators listed above, except IN, may be simple relational operators. The operands must be type compatible, but the compiler may implicitly convert numeric types before evaluation. For more information about converting numeric types, refer to the section "Arithmetic Operators" in this chapter.

For numeric operands, simple relational operators impose the ordinary definition of ordering. For char operands, the ASCII collating sequence

defines the ordering. For enumerated operands, the sequence in which the constant identifiers appear in the type definition defines the ordering. If both operands are *Boolean*, the operator = denotes *equivalence*, <= denotes *implication*, and <> denotes *exclusive OR*. Therefore, the predefinition of Boolean as:

```
TYPE Boolean = (false, true);
```

means that false < true.

Example

```
PROGRAM show_simple_relational;

VAR
  b: Boolean;
BEGIN
   .
   .
  b := 5 > 2;
  b := 5 < (25.OL+1);
END.
```

Set Relational Operators

A *set relational* operator has set operands. The set relational operators are =, <>, >=, <=, and IN. The operators = and <> compare two sets for equality or inequality, respectively. The <= operator denotes the subset operation, while >= indicates the superset operation such that Set A is a subset of Set B, if every element of A is also a member of B. When this is true, B is said to be the superset of A.

The IN operator determines if the left operand is a member of the set specified by the right operand. When the right operand has the type SET OF T, the left operand must be type compatible with T. To test the negative of the IN operator, the following form must be used:

```
NOT (element IN set)
```

Example

```
PROGRAM show_set_relational; (output)

TYPE
   color= (red,yellow,blue);
VAR
   b: boolean;
   s,t: SET OF color;
   col: color;
BEGIN
  col:= red;
  s:= [red];
  t:= [blue];
  b:= s <> t;
  writeln (b);
  b:= s <= t;
  writeln (b);
  b:= col IN [yellow,blue];
  writeln (b);
END.
```

Output:

```
TRUE
FALSE
FALSE
```

IN.

This operator returns *true* if the specified element is a member of the specified set. The result is *false* if the expression is not a member of the set. Both the element being tested and the elements in the set must be of compatible types.

Example

```
PROGRAM show_in(output);

VAR
   ch : char;
   good : SET OF char;
   member : Boolean;

BEGIN
   ch := 'y';
   good := ['y','Y','n','N'];
   IF ch IN good THEN
      member := true
   ELSE
      member := false;
   writeln(member);
END.
```

Output:

```
TRUE
```

Pointer Relational Operators

The *pointer relational* operators = and <> can be used to compare two *pointers* for equality or inequality, respectively. Two pointers are equal only if they point to exactly the same object or both contain the value NIL. Only two pointers of identical type or the constant NIL can be compared.

Example

```
PROGRAM show_pointer_relational;

VAR
   a, b: boolean;
   p, q: ^boolean;
      x: ^char;
BEGIN
   .
   .
  IF (p = q) AND (p <> NIL) THEN p^:= a = b;  { pointer }
  b := x <> q; { is an error - x and q are not compatible }
END.
```

NOTE No assumptions should be made about the integer values of pointers and their integer value relations. Such values and relations are undefined.

String Relational Operators

The *string relational* operators =, <>, <, <=, >, or >= may be used to compare operands of type *string*, *PAC*, *char,* or *string literals*. The system performs the comparison character by character using the order defined by the ASCII collating sequence. Note that it is not possible to compare a string variable with a PAC or char variable. In general, these guidelines are as follows:

* If one operand is a string expression, the other operand may be a string expression or string literal. If the operands are not the same length and the two are equal up to the length of the shorter, the shorter operand is less. For example, if the current value of S1 is abc and the current value of S2 is ab, then S1 > S2 is *true*.

* If one operand is a PAC expression, the other may be a PAC or string literal of any length. The shorter is blank-filled prior to comparison.

* If one operand is a char expression, the other may be a char expression or a single-character string literal.

Table 4-4 summarizes these rules. The standard function *strmax(s)* returns the maximum length of the string variable **s**. The standard function *strlen(s)* returns the current length of the string expression **s**. A string constant is considered a string literal when it appears on either side of a relational operator.

Table 4-4. String, PAC, Char, String Literal Comparison

A <relop> B	string	PAC	char	string literal
string	Length of comparison based on smaller *strlen*	Not allowed	Not allowed	Length of comparison based on smaller *strlen*
PAC	Not allowed	The shorter of the two is padded with blanks	Not allowed	The shorter of the two is padded with blanks
char	Not allowed	Not allowed	Yes	Only if strlen(B)=1
string literal	Length of comparison based on smaller *strlen*	The shorter of the two is padded with blanks	Only if strlen(A)=1	The shorter of the two is padded with blanks

Example

```
PROGRAM show_string_relational (output) ;

VAR
   s,t:  string[80];
   pac:  packed array [1..5] of char;
   chr:  char;
   b:    boolean;

BEGIN
   s:='abc';
   t:='ab';
   if s > t then b:=true          {string to string comparison. this is}
   else b:=false;                 { the same as b:= s > t              }
   writeln (b);
   b:= s > 'ab';                  {string to string literal comparison  }
   writeln (b);
   pac:='abc';
   b:= pac > 'abc';               {PAC to string literal comparison    }
   writeln (b);
   chr:= 'A';
   b:= 'c' > chr;                 {char to string literal comparison   }
   writeln (b);
END.
```

Output:

```
TRUE
TRUE
FALSE
TRUE
```

Concatenation Operator

The *concatenation operator* + concatenates two operands that may be string variables, string literals, function results of a string type, or some combination of these types. The result of the concatenation is always type string.

NOTE It is not legal to use the concatenation operator in a constant definition.

Example

```
VAR
   s1,s2: string[80];

BEGIN
   s1:='abc';
   s2:='def';
   s1:= s1 + s2;  { s1 is now 'abcdef' }
   writeln('s1 has: ',s1);
   s2:= 'The first six letters are ' + s1;
   writeln('s2 has: ',s2);
END.
```

Output:

```
   s1 has: abcdef
   s2 has: the first six letters are abcdef
```

SET Operators

The *set* operators perform set operations on two set *operands*. The result is of type set. The set operators are +, -, and *. Operands used with set operators may be *variables, constant identifiers,* or *set constructors*. The base types of the set operands must be type compatible with each other.

OPERATOR	RESULT
+ (union)	A set whose members are all the elements present in the left set operand and those in the right, including members present in both sets.
- (difference)	A set whose members are the elements which are members of the left set but are not members of the right set.
* (intersection)	A set whose members are only those elements present in both of the set operands.

Example

```
PROGRAM show_setops;

VAR
   a, b, c: SET OF 1..10;
   x : 1..10;

BEGIN
```

```
   .
   .
   a:= [1, 3, 5];
   b:= [2, 4];
   c:= [1..10];
   x:= 9;
   a:= a + b;        { Union; a is now [1, 2, 3, 4, 5].       }
   b:= c - a;        { Difference; b is now [6, 7, 8, 9, 10]. }
   c:= a * b;        { Intersection; c is now [].             }
   c:= [2, 5] + [x]  { Set constructor operands; c is now     }
END.                 { [2, 5, 9].                             }
```

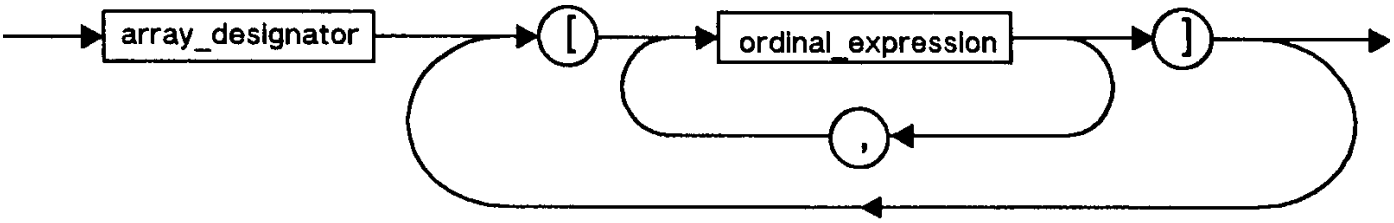
Array Selector

An *array selector* accesses a component of an array. The selector follows an array designator and consists of an ordinal expression in square brackets. For a string or PAC type, an array selector accesses a single component; for example, a character.

The ordinal expressions must be assignment compatible with the index types of the array. An array designator can be any variable with an array type that includes an array selector, a function call that returns an array, or an array constant. The symbols (. and .) may replace the left and right brackets, respectively. The list can be used to select a component of a multiple-dimensioned array.

Syntax

Array_selector:



Example

```
PROGRAM show_arrayselector;

TYPE
   a_type = ARRAY [1..10] OF integer;

VAR
   m,n         : integer;
   s_array     : ARRAY [1..3] OF 1..100;
   multi_array : ARRAY [1..5,1..10] OF integer;
   p           : ^a_type;

BEGIN
   s_array[2]:= 32;
   m:= s_array[2];           { Assigns current value of 2nd  }
                             { component of s_array to m     }
   multi_array[2,9]:= m;     { These two methods of          }
   multi_array[2][9]:= m;    { assignment are equivalent.    }

   new(p);
   p^[1]:= 1200;
   n:= p^[m MOD 10 + 1] * m; { Array in the heap with computed }
END.                         { selector.                       }
```

Record Selector

A *record selector* accesses a *field* of a record. The record selector follows a record designator and consists of a period and the name of a field. A record designator is the variable name of a record, the selected component of a structure that is a record, or a function call that returns a record.

The WITH statement "opens the scope" of a record. This makes it unnecessary to specify a record selector.

Syntax

Record_selector:

Example

```
PROGRAM show_recordselector;

TYPE
   r_type = RECORD
              f1: integer;
              f2: char;
            END;
VAR
   a,b       : integer;
   ch        : char;
   r         : r_type;
   rec_array : ARRAY [1..10] OF r_type;

BEGIN
    .
   a:= r.f1 + b;      { Adds the current value of integer field   }
    .                 { of r to b and assigns the result to a.    }
    .
   rec_array[a].f2:= ch; { Assigns current value of ch to char   }
    .                    { field of the a'th component of rec_array. }
END.
```

Set Constructor

A *set constructor* designates one or more values as members of a set whose type may or may not have been previously declared. A set constructor consists of an optional set type identifier and one or more ordinal expressions in square brackets. Two expressions may serve as the lower and upper bound of a subrange.

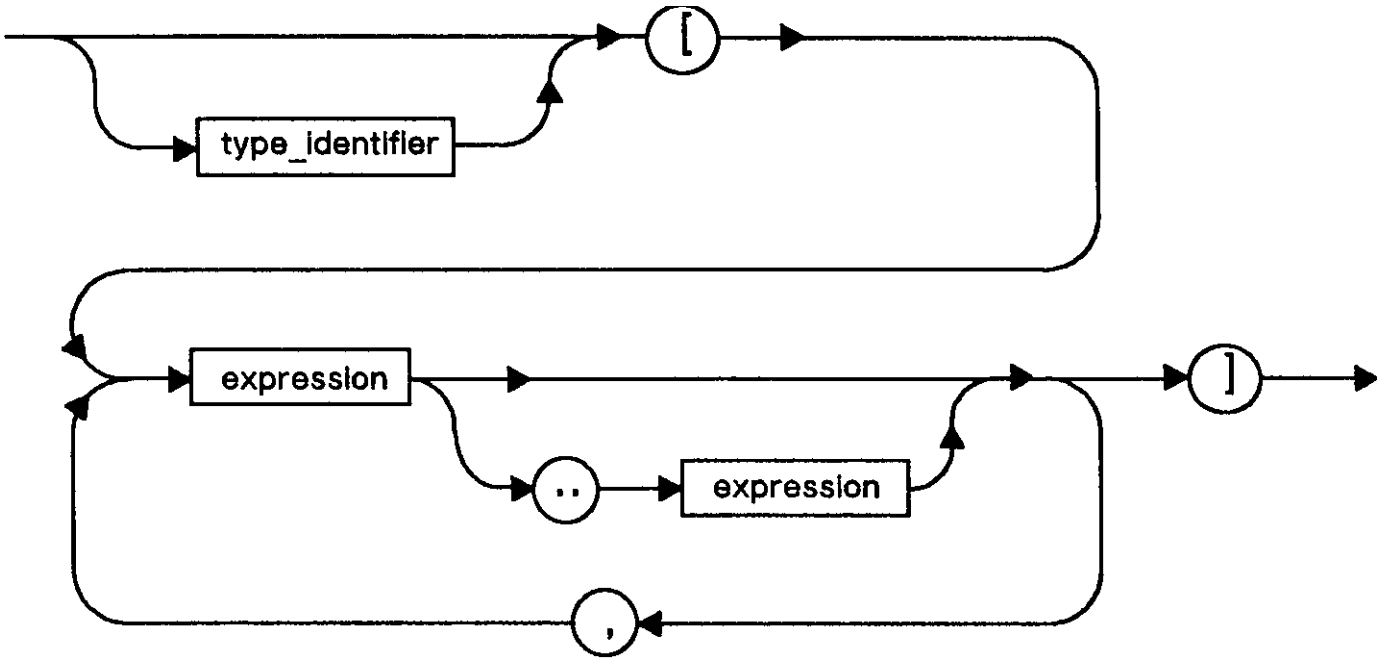
If the set type identifier is specified, the values in the brackets must be assignment compatible with the base type of the set. If no set type identifier appears, the values must be type compatible with each other. The symbols (. and .) may replace the left and right brackets, respectively.

Set constructors may appear as operands in expressions in executable statements. Set constructors with constant values are legal in the constant declaration sections.

A set constructor of the form [i..j] where i and j are integral-type variables, is defaulted to a set of integer (set of 0..255). If it appears in an expression and the size of the other operand is larger than zero to 255, [i..j] is assumed to be the size of the other operand.

Syntax

Set_constructor:



Example 1

```
PROGRAM show_setconstructor;
TYPE
   int_set = SET OF 1..100;
   cap_set = SET OF 'A'..'Z';

VAR
   a,b: 0..255;
   s1: SET OF integer;
   s2: SET OF char;

BEGIN
    .
    .
   s1:=[b, 7, 10]; { no type identifier }
   s1:= int_set[(a MOD 100) + (b MOD 100)];
   s2:= cap_set['B'..'T', 'X', 'Z'];
END.
```

Example 2

```
VAR
   s1 : set of 0..366;
   i,j : integer;

BEGIN
   s1 := [i..j] * s1;  {in this context, [i..j] becomes a set of 0..366.}

     .
     .
     .
END.
```

File Buffer Selector

A file buffer selector accesses the contents, if any, of the file buffer variable associated with the current position of a file. The selector follows a file designator and consists of the caret symbol (^).

A file designator is the name of a file or the selected component of a structure which is a file. The @ symbol may replace the caret. If the file buffer variable is not defined at the time of selection, a run-time error occurs.

Syntax

File_selector:

Example

```
PROGRAM show_file_selector(output);

VAR
   f1:  FILE OF integer;

BEGIN
   rewrite(f1);
   f1^:= 5;
   put(f1);
   reset(f1);
   IF f1^ <> 5 THEN
      writeln('error')
   ELSE
      writeln('success');
END.
```

Output:

```
success
```

Pointer dereferencing

A pointer variable points to a dynamically allocated variable on the heap. The current value of this variable may be accessed by dereferencing its pointer value. *Pointer dereferencing* occurs when the *caret* symbol (^) appears after a pointer designator in source code. A dereferenced pointer can be an operand in an expression.

The pointer designator may be the name of a pointer or selected component of a structured variable that is a pointer. The @ symbol may replace the caret. It is an error to dereference NIL or an undefined pointer value.

Syntax

Pointer_deref:

Example

```
PROGRAM show_pointerderef (output);

TYPE
   p = ^integer;
VAR
   a,b     : integer;
   p_array : ARRAY [1..10] OF p;
   ptr     : p;

BEGIN
    .
   p_array[a]^:= a + b;
    .
   writeln(ptr^ * 2);      { Dereferenced pointer is operand. }
    .
END.
```

Function Calls

A *function call* invokes the block of a standard or user defined function and returns a value to the calling point of the program. An operator can perform some action on this value, and, for this reason, a function result is an expression. See Chapter 8 for a complete description of function calls.

Example

```
PROGRAM show_function_call;

VAR x: integer;

FUNCTION sum (A,B: integer): integer;
BEGIN
   sum := A + B;
END;

BEGIN
   x:= sum (1,2) + 3;
   x:= sum(x,sum(x,sum(0,1)));
END.
```

Chapter 5 The Declaration Section

The first two parts of an HP Pascal block are the heading and the declaration section. The heading specifies the name of the program, module, procedure, or function. The declaration section contains sections that define constants and user-defined types, and sections that declare labels, variables, procedures, functions, and modules. Each of these sections is introduced by an appropriate reserved word such as LABEL, CONST, IMPORT, MODULE, TYPE, VAR, PROCEDURE, or FUNCTION. A block need not include all of these sections. In HP Pascal, CONST, TYPE, VAR, MODULE, and IMPORT declaration sections can be intermixed and must follow label declarations and precede function or procedure declarations.

This chapter describes *constant definitions, label declarations, type definitions,* and *variable declarations*. For information on procedure, function, module, and import declarations, see Chapter 7 .

Constant Definition

A *constant definition* establishes an *identifier* as a synonym for a *constant value*. The identifier may then be used in place of the value. The value of a symbolic constant may not be changed by a subsequent constant definition in the same scope or by an assignment.

The reserved word CONST precedes one or more constant definitions. A constant definition consists of an identifier, the equal sign, (=) and a constant value. For more information about CONST, refer to the section "CONST" in this chapter.

The reserved word NIL is a pointer value representing a NIL value for all pointer types. Predeclared constants include the standard constants *maxint* and *minint*, as well as the standard Boolean constants *true* and *false*. These constants are discussed in detail in the following pages of this chapter.

Constant expressions are a restricted class of HP Pascal expressions. Consequently, operands in constant expressions must be *integers*, *reals*, or *ordinal declared constants*. Operators must be +, -, *, /, DIV, or MOD. Note that all other operators are excluded. Furthermore, only calls to the standard functions *abs*, *binary*, *chr*, *hex*, *octal*, *odd*, *ord*, *pred*, *strlen*, and *succ* are legal.

One exception to the restrictions on constant expressions is permitted; the sign of a *real* or *longreal* declared constant may be changed using the negative real *unary* operator (-). The positive operator (+) is legal, but has no effect.

In HP Pascal, constant definitions must follow label declarations and precede function or procedure declarations. CONST, TYPE, VAR, MODULE, and IMPORT sections may be intermixed.

Example

```
CONST
   fingers   = 10;              { Unsigned integer.          }
   pi        = 3.1415;          { Unsigned real.             }
   message   = 'Use a fork!';   { String literal.            }
   nothing   = NIL;
   delicious = true;            { Standard constant.         }
   neg_pi    = -pi;             { Real unary operator.       }
   hands     = fingers DIV 5;   { Constant expression.       }
   numforks  = pred(hands);     { Constant expression with   }
                                { call to standard function. }
```

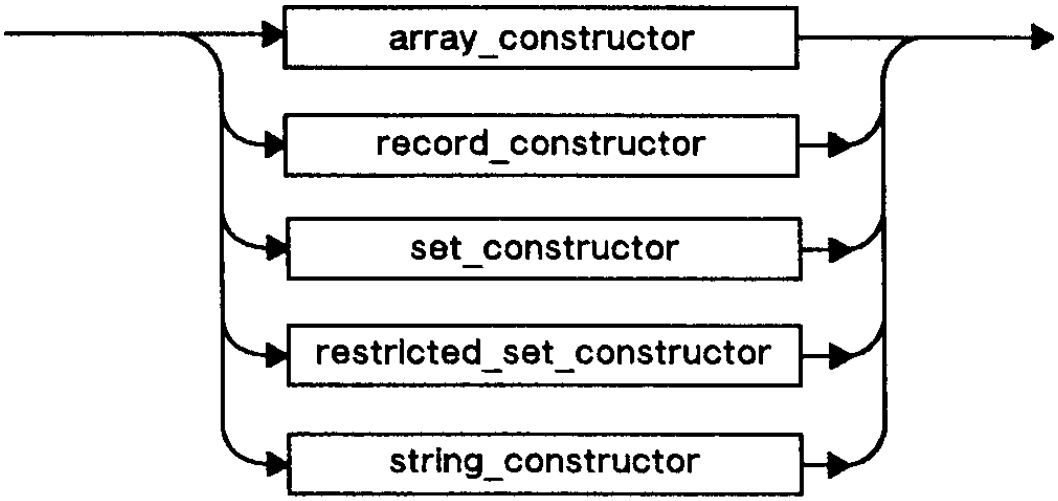
CONST

This reserved word indicates the beginning of one or more *constant* definitions that introduces an identifier as a synonym for a constant value. The identifier may then be used in place of that value.

Constant definitions appear after the program header or any LABEL declarations, and before any procedure or function definitions. In HP Pascal, CONST, TYPE, VAR, MODULE, and IMPORT definitions may be intermixed.

Syntax

Const_decl:



Example

```
PROGRAM show_CONST;

LABEL 1;

TYPE
   type1 = integer;
   type2 = Boolean;
   str1  = string[5];

CONST
   const1 = 3.1415;                { constant }
   const2 = true;
   strconst = str1['abcde'];       { string_constructor }

VAR
   var1 : type1;

BEGIN
END.
```

For examples of structured constants, see the appropriate sections.

false

This predefined *Boolean* constant is equal to the *Boolean* value *false*. The ordinal value of *false* is 0.

Example

```
PROGRAM show_false(output);

VAR
   what, lie : Boolean;

BEGIN
   IF false THEN writeln('Always false, never printed.');
```

```
   what := false;
   lie := NOT true;
   IF what = lie THEN writeln('Would I lie?');
END.
```

Output:

```
Would I lie?
```

true

This predefined *Boolean* constant is equal to the *Boolean* value *true*. The *ordinal* value of *true* is 1.

Example

```
PROGRAM show_true(output);

VAR
   what, truth : boolean;

BEGIN
   IF true THEN writeln('Always true, always printed.');
   what := true;
   truth := NOT false;
   IF what = truth THEN writeln('Everything I say is a lie.');
END.
```

Output:

```
Always true, always printed.
Everything I say is a lie.
```

maxint

This standard constant returns the upper bound of the integer type. The value is implementation defined, however, it must allow for at least nine decimal digits. For more information, see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

Example

```
PROGRAM show_maxint(input,output);

VAR
   i,j : integer;
   r   : real;

BEGIN
   readln(i,j);
   r := i + j;
   IF r > maxint THEN writeln('Sum too large for integers.');
END.
```

minint

This standard constant returns the lower bound of the integer type. The value is implementation defined, however, it must allow at least nine decimal digits. In general, the range of signed integers allows the absolute value of *minint* to be greater than *maxint*. For more information, see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

Example

```
PROGRAM show_minint(input,output);

VAR
   i,j : integer;
   r   : real;

BEGIN
   readln(i,j);
   r := i - j;
   IF r < minint THEN writeln('Difference too large for integers.');
END.
```

NIL

This predefined constant is the value of a *pointer* that designates that the pointer does not point at anything. NIL is compatible with any pointer type. A NIL pointer or pointer that has been assigned to NIL does not point to any variable at all. It is an error to dereference a NIL valued pointer.

NIL pointers are useful in linked list applications where the link pointer points to the next element of the list. The last element's pointer can be assigned to NIL to indicate that there are no further elements in the list.

Array Constants and Array Constructors

An *array constant* is a declared constant defined with an array constructor that specifies values for the components of an array type. The values for all elements of the structured type must be specified and must have a type identical to the type of the corresponding elements.

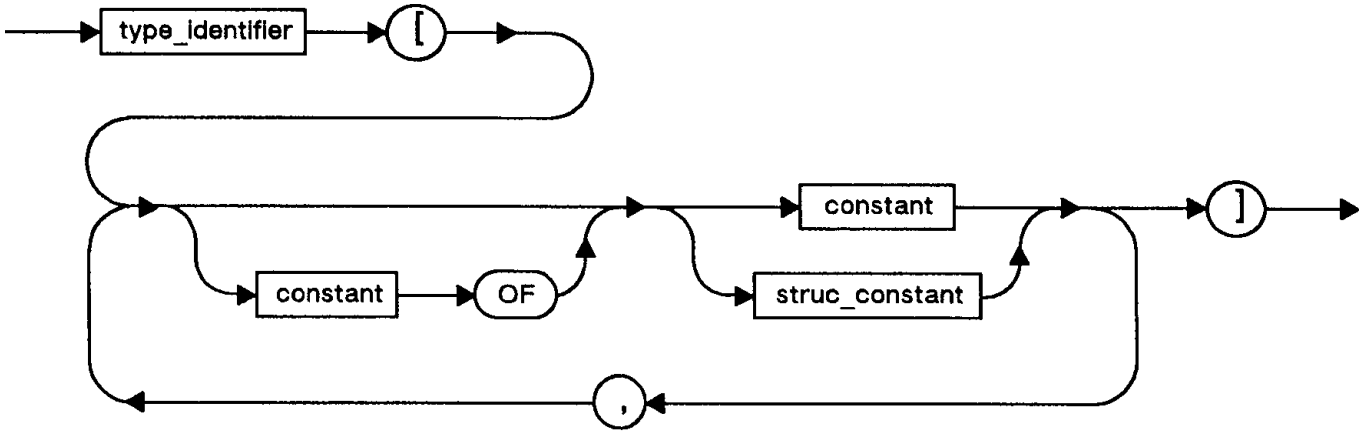
An *array constructor* consists of a previously defined array type identifier and a list of values in square brackets. Array constructors are only legal in a CONST section of a declaration part. They cannot appear in other sections or in executable statements. Each component of the array type must receive a value that is assignment compatible with the component type. There is a shorthand allowed for PAC and string constants where a string literal may be used to assign values to multiple components. An array constant may not contain files.

An array constant may be used to initialize a variable in the executable part of a block. Individual components of an array constant may also be accessed in the body of a block, but not in the definition of other constants.

Within the square brackets, the reserved word OF indicates that a value occurs repeatedly. For example, 3 OF 5 assigns the integer value 5 to three successive array components. The symbols (. and .) may replace the left and right square brackets, respectively.

Syntax

Array_constructor



Example

```
TYPE
   Boolean_table = ARRAY [1..5] OF Boolean;
   table         = ARRAY [1..100] OF integer;
   row           = ARRAY [1..5] OF integer;
   matrix        = ARRAY [1..5] OF row;
   color         = (red, yellow, blue);
   color_string  = PACKED ARRAY [1..6] OF char;
   color_array   = ARRAY [color] OF color_string;
```

```
CONST
   true_values   = Boolean_table [5 OF true];
   init_values1  = table [100 OF 0];
   init_values2  = table [60 OF 0, 40 OF 1];
   identity      = matrix [row [1, 0, 0, 0, 0],
                           row [0, 1, 0, 0, 0],
                           row [0, 0, 1, 0, 0],
                           row [0, 0, 0, 1, 0],
                           row [0, 0, 0, 0, 1]];
   colors        = color_array [color_string ['RED', 3 OF ' '],
                                color_string ['YELLOW'],
                                color_string ['BLUE', 2 OF ' ']];
```

The name of the previously declared constant may be specified within a structured constant. The previous example can also be written as indicated below. Note that for the special case of PAC that if all of the components are not specified, as in the example below, the remaining components are filled with blanks as assignment compatibility indicates.

```
CONST
   red    = 'RED';
   yellow = 'YELLOW';
   blue   = 'BLUE';

   colors  =  color_array [ color_string[red];
                            color_string[yellow];
                            color_string[blue] ];
```

Record Constructor

A *record constant* is a declared constant defined with a *record constructor* that specifies values for the fields of a record type. A record constant may be used to initialize a variable in the body of a block. Individual fields of a record constant in the body of a block may be selected, but not when defining other constants.

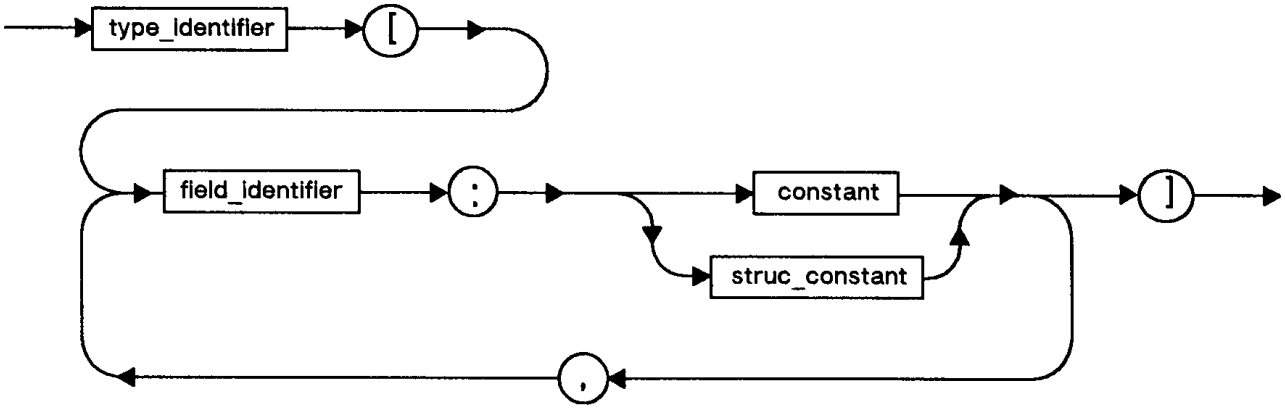
A *record constructor* consists of a previously declared record type identifier and a list in square brackets of fields and values. All fields of the record type must appear, but not necessarily in the order of their declaration. Values in the construct or must be assignment compatible with the fields. Note that a record constructor is only legal in the CONST section of a declaration part. It cannot appear in other sections or in an executable statement.

For records with variants, the constructor must specify the tag field before any variant fields. Then only the variant fields associated with the value of the tag may appear. For records with tagless variants, the initial variant field selects the variant.

The values may be constant values or constructors. To use a constructor as a value, the field in the record type must be defined with a type identifier. A record constant may not contain a file.

Syntax

```
Record_constructor:
```



Example

```
TYPE
   securtype = (light, medium, heavy);
   counter   = RECORD
                 pages: integer;
                 lines: integer;
                 characters: integer;
               END;
   report    = RECORD
                 revision: char;
                 price:    real;
                 info:     counter;
                 CASE securtag: securtype OF
                   light:   ();
                   medium:  (mcode: integer);
                   heavy:   (hcode:    integer;
                             password: string[10]);
               END;

CONST
   no_count   = counter [pages: 0, characters: 0, lines: 0];
   big_report = report [revision: 'B',
                        price:    19.00,
                        info:     counter [pages:      19,
                                           lines:      25,
                                           characters: 900],
                        securtag: heavy,
                        hcode:    999,
                        password: 'unity'];

   no_report  = report [revision : ' ';
                        price    : 0.00;
                        info     : no_count;
                        securtag : light];
```

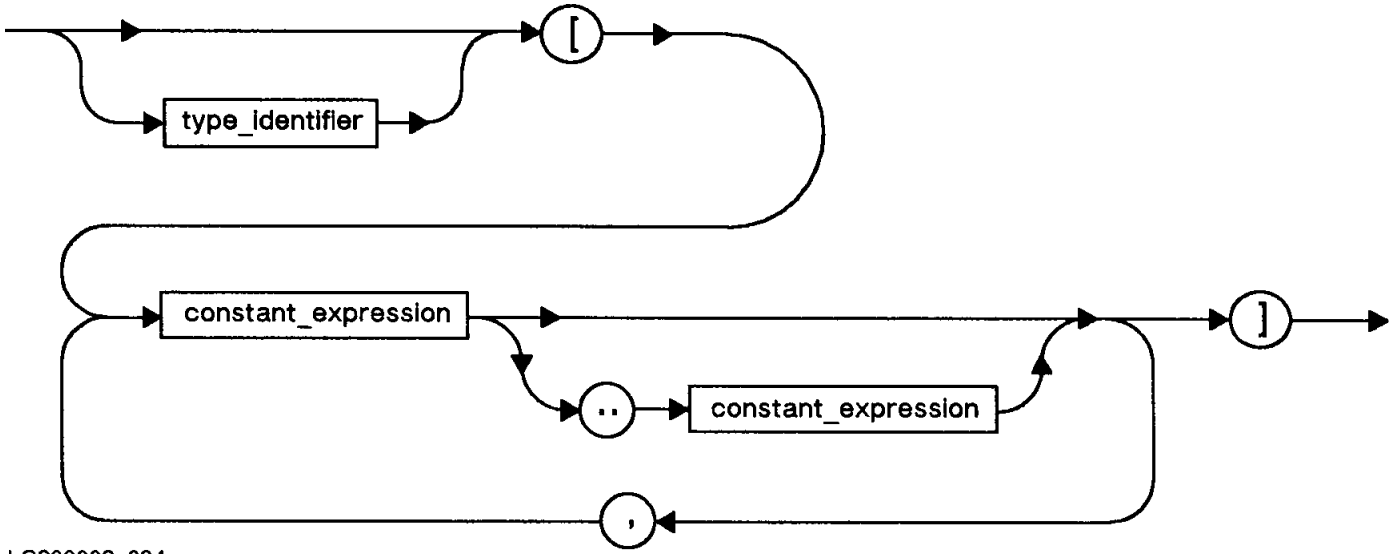
Restricted Set Constructor

A set constant is a declared constant defined with a *restricted set constructor* that specifies set values. A restricted set constructor consists of an optional previously declared set type identifier and a list of constant values in square brackets. Subranges may appear in this list. Restricted set constructors may appear in a CONST section of a declaration part, or in executable statements and can be used to initialize a set variable in the body block.

The constant must be an ordinal constant value or an ordinal subrange. A constant expression is legal as a value. The symbols (. and .) may replace the left and right square brackets, respectively.

Syntax

```
Restricted_set_constructor:
```



Example

```
TYPE
   digits  = SET OF 0..9;
   charset = SET OF char;

CONST
   all_digits = digits [0..9];              { Subrange. }
   odd_digits = digits [1, 1+2, 5, 7, 9];
   letters    = charset ['a'..'z', 'A'..'Z'];
   no_chars   = charset [ ];
   no_iden    = [2, 4, 6, 8]       { No set identifier. }
```

String Constructor

A string constant is a declared constant defined with a *string constructor* that specifies values for a string type. The length of the string constant may not exceed the maximum length of the string type used in its definition. The number of characters in the definition determines the current length of the string constant.

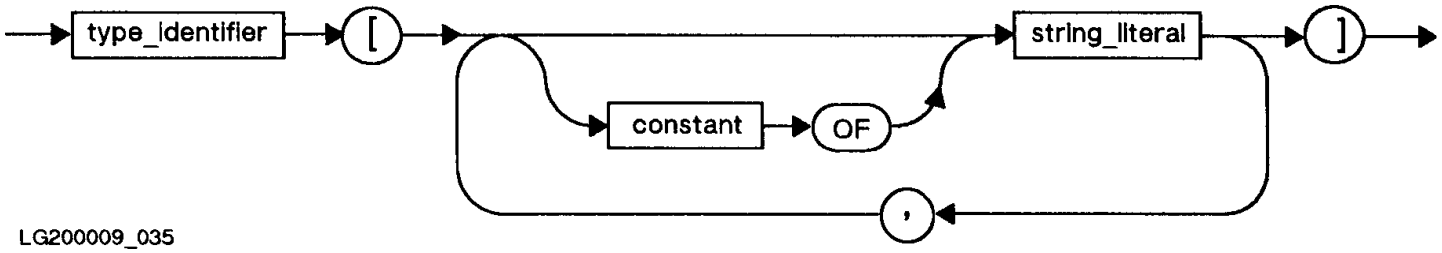
A string constructor consists of a previously defined string type identifier and a list of values in square brackets. Note that string constructors are only legal in a CONST section of a declaration part. They cannot appear in other sections or in executable statements.

Within the square brackets, the reserved word OF indicates that a value occurs repeatedly. For example, 3 OF 'a' assigns the character a to three successive string components. The symbols (. and .) may replace the left and right square brackets, respectively. String literals of more than one character may appear as values.

A string constant may be used to initialize a variable in the statement part of a block. Individual components of a string constant in the body of the block may be accessed, but not in the definition of other declared constants.

Syntax

String_constructor:



Example

```
TYPE
   s = string[80];

CONST
   blank = ' ';
   greeting = s['Hello!'];
   farewell = s['G',2 OF 'o','d','bye'];
   blank_string = s[10 OF blank];
```

Label Declaration

A *label declaration* specifies integer labels that mark executable statements in the body of the block. The reserved word LABEL precedes one or more integers separated by commas. Control is transferred to a labeled statement by a GOTO statement. For more information about GOTO statements, see Chapter 6 .

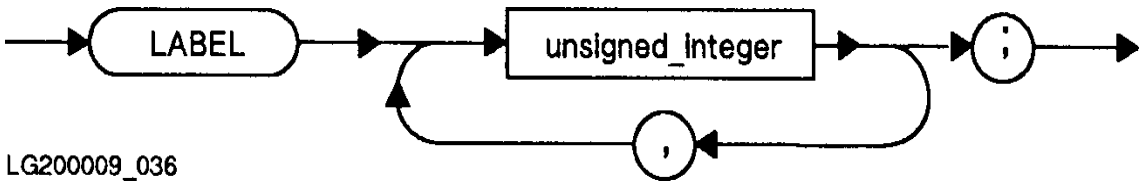
Integers must be in the range 0 to 9999. Leading zeros are not significant. For example, the labels 9 and 00009 are identical.

In HP Pascal, label declarations must come first in the declaration part of a block.

A label must occur in the block of the procedure, function, or program where the label is declared. For every label there must be one and only one statement with that label.

Syntax

Label_decl:



Example

```
LABEL 9, 19, 40;
.
.
.
40 : x:=10;
.
.
GOTO 40;
```

Type Definition

A TYPE section introduces the name and set of values for a user-defined type. HP Pascal requires that a *type identifier* be defined before its

subsequent use in the definitions of other types. In the only exception to this rule, a base type identifier in a pointer type definition is allowed before the base type is defined. However, the base type must be defined before the end of the TYPE section in which it is first mentioned.

TYPE

This reserved word delimits the start of the *type* declarations in a program, module, procedure, or function. A type definition establishes an identifier known as *type identifier* as a synonym for a *data type*. The identifier may then appear in subsequent type or constant definitions or in variable declarations.

The reserved word TYPE precedes one or more type definitions. A *type definition* consists of an identifier, the equals sign (=), and a type.

A *data type* determines a set of attributes that includes the following:

* The set of permissible values.
* The set of permissible operations.
* The amount of storage required.

The three most general categories of data type are *simple*, *structured*, and *pointer*.

Simple data types are the types *ordinal*, *real*, or *longreal*. Ordinal types include the standard types *integer*, *char*, *bit16*, *bit32*, *bit52*, *shortint*, *longint*, and *boolean* as well as user-defined *enumerated* and *subrange* types.

Structured data types are the types *array, record, set,* and *file*. The standard type *string* is also a structured data type. The standard type *text* is a variant of the file type.

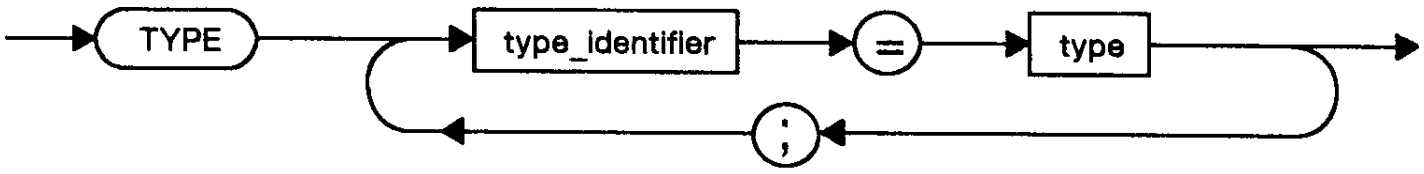
Pointer data types define pointer *variables* that point to dynamically allocated variables on the heap. For a detailed description of HP Pascal data types, refer to Chapter 3 .

CONST, TYPE, VAR, MODULE, and IMPORT sections may be intermixed.

Syntax

Type_decl:

.



Example

```
TYPE
   units = (inches,feet,miles);              { Simple type     }
   files = ARRAY [1..10] OF text;            { Structured type }
   PTR1  = ^units;                           { Pointer type    }
```

Variable Declaration

A *variable declaration* introduces an identifier as a variable of a specified type. Each variable is a statically-declared object that occupies storage and is accessible for the activation and duration of the program, procedure, or function in which it is declared.

Components of a structured variable may be accessed using an appropriate selector. Pointer variable dereferencing accesses dynamic variables on the heap. Module variables are accessible for the duration of the

program that imports the module.

Several identifiers may be combined in the same variable declaration if the variables are of the same type.

HP Pascal predefines two standard variables, *input* and *output*, that are textfiles. Formally,

```
VAR
  input, output: text;
```

These standard *textfiles* commonly appear as program *parameters* and serve as default files for various file operations. For more information on textfiles, refer to Chapter 3 .

Every declaration of a file variable F with components of type T implies the additional declaration of a buffer variable of type T. The buffer variable, denoted as F^, may be used to access the current component of the file F.

Global Variables

Global variables are declared at the beginning of the outermost block of a program and are available to all the procedures and functions within that program.

Local Variables

Local variables are variables declared within a particular procedure or function or in the headings as parameters, and their scope is limited to that procedure or function during the execution of the procedure or function. When optimization is requested, the compiler will issue warnings about local variables that are used prior to their initialization.

Module Variables

Module variables are declared in either the EXPORT or IMPLEMENT section of a module. Variables declared in the EXPORT part are available to all the procedures and functions within the program which imports the modules. Those declared in the IMPLEMENT section are only available inside the module.

VAR

This reserved word delimits the beginning of variable declarations in an HP Pascal program or module. A variable declaration associates an identifier with a type. The identifier may then appear as a variable in executable statements.

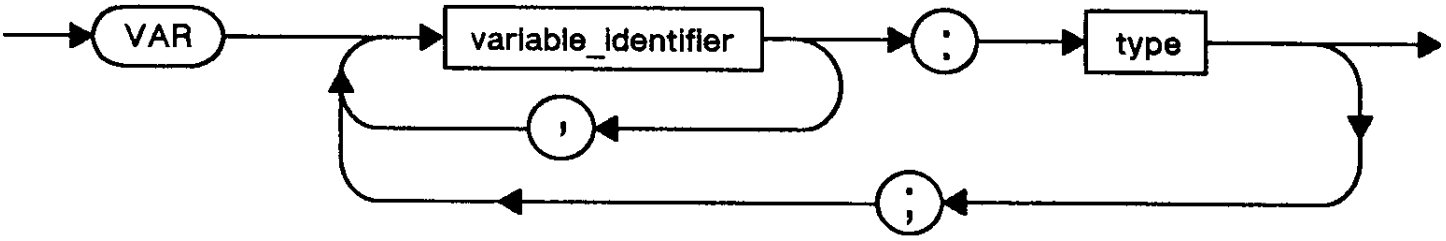
The reserved word VAR precedes one or more variable declarations. A variable declaration consists of an *identifier*, a *colon* (:), and a *type*. Any number of identifiers may be listed, separated by commas. These identifiers are then variables of the same type.

The type may be any *simple, structured,* or *pointer* type. The form of the type may be a *standard* identifier, a *declared* type identifier, or a *data* type.

VAR sections may be repeated and intermixed with CONST, TYPE, MODULE, and IMPORT sections.

Syntax

```
Variable_decl:
```



Example

```
TYPE
   answer = (yes, no, maybe);

VAR
   pagecount,
   linecount,
   charcount: integer;            { Standard identifier.      }
   whats_the: answer;             { User-declared identifier. }

   album    : RECORD              { Data type.                }
                 speed: (lp, for5, sev8);
                 price: real;
                 name : string[20];
              END;
```

Side-Effects

A *side-effect* is the modification by a procedure or function of a variable that is global or nonlocal in scope to the procedure or function. If a local variable is declared using the same identifier as a global variable, the local variable may be modified without affecting the global variable.

Example

```
PROGRAM show_effects(output);

VAR
   i,j:  integer;                        { Global variables }

PROCEDURE oops(i : integer);  { i is local to the procedure }

BEGIN
   IF i > 0 THEN j := j - 1;       { j is a global variable }
END;

BEGIN
   i := 2;
   j := 3;
   oops(i);
   IF i = j THEN writeln('There was a side effect.');
END.
```

Output:

```
There was a side effect.
```

NOTE Side effect modifications may cause an optimizer to be more conservative in its choices for code improvement, thereby decreasing execution performance.

Chapter 6 Statements

A *statement* is a sequence of special *symbols*, reserved *words*, and *expressions* that either performs a specific set of actions on a program's data or controls program flow. Table 6-1 lists and describes statements.

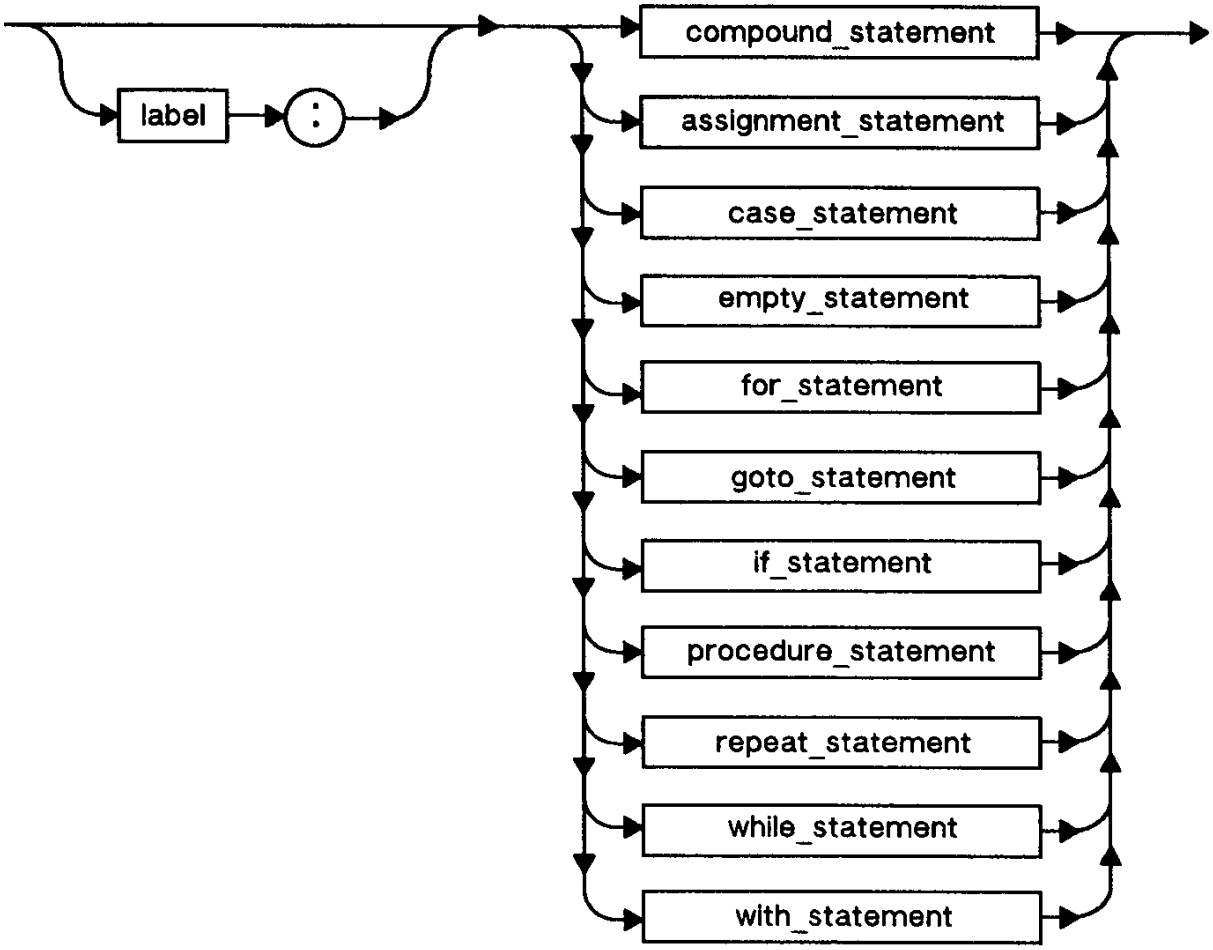
Table 6-1. HP Pascal Statements and Purposes

Statement Type	Purpose
compound	Group statements
empty	Do nothing
assignment	Assign a value to a variable
procedure	Invoke a procedure
GOTO	Transfer control unconditionally
IF, CASE	Conditional selection
WHILE, REPEAT, FOR	Iterate a group of statements
WITH	Manipulate record fields

The empty, assignment, procedure, and GOTO statements are commonly called *simple* statements. The *compound*, IF, CASE, WHILE, REPEAT, FOR, and WITH statements are referred to as *structured statements* because they themselves may contain other statements.

Syntax

Statements:



Compound Statements

A *compound* statement is a sequence of statements bracketed by the reserved words BEGIN and END. A semicolon (;) delimits one statement from the next. Certain statements may alter the flow of execution in order to achieve effects such as selection, iteration, or invocation of another procedure or function. For instance, after the last statement in the body of a routine has executed, control is returned to the point in the program from which the routine is called. Note the use of non-local GOTOs voids this statement. The program terminates after the last statement is executed.

A compound statement has two primary uses. First, it defines the statement part of a block and second, it groups a series of statements into a single statement. A compound statement may also serve to logically group a series of statements.

Note that compound statements are allowed, but are unnecessary in the following cases:

* The statements between REPEAT and UNTIL.
* The statements between OTHERWISE and the end of the CASE statement.

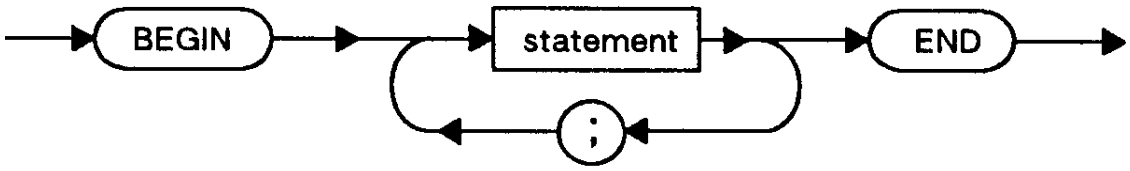
Example

```
PROCEDURE check_min;

BEGIN                                                    { This        }
   IF min > max THEN                                     { compound    }
      BEGIN                              { Compound     } { statement   }
         writeln('Min is wrong.');       { statement is } { is          }
         min := 0;                       { part of IF   } { the         }
      END;                               { statement.   } { procedure's }
END;                                                     { body.       }
. . .

BEGIN                           { Nested compound statements         }
   IF part_to_start=part_1 THEN
      BEGIN                        { for logically grouping statements. }
         start_part_1;
         finish_part_1;
                                { empty statement here }
      END
   ELSE
      BEGIN
         start_part_2;
         finish_part_2;
      END;
END;
...
```

BEGIN .. END

BEGIN and END are reserved words that signify the beginning and ending of a compound statement or block. BEGIN indicates to the compiler that a compound statement or block has started, whereas END indicates that a compound statement or block has terminated.

Syntax



Example

```
PROGRAM show_begin_end(input, output);

VAR
   running : Boolean;
   i, j    : integer;

BEGIN                 {begin of program block}
   i := 0;
   j := 1;
   running := true;
   writeln('See Dick run.');
   writeln('Run Dick run.');
   IF running then
      BEGIN           {begin of compound statement}
         i := i + 1;
         j := j - 1;
      END;            {end of compound statement}
END.                  {end of program block}
```

Output:

```
See Dick run.
Run Dick run.
```

Empty Statements

The *empty* statement causes only the advancement of program flow to the next statement. It is often used to indicate that nothing occurs. In the example, no action occurs when i equals 2, 3, 4, 6, 7, 8, 9, or 10.

Example

```
CASE i OF
   0    : start;
   1    : proceed;
   2..4 : ;
   5    : report_error;
   6..10: ;
   11   : stop;
   OTHERWISE fatal_error;
END;

IF i IN [2..4,6..10] THEN
   { do nothing }
ELSE
   { cases }
```

NOTE In the following example, the last semicolon is not required. Its presence means that there is an empty statement before END. If the semicolon were removed, there would not be an empty statement. Empty statements do not affect the run-time speed of your program.

```
BEGIN
   I:= J + 1;
   K:= I + J;
END
```

Assignment

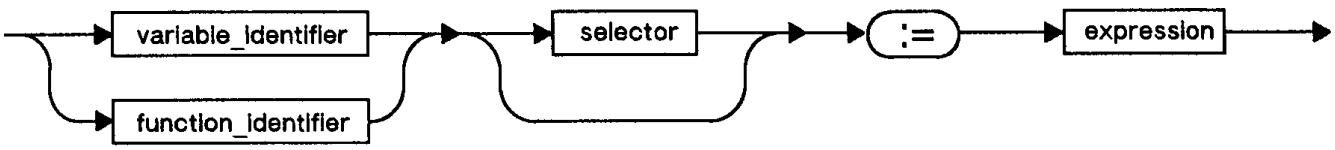
An *assignment* statement assigns a value to a variable access or a function result. The assignment statement consists of a variable or function *identifier,* an optional *selector,* a special *symbol* (:=), and an *expression* that computes a value. The type of the expression must be assignment compatible with the type of the receiving element.

The receiving element may be of any type except file, or a structured type containing a file type component. An appropriate selector permits assignment to a component of a structured variable or structured function result.

NOTE An implementation may evaluate the variable access and the expression in any order.

Syntax

Assignment_statement



Example

```
PROGRAM show_assign(input,output);

VAR
   aaa:  integer;

FUNCTION show_assign: integer;

TYPE
   rec = RECORD
            f: integer;
            g: real;
         END;

   index = 1..3;
   table = ARRAY [index] of integer;

CONST
   ct = table [10, 20, 30];
   cr = rec [f:2, g:3.0];

VAR
   s: integer;
   a: table;
   i: index;
   r: rec;
   p1,
   p: ^integer;
   strg: string[10];

BEGIN          { show_assign }
   s:= 5; i:= 3;
   a:= ct;
   a [i] := s + 5;
   r:= cr;
   r.f:= 5;
   new (p1);
   p:= p1;
   p^:= r.f - a [i];
   strg:= 'Hi!';
   show_assign := p^;
END;  {show_assign}

BEGIN
   aaa:= show_assign;
END.
```

CASE

The CASE statement selects a certain action based upon the value of an ordinal expression. It consists of the reserved word CASE, a selector, the reserved word OF, a list of case constants and statements, and the reserved word END. Optionally, the reserved word OTHERWISE and a list of statements may appear after the last constant and its statement.

The *selector* must be an ordinal expression in that it must return an ordinal value. A *case constant* may be a literal, a constant identifier, or a constant expression that is type compatible with the selector. Subranges may also appear as case constants.

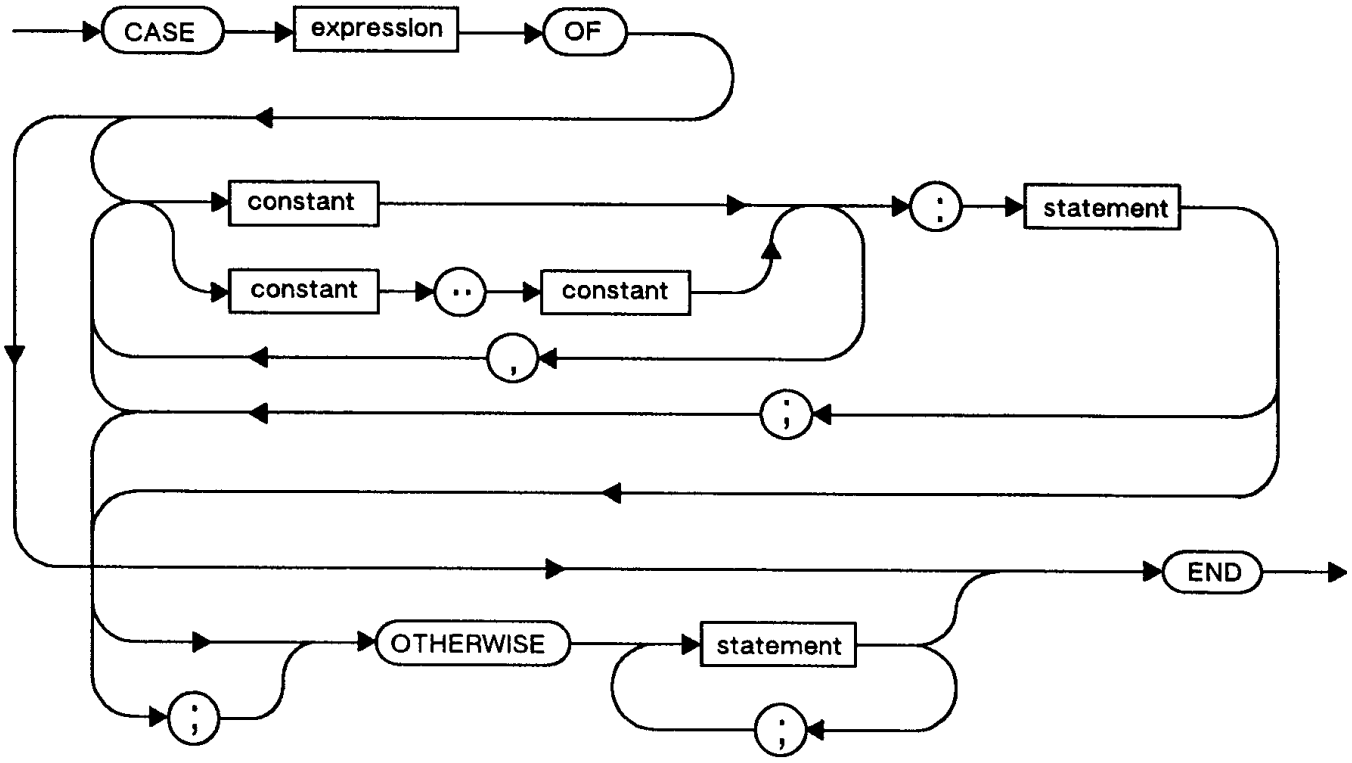
A case constant cannot appear more than once in a list of case constants. Subranges used as case constants may not overlap other constants or subranges. However, several constants may be associated with a particular statement by listing them separated by commas.

Note that statements between OTHERWISE and END need not be bracketed with BEGIN..END.

When the system executes a CASE statement, the following occurs:

1. It evaluates the selector.
2. If the value corresponds to a specified case constant, it executes the statement associated with that constant. Control then passes to the statement following the CASE statement.
3. If the value does not correspond to a specified case constant, it executes the statements between OTHERWISE and END. Control then passes to the statement after the CASE statement. The OTHERWISE clause must be present or the selector must match any CASE label.

Syntax



Example

```
PROCEDURE scanner;

BEGIN
   get_next_char;
   CASE current_char OF
      'a'..'z',                  { Subrange CASE label }
      'A'..'Z':
         scan_word;

      '0'..'9':
         scan_number;

      OTHERWISE scan_special;
```

```
      END;
   END;
   . . . .

   FUNCTION octal_digit (d:digit): Boolean; { TYPE digit = 0..9 }

   BEGIN
      CASE d OF
         0..7: octal_digit := true;
         8..9: octal_digit := false;
      END;
   END;
   . . . .

   FUNCTION op   { TYPE operators=(plus,minus,times,divide) }
      (operator: operators;
      operand1,
      operand2: real)
      : real;

   BEGIN
      CASE operator OF
         plus:   op := operand1 + operand2;
         minus:  op := operand1 - operand2;
         times:  op := operand1 * operand2;
         divide: op := operand1 / operand2;
      END;
   END;
```

IF .. THEN

IF .. THEN .. ELSE

An IF statement specifies a statement the system executes, if a particular condition is *true*. If the condition is *false,* then the system doesn't execute that statement, and optionally, it executes another statement starting after the ELSE.

The IF statement consists of the reserved word IF, a Boolean expression, the reserved word THEN, a statement, and, optionally, the reserved word ELSE and another statement. The statements after THEN or ELSE may be any HP Pascal statements, including other IF statements or compound statements. No semicolon separates the first statement and the reserved word ELSE.

When an IF statement is executed, the Boolean expression is evaluated to either *true* or *false*, and one of the following three actions is performed:

* If the value is true, the statement following THEN is executed.

* If the value is false and ELSE is specified, the statement following the ELSE is executed.

* If the value is false and no ELSE is specified, execution continues with the statement following the IF statement.

The following IF statements are equivalent:

```
IF a = b THEN
   IF c = d THEN
      a := c
   ELSE
      a := e;
```

```
IF a = b THEN
   BEGIN
      IF c = d THEN
         a := c
      ELSE
         a := e;
   END;
```

NOTE ELSE parts are always associated with the nearest preceding unmatched IF statement.

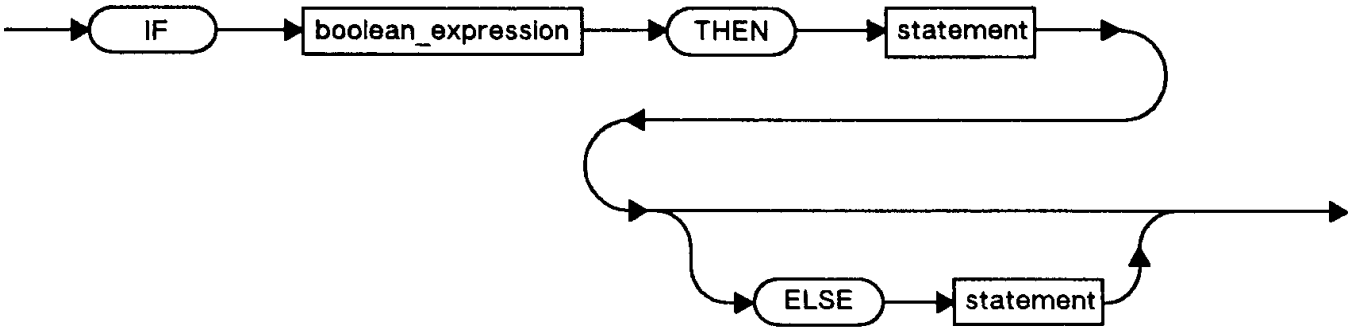
A common use of the IF statement is to select an action from several choices. This often appears in the following form:

```
IF e1 THEN
 ...
ELSE IF e2 THEN
 ...
ELSE IF e3 THEN
 ...
ELSE
 ...
```

This form is particularly useful to test for conditions involving real numbers or string literals of more than one character, since these types are not legal in CASE labels.

Syntax

If_statement



Example

```
PROGRAM show_if (output);

VAR
   i,j  :  integer;
   s    :  PACKED ARRAY [1..5] OF char;
   found:  Boolean;

BEGIN
   .
   .
   IF i = 0 THEN writeln ('i = 0');    { IF with no ELSE.       }
   IF found THEN                       { IF with an ELSE part.  }
      writeln ('Found it')
   ELSE
      writeln ('Still looking');
     .
     .
   IF i = j THEN                       { Select among different }
      writeln ('i = j')                { Boolean expressions.   }
   ELSE IF i < j THEN
      writeln ('i < j')
   ELSE { i > j }
      writeln ('i > j');
     .
```

```
        .
      IF s = 'RED' THEN                  { This IF statement      }
         i := 1                          { cannot be rewritten as }
      ELSE IF s = 'GREEN' THEN           { a CASE statement.      }
         i := 2
      ELSE IF s = 'BLUE' THEN
         i := 3;
   END.
```

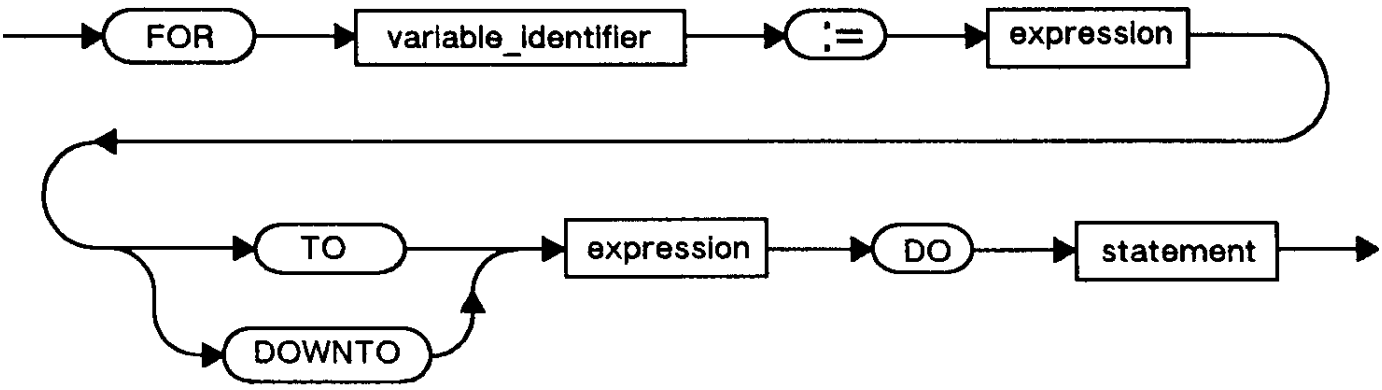
FOR .. DO

The FOR statement executes a statement a predetermined number of times. The FOR statement consists of the reserved word FOR, a control variable initialized by an ordinal expression known as the *initial value*, either the reserved word TO indicating an increment or the reserved word DOWNTO indicating a decrement, another ordinal expression known as the *final value*, the reserved word DO, and a statement.

The control variable is assigned each value of the range during the corresponding iteration of the statement. It must be an ordinal variable and may not be a component of a structured variable or a locally declared procedure or function parameter. The control variable may be a local or global variable. Non-local variables are not allowed. The initial and final values are ordinal expressions that must be assignment compatible with the control variable. After completion of the FOR statement, the control variable is undefined.

Syntax

For_statement:



When the system executes a FOR statement, the following occurs:

1. It evaluates the initial and final values and assigns the initial value to the control variable.

2. It executes the statement after DO.

3. It repeatedly tests the current value of the control variable and final value for inequality, increments or decrements the control variable, and executes the statement after DO.

In a FOR..TO construction, the system never executes the statement after DO if the initial value is greater than the final value. In a FOR..DOWNTO construction, the statement is never executed if the initial value is less than the final value.

The FOR statement:

```
FOR control_var := initial TO final DO
   statement
```

is equivalent to the statement:

```
BEGIN
   temp1 := initial;        {No evaluation order is required     }
   temp2 := final;          {for temp1 and temp2.                }
   IF temp1 <= temp2 THEN
      BEGIN
         control_var := temp1;
         WHILE control_var <= temp2 DO
            BEGIN
               statement;
               control_var := succ(control_var);  { increment    }
            END;
      END
   ELSE;                    { Don't execute the statement at all;}
END;                        { control_var is now undefined.      }
```

The FOR statement:

```
FOR control_var := initial DOWNTO final DO
   statement
```

is equivalent to the statement:

```
BEGIN
   temp1 := initial;        {No evaluation order is required     }
   temp2 := final;          {for temp1 and temp2.                }
   IF temp1 >= temp2 THEN
      BEGIN
         control_var := temp1;
         WHILE control_var >= temp2 DO
            BEGIN
               statement;
               control_var := pred(control_var);  { decrement    }
            END;
      END
   ELSE;                    { Don't execute the statement at all;}
END;                        { control_var is now undefined.      }
```

In the statement after DO, it is an error if assignment is made to the control variable. It cannot be used on the left-hand side of an assignment statement, passed as a reference parameter or used as the control variable of a second FOR statement nested within the first. Furthermore, it may not appear as a parameter for the standard procedures *read* or *readln*.

The system determines the range of values for the control variable by evaluating the two ordinal expressions once, and only once, before making any assignment to the control variable. So the statement sequence:

```
i := 5;
FOR i := pred(i) TO succ(i) DO writeln('i=',i:1);
```

writes:

```
i=4
i=5
i=6
```

instead of:

```
i=4
i=5
```

Example

```
{ VAR color: (red, green, blue, yellow); }

FOR color := red TO blue DO
   writeln ('Color is ', color);
   .
   .
FOR i := 10 DOWNTO 0 DO
   writeln (i);
writeln ('Blast Off');
      .
      .
FOR i := (a[j] * 15) TO (f(x) DIV 40) DO
   IF odd(i) THEN
      x[i] := cos(i)
   ELSE
      x[i] := sin(i);
```

REPEAT .. UNTIL

A REPEAT statement executes a statement or group of statements repeatedly until a Boolean expression is *true*. It consists of the reserved word REPEAT, one or more statements, the reserved word UNTIL, and a Boolean expression (the condition). The statements between REPEAT and UNTIL need not be bracketed with BEGIN..END.

When the system executes a REPEAT statement, the following occurs:

1. It executes the statement sequence, and then evaluates the Boolean expression.

2. If it is false, it executes the statement sequence and evaluates the Boolean expression again.

3. If it is true, control passes to the statement after the REPEAT...UNTIL statement.

The statement:

```
REPEAT
   statement;
UNTIL condition
```

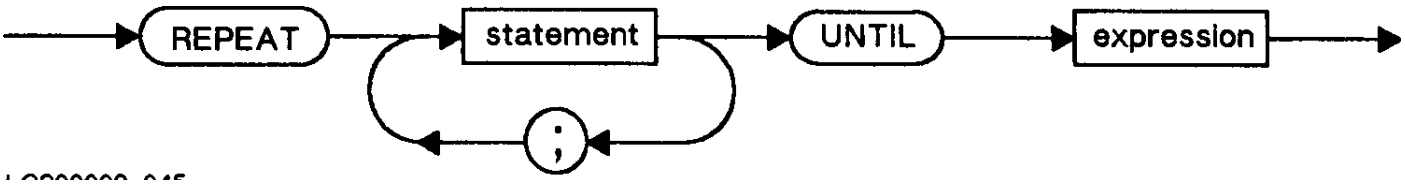
is equivalent to the following:

```
1: statement;
   IF NOT condition THEN GOTO 1;
```

Usually the statement sequence modifies data at some point so that the condition becomes *true*. Otherwise, the REPEAT statement loops forever.

Syntax

Repeat_statement:



Example

```
sum := 0;
count := 0;

REPEAT
   writeln('Enter trial value, or "-1" to quit');
   read (value);
   sum := sum + value;
   count := count + 1;
   average := sum / count;
   writeln ('value = ', value, '   average = ',average)
UNTIL (count >= 10) OR (value = -1);
   .
   .
REPEAT
   writeln (real_array[index]);
   index := index + 1;
UNTIL index > limit;
```

WHILE .. DO

The WHILE statement executes a statement repeatedly as long as a given condition is *true*. The WHILE statement consists of the reserved word WHILE, a Boolean expression (the condition), the reserved word DO, and a statement.

When the system executes a WHILE statement, the following occurs:

1. It evaluates the condition.

2. If the condition is true, it executes the statement after DO, and then re-evaluates the condition. When the condition becomes false, execution resumes at the statement after the WHILE statement.

3. If the condition is false at the beginning, the system never executes the statement after DO.

The statement:

```
WHILE condition DO statement
```

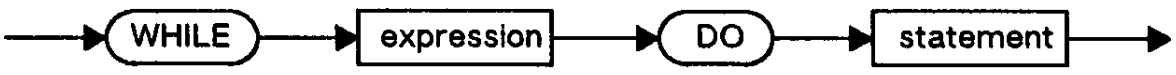
is equivalent to:

```
1: IF condition THEN
     BEGIN
       statement;
       GOTO 1;
     END;
```

Usually a program modifies data at some point so that the condition becomes *false*. Otherwise, the statement repeats indefinitely.

Syntax

While_statement



Example

```
WHILE index <= limit DO
   BEGIN
      writeln (real_array[index]);
      index := index + 1;
   END;
    .
    .
WHILE NOT eof (f) DO
   BEGIN
      read (f, ch);
      writeln (ch);
   END;
```

WITH .. DO

A WITH statement allows reference to *record fields* by field name alone. A WITH statement consists of the reserved word WITH, one or more record designators, the reserved word DO, and a statement. A record designator may be a record identifier, a function call that returns a record, or a selected record component.

The statement after DO may be a compound statement. In this statement, reference to a record field contained in one of the designated records can be made without mention of the record to which it belongs. The appearance of a function reference as a record designator is an invocation of the function. Note that a new value may not be assigned to a field of a record constant or a field of a record returned by a function.

When the program executes a WITH statement, the following occurs:

1. References to the record designators are evaluated.

2. The statement after the DO statement is executed.

The following statements are equivalent:

```
WITH rec DO                        BEGIN
   BEGIN                               rec.field1 := e1;
      field1 := e1;                    writeln(rec.field1
      writeln(field1 * field2);                * rec.field2);
   END;                            END;
```

Because the program evaluates a reference to a record designator once and only once before it executes the statement, the following statement sequences are equivalent:

f designates a field in the example above.

```
i := 1;
WITH a[i] DO
   BEGIN
      writeln(f);
      i:=2;
      writeln(f)
   END;

writeln(a[1].f);
writeln(a[1].f);  { NOT writeln(a[2].f) }
```

That is, within the WITH statement, the implied value of a[i] is not affected by the change to i.

Records with identical field names may appear in the same WITH statement. The following interpretation resolves any ambiguity.

The statement:

```
WITH record1, record2, ..., recordn DO
   BEGIN
      statement;
   END;
```

is equivalent to:

```
WITH record1 DO
  BEGIN
     WITH record2 DO
        BEGIN
           ...
           WITH recordn DO
              BEGIN
                 statement;
              END;
           ...
        END;
  END;
```

Therefore, if field f is a component of both record1 and record2, the compiler interprets an unselected reference to f as a reference to record2.f. The synonymous field in record1 can be accessed using normal field selection; for example, record1.f.

This interpretation also means that if r and f are records, and f is a field of r, the statement:

```
WITH r DO
   BEGIN
      WITH r.f DO
         BEGIN
            statement;
         END;
   END;
```

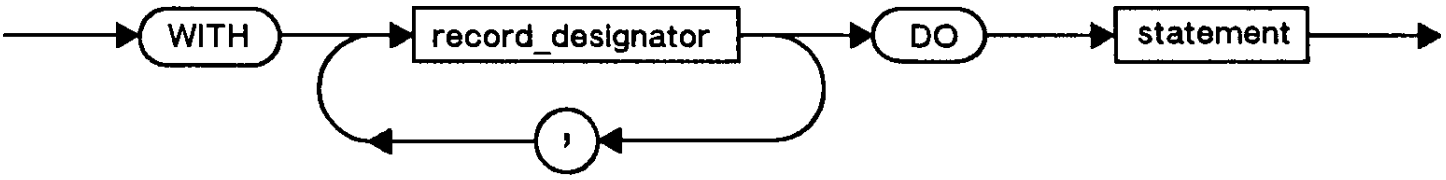
is equivalent to

```
WITH r,f DO
   BEGIN
      statement;
   END;
```

If a *local* or *global* identifier has the same name as a field of a designated record in a WITH statement, then the appearance of the identifier in the statement after DO is always a reference to the record field. The *local* or *global* identifier is inaccessible if it happens to have the same name as the field name in the record.

Syntax

With_statement



Example

```
PROGRAM show_with;

TYPE
   status = (married, widowed, divorced, single);
   date   = RECORD
               month  : (jan, feb, mar, apr, may, jun,
                         july, aug, sept, oct, nov, dec);
               day    : 1..31;
               year   : integer;
            END;
   person = RECORD
              name   : RECORD
                          first, last: string[10]
                       END;
              ss     : integer;
              sex    : (male, female);
              birth  : date;
              ms     : status;
              salary : real
            END;

VAR
   employee : person;

BEGIN {show_with}
   .
   WITH employee, name, birth DO
      BEGIN
         last := 'Hacker';
         first := 'Harry';
         ss := 214748364;
         sex := male;
         month := feb;
         day := 29;
         year := 1952;
         ms := single;
         salary := 32767.00
      END;
   .
END. {show_with}
```

GOTO

A GOTO statement transfers control unconditionally to a statement marked by a label. It consists of the reserved word GOTO and the specified label.

The scope of labels is restricted. They may only mark statements appearing in the executable portion of the block where they are declared. They cannot mark statements in inner blocks. GOTO statements, however, may appear in inner blocks and reference labels in an outer block. Therefore, it is possible to jump out of a procedure or function, but not into one.

A GOTO statement may not lead into a structured statement from outside that statement or from another component statement of that statement. For example, it is illegal to branch to the ELSE part of an IF statement from either the THEN part, or from outside the IF statement. Note that a GOTO statement that refers to a non-local label declared in an outer routine, causes any local files to be closed.

Labels are numeric values in the range 0 through 9999.

NOTE The use of the non-local label form of GOTO may increase execution time of the program.

Syntax

Goto_statement

Example

```
PROGRAM show_goto (output);

LABEL 500, 501;

TYPE
   index = 1..10;

VAR
   i: index;
   target: integer;
   a: ARRAY[index] OF integer;

PROCEDURE check;

VAR
   answer: string [10];

BEGIN
   .
   { ask user if OK to search }
   IF answer= 'no' THEN GOTO 501; { jumping out of procedure }
   .
END;

BEGIN { show_goto }
   .
   check;
   .
FOR i := 1 TO 10 DO
   IF target = a[i] THEN GOTO 500;
writeln (' Not found');
GOTO 501;
500:
   writeln (' Found');
501:
END. { show_goto }
```

Procedures

A *procedure statement* transfers program control to the block of a declared or standard procedure. After the procedure has executed, control is returned to the statement following the procedure call. A procedure statement consists of a procedure identifier and, if required, a list of actual parameters in parentheses.

The *procedure identifier* must be the name of a standard procedure or a

procedure declared in a previous procedure declaration. If a procedure declaration includes a formal parameter list, the procedure statement must supply the actual parameters. The actual parameters must match the formal parameters in number, type and order. There are four kinds of parameters: *value, reference, procedural,* and *functional*.

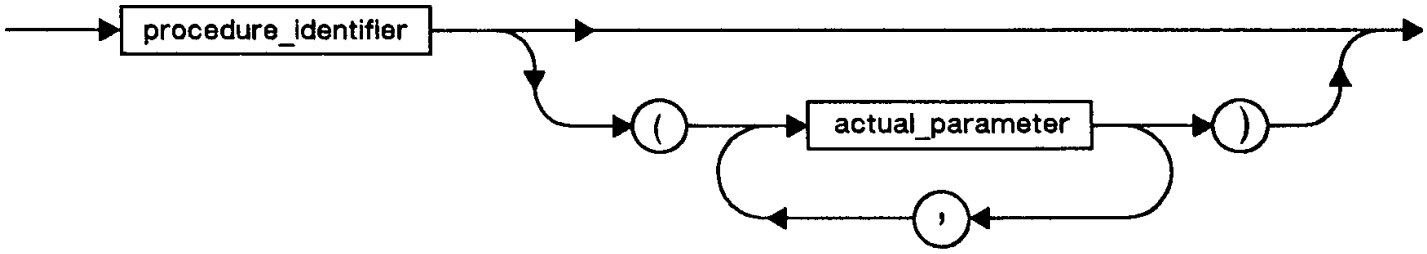
Actual *value parameters* are expressions that must be assignment compatible with the formal value parameters or, in the case of value conformant array parameters, conformable with the conformant array schema. Actual reference parameters are variables that must be type identical with the formal reference parameters or, in the case of reference conformant array parameters, conformable with the conformant array schema. Components of a packed structure cannot appear as actual procedural or functional para meters. Actual procedural or functional parameters are the names of procedures or functions declared in the program. Standard procedures or functions cannot be actual parameters to procedures or functions.

If a *procedure* or *function* that was passed as an actual parameter accesses any entity non-locally upon activation, then the entity accessed is one that is accessible to the procedure or function when it is passed as a parameter. For example, suppose Procedure A uses the non-local variable x. If A is then passed as an actual parameter to Procedure B, it is still able to use x, even if x is not otherwise accessible from B.

The formal parameters, if any, of an actual procedural or functional parameter must be congruent with the formal parameters of the formal procedural or functional parameter.

Syntax

Procedure_statement:



Example

```
PROGRAM show_pstate(output);

PROCEDURE wow;
   BEGIN
      writeln('wow');
   END;

PROCEDURE bow;
   BEGIN
      write('bow-');
      wow;
   END;

PROCEDURE outer  (a:  integer;
                  procedure proc_parm);

   PROCEDURE inner;
```

```
            BEGIN
               bow;
            END;

      BEGIN {outer}
         writeln('Hi');
         inner;
         proc_parm;
      END;  {outer}

      BEGIN { show_pstate }
         outer(30, bow);
      END.  { show_pstate }
```

Output:

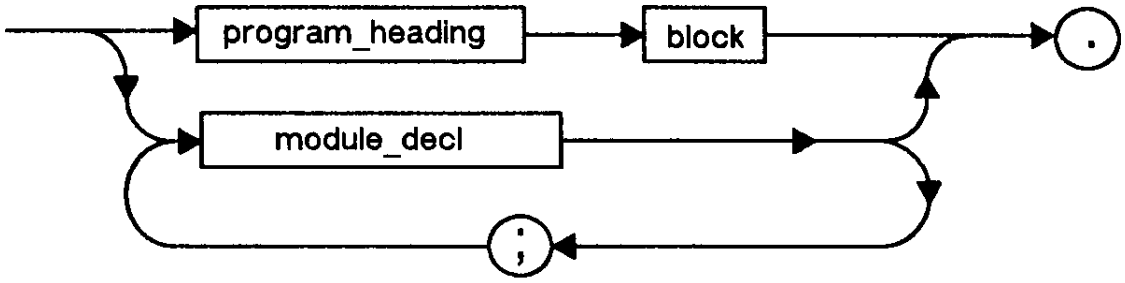
```
      Hi
      bow-wow
      bow-wow
```

Chapter 7 Program Structure

An HP Pascal program consists of two major parts: the program *heading* and the program *block*. The program *block* includes the declaration part which consists of definitions of constants and types, and declarations of labels, variables, procedures, functions, and modules. This chapter describes in detail the program *heading* and program *block*. This includes the declaration part and module as well as the function and procedure. Below is an example of an HP Pascal program.

Syntax

Compilation_unit:



Example

```
PROGRAM minimum;            { The minimum program that the HP Pascal }

BEGIN                       { compiler will process successfully:  }
END.                        { no program parameters. }

PROGRAM show_form1 (output); { Uses the standard textfile output   }

BEGIN
   writeln ('Greetings!')    { and the standard procedure writeln. }
END.

PROGRAM show_form2 (input,output);

VAR
   a,b,total: integer;

FUNCTION sum (i,j: integer): integer;   { Function declaration     }

BEGIN
   sum:= i + j
END;

BEGIN
   prompt ('Enter two integers: ');
   readln (a,b);
   total:= sum (a,b);
   writeln ('The total is: ', total)
END.
```

Program Heading

The *program heading* consists of the reserved word PROGRAM, an identifier that specifies the program name and an optional parameter list. The

program block consists of the declaration part and the statement or statements.

The identifiers in the parameter list are variables that must be declared in the outer block, except for the standard textfiles *input* and *output*.

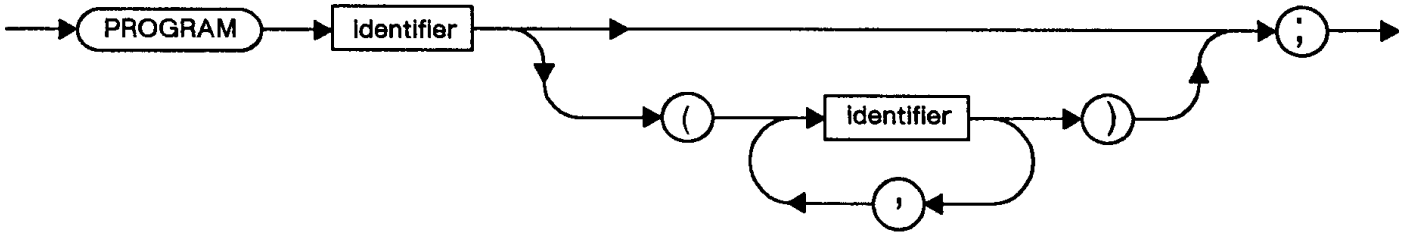
Input and *output* are standard file variables that the system associates by default with system dependent files. These files are opened automatically at the beginning of program execution. *Input* or *output* need only appear as program parameters if some file operation (for example, *read* or *write*) refers to them explicitly or by default.

Program parameters are usually the names of file variables. The association between logical and physical files is system-dependent. The association between formal and actual program parameters is also system-dependent.

The program block consists of an optional declaration part and a required statement part.

Syntax

Program_heading:

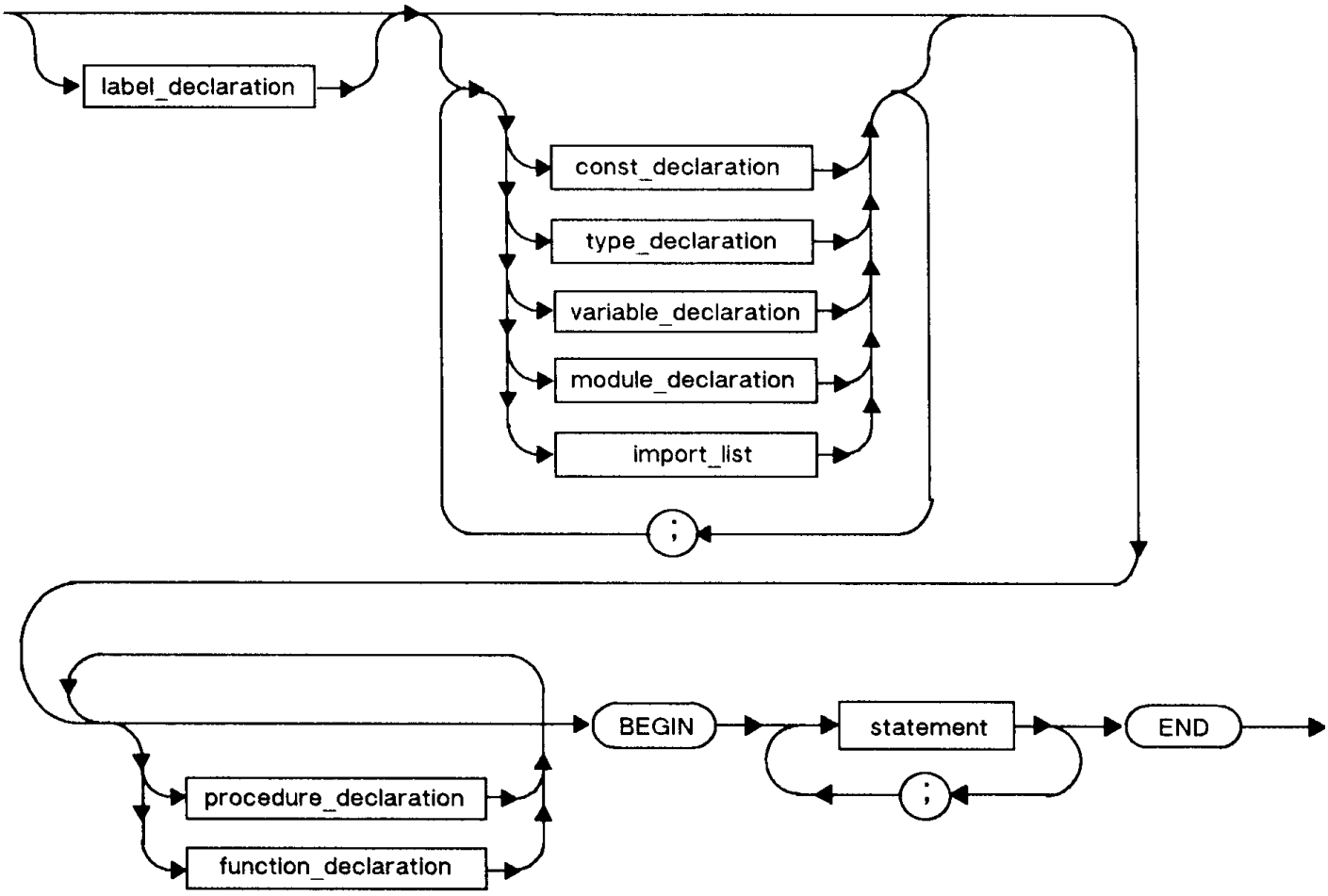


Block

A *block* is a syntactically complete section of code. There are two parts to a block; the *declaration* part and the *executable* part. Blocks may be *nested*. It is important that all objects appearing in the executable part be defined in the declaration part or in the declaration part of an outer block.

Syntax

block:



NOTE MODULE declarations and IMPORT lists cannot appear in inner blocks such as in procedures or functions.

Declaration Part

The *declaration part* consists of definitions of *constants* and *types*, and declarations of *labels*, *variables*, *procedures, functions*, and *modules*. The statement part is made up of a compound statement that may be empty or may contain several simple or structured statements. The statement part is also termed the body or executable portion of the block. For more information about statements, refer to Chapter 6 .

The reserved word LABEL precedes the declaration of labels. CONST or TYPE precedes the definition of declared constants or types. VAR precedes the declaration of variables. IMPORT precedes a list of imported module names. MODULE precedes the declaration of a module. PROCEDURE or FUNCTION precedes the declaration of a procedure or a function.

Within a declaration part, label declarations must come first, whereas procedure or function declarations come last. In HP Pascal, CONST, TYPE, IMPORT, VAR, and MODULE declarations may be intermixed and repeated. For more information on declarations, refer to Chapter 5 .

NOTE ANSI/IEEE770X3.97 - 1983 Standard Pascal allows the following reserved words, LABEL, CONST, TYPE, or VAR to be used only once in that order.

A predefined *constant, type, variable, procedure*, or *function* may be redeclared in a declaration part. However, access to the previous definition associated with that item is lost within the scope in which it is redefined.

Example

```
PROGRAM show_declarepart;

LABEL  25;

VAR
   birthday: integer;

TYPE
   friends = (Joe, Simon, Leslie, Jill);

CONST
   maxnuminvitee = 3;

VAR
  invitee: friends;
PROCEDURE hello;

BEGIN
   writeln('Hi');
END;                          { End of declaration part. }

BEGIN                         { Beginning of body.       }
 .
 .
END.
```

PROCEDURE

A *procedure* is a block that is invoked with a PROCEDURE statement. A procedure declaration consists of a procedure heading, a semicolon (;), and a block or a directive followed by a semicolon.

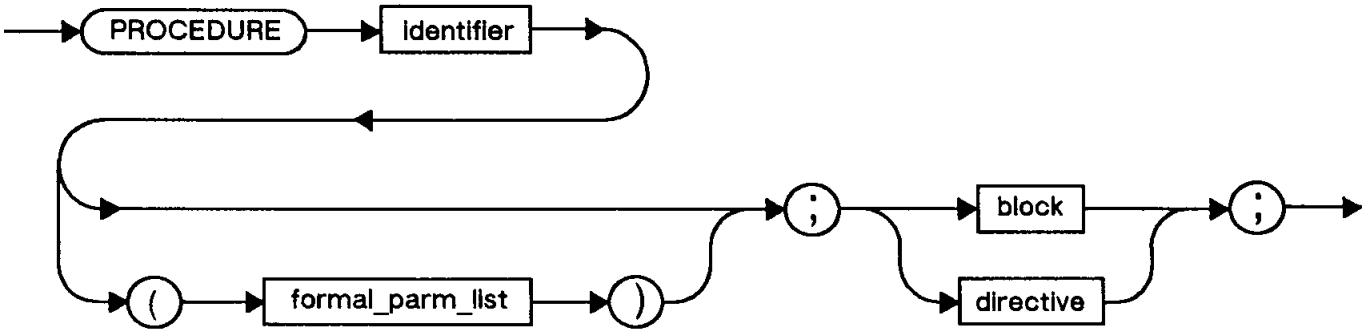
The procedure heading consists of the reserved word PROCEDURE, an identifier that specifies the procedure name, and optionally, a formal parameter list.

A *directive* can replace the procedure block to inform the compiler of the location of the block. FORWARD is one of the directives. Other directives are implementation dependent. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for information on other directives. A procedure block consists of an optional declaration part and a compound statement.

Procedure declarations must occur at the end of a declaration part after label, constant, type, and variable declarations and after the module declarations in the outer block. Note that procedure and function declarations may be intermixed.

Syntax

Procedure_declaration:



FUNCTION

A *function* is a block that is invoked with a function call and that returns a *value*. A function declaration consists of a function heading, a semicolon (;), and a block or a directive followed by a semicolon (;).

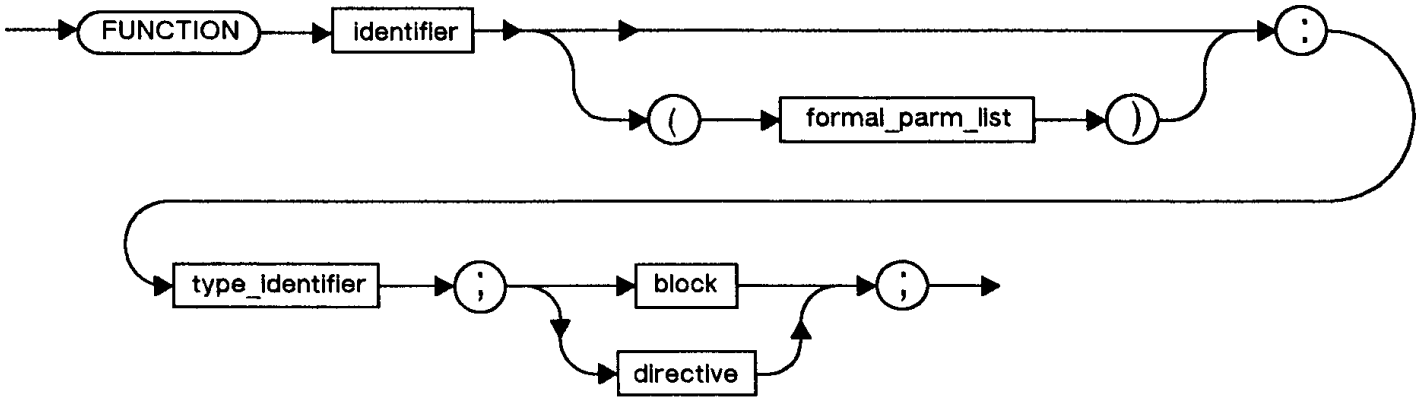
A *function heading* consists of the reserved word FUNCTION, an identifier that specifies a function name, an optional formal parameter list, and a result type. The result type may be any type, except a file type or a structured type containing a file.

A *directive* can replace the function block to inform the compiler of the location of the block; for example, FORWARD. Other directives are implementation dependent. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for information on other directives. In the body of a function block there must be at least one statement assigning a value to the function identifier. This assignment statement determines the function result. If the function result is a structured type, a value must be assigned to each of its components using an appropriate selector.

Function declarations may occur at the end of a declaration section after label, constant, type, variable declarations, and MODULE declarations at the outer level. Function declarations may be intermixed with procedure declarations.

Syntax

Function_declaration:



MODULE

A *module* provides a mechanism for separate compilation of program segments. It is a program fragment with a completely defined interface that can be compiled independently and later used to construct programs. A module usually defines some data *types*, *constants*, *variables*, and some *procedures* and *functions* that operate on this data. Such definitions are made accessible to users of the module by its export declarations. Modules can only access data or procedures in other modules and then only by importing them.

Any module used by a program, whether appearing in the program's globals or compiled separately, must be named in an import declaration. The objects that modules export always belong to the global scope of the importer.

The source text input to a compiler that is the complete unit of compilation may be a program or a list of modules separated by semicolons (;). An implementation may allow only a single module to be compiled at a time, thus requiring multiple invocations of the compiler to process several modules. The input text is terminated by a period.

A module cannot be imported before it has been compiled, either as part of the importing program or by a previous invocation of the compiler. This prevents construction of mutually-referring modules. Access to separately compiled modules is discussed below.

Although a module declaration defines data and procedures that become *globals* of any program importing the module, not everything declared in the module becomes known to the importer. A module specifies exactly what is exported to the "outside world" and lists any other modules on which it is itself dependent.

The *export declaration* defines constants and types, declares variables, and gives the headings of procedures and functions whose complete specifications appear in the *implement* part of the module. It is only the items in the export declaration that become accessible to any other code that subsequently imports the module.

There need not be any procedures or functions in a module if its purpose is solely to declare types and variables for other modules.

Any constants, types, and variables declared in the *implement* part are not made known to importers of the module; they are only known inside the module, and outside it they are hidden. Variables of the *implement* part of a module have the same life time as global program variables, even though they are hidden.

Any procedures or functions whose headings are exported by the module must subsequently be completely specified in its *implement* part. In this respect, the headings in the export declaration are like FORWARD directives, and in fact the parameter list of such procedures need not be, but may be, repeated in the implement part. Procedures and functions that are not exported may be declared in the implement part; they are known only within the module and are hidden from the rest of the program.

Separately compiled modules are called *library modules*. To use library modules, a program imports them just as if they had appeared in the program block. Refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for further information.

When an *import declaration* is seen, a module must be found matching each name in the import declaration. If a module of the required name appears in the compilation unit before the import declaration, the reference is to that module. Otherwise, external module libraries must be searched. See "SEARCH" , and the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for

more information.

In order for a procedure in a module to read from input or write to output (for example, readln from *input* or writeln to *output*), that module must import the standard modules *stdinput* or *stdoutput*, respectively.

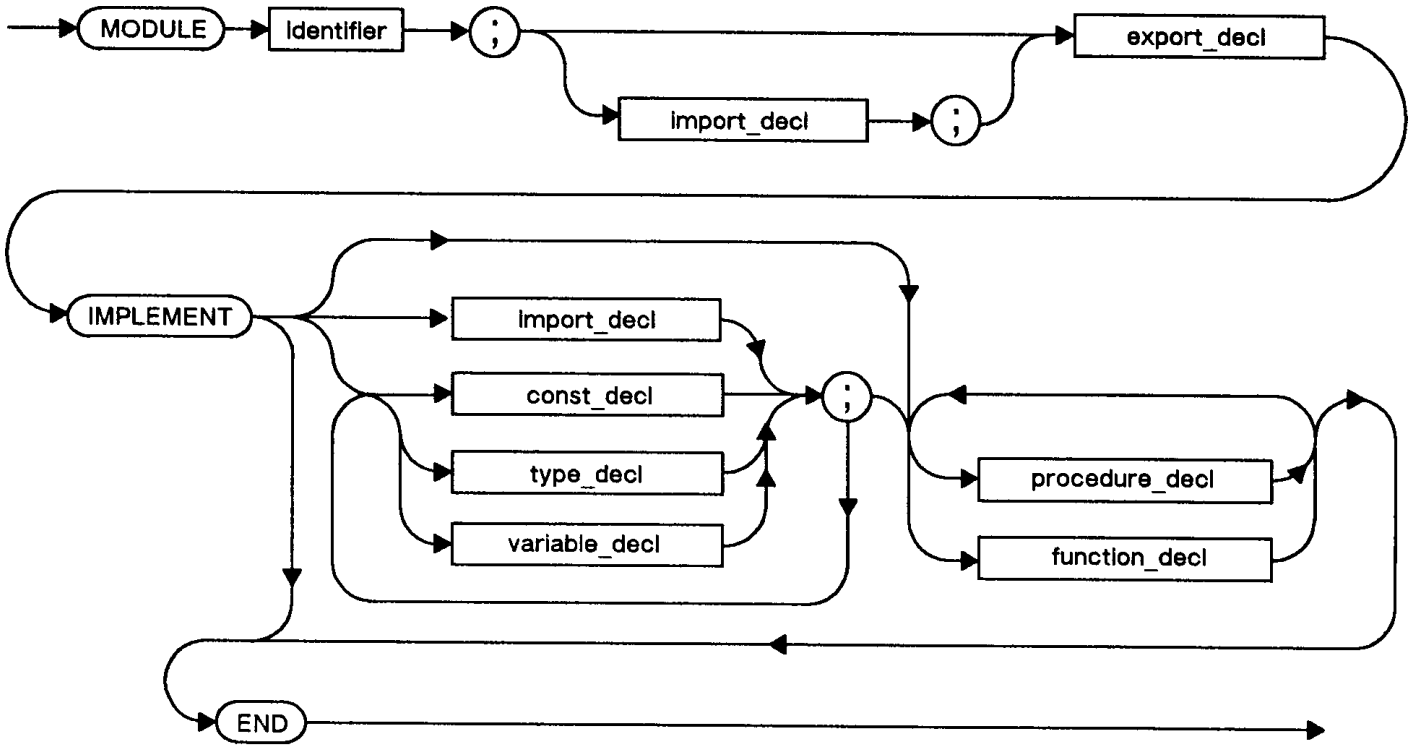
On HP-UX the standard modules *stdinput*, *stdoutput*, and *stderr* are contained in the predefined library /usr/lib/paslib. On MPE/iX the standard modules *stdinput* and *stdoutput* are contained in the predefined library PASLIB.PUB.SYS.

When a program, either directly or indirectly, imports a module that imports *stdinput* or *stdoutput*, the program must specify *input* or *output*, respectively, in the program parameter. If a program does not specify *input* or *output* and a module imports the standard modules, the program will not link.

On HP-UX only, if a program, either directly or indirectly, imports a module that imports *stderr*, the program must specify *stderr* in the program parameter. If a program does not specify *stderr* and a module imports the standard module *stderr*, the program will not link. In addition, if a procedure in a module writes to *stderr*, that module must import the standard module, *stderr*.

Syntax

Module_declaration:



Example

This example shows a source file that contains definitions for the modules bit_types and char_info. MODULE bit_types and MODULE char_info are compiled into an object file called mod1.o. Note that mod1.o is referenced in the examples in section "IMPORT" .

```
MODULE bit_types;                              { Module declaration     }
```

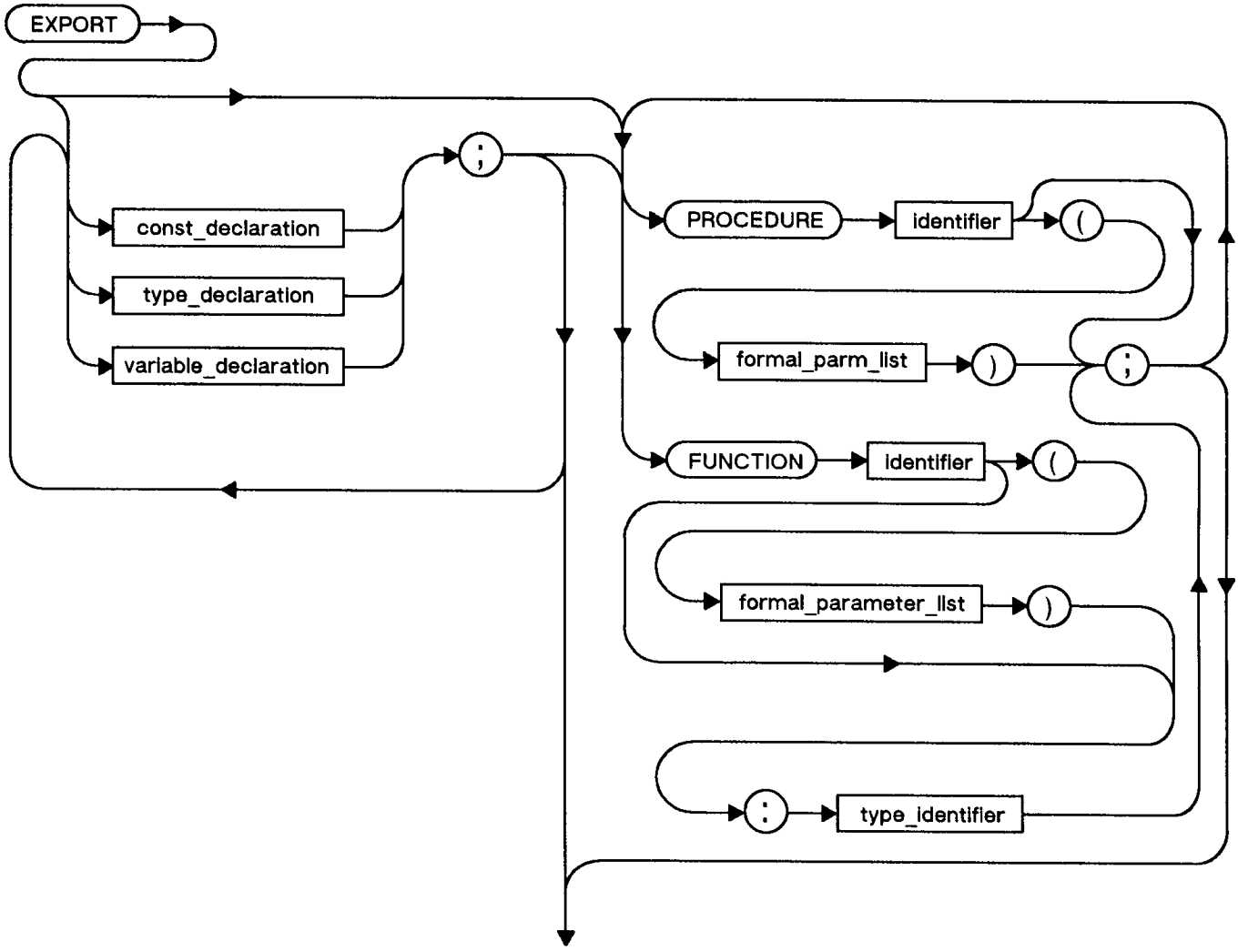
```
EXPORT                                          { Exported  types         }

   TYPE
      bits8 = 0..255;                           { Exported type           }

IMPLEMENT                                       { No implement, part, module  }
END;                                            { only provides data types    }

MODULE char_info;                               { Module declaration          }

IMPORT
   bit_types;                                   { Import other modules needed }
                                                { to compile this module      }
EXPORT                                          { Start of export text        }

   TYPE
      byte = bits8;                             { Exported type               }

   VAR

      last_byte: byte;                                    { Exported variable }

   FUNCTION control (i:byte; flag:BOOLEAN): BOOLEAN;   { Exported function }

IMPLEMENT                                       { Start of implementation     }

   IMPORT stdoutput;                            { Required for using output   }

   CONST
      error = 'non-ASCII character';            { Non-exported constant       }

   FUNCTION check (i: byte; flag: BOOLEAN): BOOLEAN; {Non-exported function}
   BEGIN
      IF i > 127 THEN
         BEGIN
            check := false;
            IF flag THEN writeln (error);
         END
      ELSE
         check := true;
   END;

   FUNCTION control (i: byte; flag: BOOLEAN): BOOLEAN;{ Exported function  }
   BEGIN
      last_byte := i;
      control := check (i,flag) AND (i < 32);
   END;
END.
```

EXPORT

This reserved word precedes the *constants*, *types*, *variables*, *procedures*, and *functions* of a MODULE that can be used or imported by other programs and modules. The EXPORT section is used to define the constants, types, variables, procedures, and functions that the module supplies to any program or module that imports it. Procedures and functions are presented as headings without blocks or directives. The EXPORT section may make use of things that were exported from modules listed in the IMPORT section. Every module must have an EXPORT section.

Syntax

```
Export_declaration:
```



Example

```
EXPORT                                          { Start of export text }

TYPE
   control_num: 0..255;                         { Exported type        }

VAR
   last_num: control_num;                       { Exported  variable   }

FUNCTION control (i: control_num; flag : Boolean) : Boolean;{Exported function}
```

IMPORT

This reserved word indicates which modules are needed to compile a program or module. The IMPORT section is used to name all other modules upon which the present one depends. One module, m1, depends on another, m2, if m1 makes use of the objects exported from m2. For instance, m1 calls procedures in m2, or assigns to m2's variables, or declares variables of a type exported from m2. There is no IMPORT section if the module is independent of all other modules.

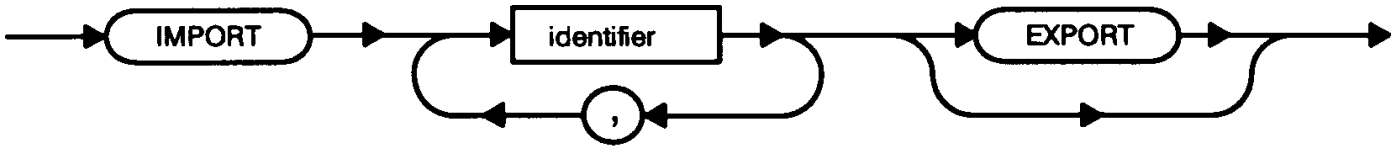
You must use $SEARCH to import a non-standard module that is not defined within the same compilation unit that contains the import statement. See "SEARCH" for more information.

When you want to export modules, as well as procedures and types, insert the reserved word EXPORT following the module name.

When EXPORT is used to specify an export of a module, that module is only available to the program or module importing the current module.

Syntax

Import_declaration:



Example 1

In this example, module bit_types is defined in another compilation unit (see example in section "MODULE"). bit_types is compiled into an object file called mod1.o. $SEARCH is used because bit_types is not in the same compilation unit as the main program.

```
PROGRAM show_import (output);

$SEARCH 'mod1.o'$     { Object file that contains bit_types.}

IMPORT                { Import the module bit_types, under  }
   bit_types;         { "Modules", that is needed to        }
                      { compile this program.               }
VAR
   A,B: bits8;

BEGIN
   A:= 100;
   writeln(A);
END.
```

Example 2

Module show_import_export both imports and exports module bit_types at the same time. The main program uses type bits8. bits8 is defined in bit_types, but is available to the main program because[REV BEG] it imports show_import_export which exports bit_types.

Module show_export is compiled into an object file called mod2.o and bit_types is compiled into an object file called mod1.o (see section "MODULE"). The main program imports module show_import_export only. However, the $SEARCH statement must include both object files mod1.o and mod2.p, even thought the main program does not directly import module bit_types.[REV END]

```
MODULE show_import_export;

$SEARCH 'mod1.o'$      {Object file that contains bit_types.}

IMPORT
   bit_types EXPORT;

EXPORT

   TYPE
     byte_rec = record;
       a, b : bits8
     end;
```

```
IMPLEMENT

END.

PROGRAM show_import_export_prog (output);

$SEARCH 'mod1.o, mod2.o'$      {Object files that contain bit_types}
                               {and show_import_export.            }
IMPORT
  show_import_export;

VAR

   little_bit  : bits8;        { bits8 is defined in module bit_types }
   little_byte : byte_rec;     {byte_rec is defined in module show_import_export}

   BEGIN
     little_bit  := 9;
     little_byte := little_bit;
   END.
```

IMPLEMENT

This reserved word indicates the beginning of the internal part of a MODULE. The IMPLEMENT section may be empty or it may contain declarations of the *constants, types, variables, procedures*, and *functions* that are only used within the module. In addition, it contains the bodies of the procedures and functions whose headings appeared in the EXPORT section. A module does not have to *export* procedures or functions. It may be used simply to *create* data or data types. In such a case, there will be nothing between the words IMPLEMENT and END. That is, every module must have an IMPLEMENT section, but it may be empty.

Example

```
MODULE A_module;

EXPORT                                    { Exported Type             }
   TYPE
      byte = 0..255;

   FUNCTION check (i:byte):Boolean;       { Exported Function         }

IMPLEMENT                                 { Start of implement section }

IMPORT stdoutput;

   FUNCTION check (i: byte;): Boolean;

   BEGIN

      IF i > 127 THEN
         BEGIN
           check := false;
           IF flag THEN
             writeln (error);
         END
      ELSE
          check := true;
   END;

FUNCTION control (i: byte; flag: Boolean):Boolean; {Exported function}

  BEGIN
    control := check (i,flage) AND (i < 32);
  END;

END.
```

Chapter 8 Procedures and Functions

When a procedure or function is declared, the heading may optionally include a list of *parameters*. This list is called the *formal parameter list*. A procedure statement or function call in the body of a block provides the matching actual parameters that correspond by their order in the list. The four kinds of formal parameters are *value, reference, functional*, and *procedural* parameters. *Value parameters* are identifiers followed by a colon (*:*) and a *type identifier* or a *conformant array schema*. *Reference parameters* are declared like value parameters, but are preceded by the reserved word VAR. *Functional* or *procedural parameters* are function or procedure headings.

The four types of formal parameters may be repeated and intermixed. Several identifiers may appear separated by commas. These identifiers then represent formal reference or value parameters of the same type, even if the type is a *conformant array schema*.

A *formal value parameter* appears as a local variable during execution of the procedure or function. It receives its initial value from the matching actual parameter. Modification of the formal parameter cannot affect the actual parameter which may be an *expression*. The actual parameter must be assignment compatible with the formal parameter or, in the case of a conformant array parameter, must conform with the formal parameter.

A *formal reference parameter* represents the actual parameter during execution of the procedure. Any changes in the value of the formal reference parameter alters the value of the actual parameter, which must be a variable access. The actual parameter must have a type identical with the formal parameter or conform with the formal parameter, in the case of a *conformant array schema*.

When a *conformant array schema* is specified, the value of the upper bound and the value of the lower bound identifiers in the schema vary according to the actual bounds of the array passed as the actual parameter. They can be accessed as value parameters in the procedure, except their values cannot be changed. Their names have the same scope as a parameter. The type of the actual parameters must be conformable with the conformant array schema. The formal parameters have a type that is distinct from any other type. This means that the actual parameters are not assignable to any other variable or parameter except those of the same type. The type cannot be a PAC type since the lower bound cannot be fixed as one. This makes passing string literals as actual conformant array parameters an error in ISO Pascal. HP Pascal is extended to allow the passing of string literals as parameters. However, a conformant array cannot be manipulated as a string.

An actual conformant array parameter can be passed as a reference conformant array parameter, but not as a value parameter of any kind.

A *formal procedural* or *functional parameter* is a synonym for the actual procedure or function parameter. The parameter lists, if any, of the actual and formal procedural or functional parameters must be congruent.

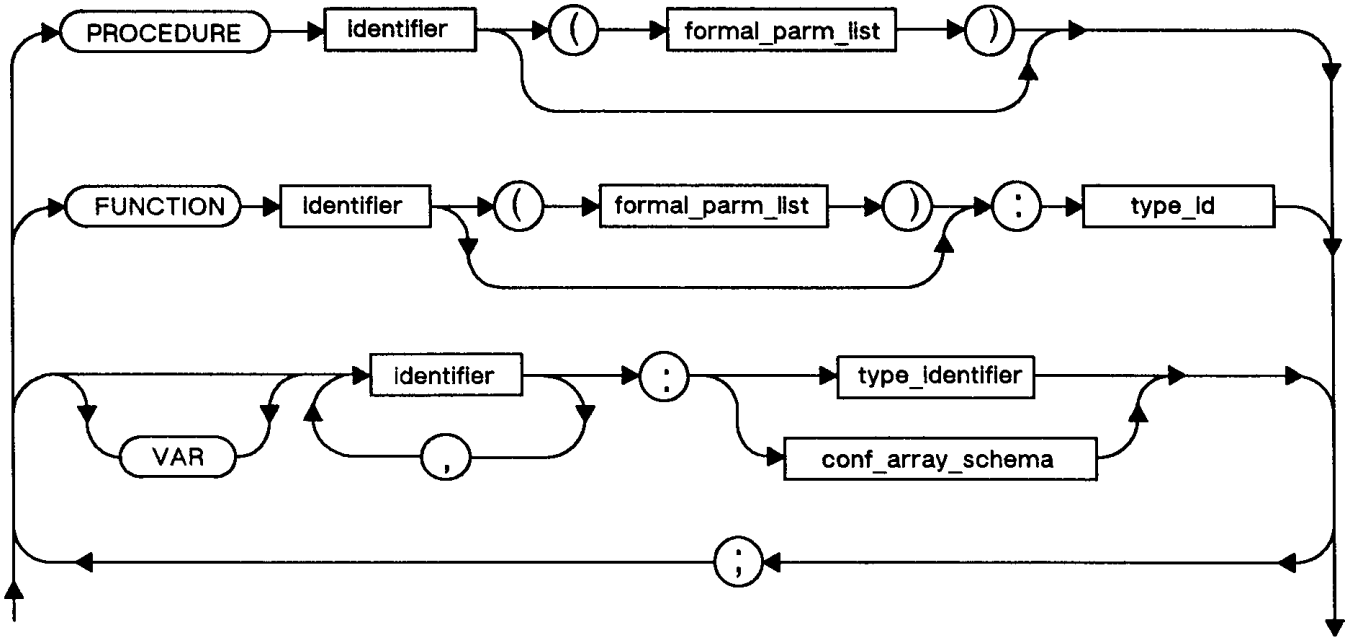
Two formal parameter lists are congruent if they contain an equal number of parameters, and the parameters in corresponding positions are equivalent. Two parameters are equivalent if any of the following conditions are *true*:

* They are both value parameters of the identical type.
* They are both reference parameters of the identical type.

* They are both procedural parameters with congruent parameter lists.
* They are both functional parameters with congruent parameter lists and identical result types.
* They are both either value conformant array specifications or both reference conformant array specifications, and in both cases, the conformant array specifications contain the same number of parameters and equivalent conformant array schemas. Two conformant array schemas are equivalent if all of the following statements are true:
    * The ordinal type identifier in each corresponding index type specification denotes the same type.
    * Either the component conformant array schemas of the conformant array schemas are equivalent, or the type identifiers of the conformant array schemas denote the same type.
    * Either both conformant array schemas are packed or both are unpacked.

Syntax

Formal_parameter_list:



Example

```
PROGRAM show_formparm (input);

VAR
   test: boolean;

FUNCTION chek1 (x, y, z: real): Boolean;

BEGIN
   { Perform some type of validity check on x, y, z  }
   { and return appropriate value.                   }
END;

FUNCTION chek2 (x, y, z: real): Boolean;

BEGIN
```

```
   { Perform an alternate validity check on x, y, z  }
   { and return appropriate value.                   }
END;

PROCEDURE read_data (FUNCTION check (a, b, c: real): Boolean);

VAR p, q, r: real;

BEGIN
   { read and validate data }
   readln (p, q, r);
   IF check (p, q, r) THEN ...
END;

BEGIN {show_formparm}
   .
   IF test THEN read_data (chek1)
   ELSE read_data (chek2);
   .
END.

PROGRAM show_varparm(output);

VAR
   i,j : integer;

PROCEDURE fix(VAR a : integer; b : integer);

BEGIN
   a := b;  { i is passed by reference; it will return equal to 42.}
   b := 0;  { j is passed by value; this assignment will           }
            { not change the value of j in the main program.       }
END;

BEGIN { show_varparm }
   i:= 0;
   j:= 42;
   fix(i,j);
   IF i = j THEN writeln('They both = 42');
END.

PROGRAM show_conformantparm;

CONST
   First=1;
   Last=10;

TYPE
   inxtype=1..100;
   arr1=ARRAY[First..Last] of Integer;
   arr2=ARRAY[First..2*Last] of Integer;

VAR
   a1,a2,a3:arr1;
   b1,b2,b3:arr2;

PROCEDURE ADD_Array(
   VAR Result:ARRAY[L..U:inxtype] OF INTEGER;
      P1,P2:ARRAY[L1..U1:inxtype] OF INTEGER
   );

VAR
   inx:inxtype;

BEGIN { ADD_Array }
   IF (L=L1) AND (U=U1) THEN
   FOR inx:=L TO U DO
      Result[inx]:=P1[inx]+P2[inx]
   ELSE
      { handle the error }
END;  { Add_Array }

BEGIN { Show_ConformantParm }

   { Initialize values for a1,a2,b1,b2 }
```

```
        { ADD_Array can be used for arrays of type arr1 and arr2
          because they conform to each other.}

        ADD_Array(a3,a1,a2);
        ADD_Array(b3,b1,b2);

    END.  { Show_ConformantParm }
```

Conformance

A *conformable test* must be passed to pass an array as an actual *conformant array parameter*. Actual conformant array parameters must have a type conformable with the *conformant array form* corresponding to the parameter in the procedure declaration.

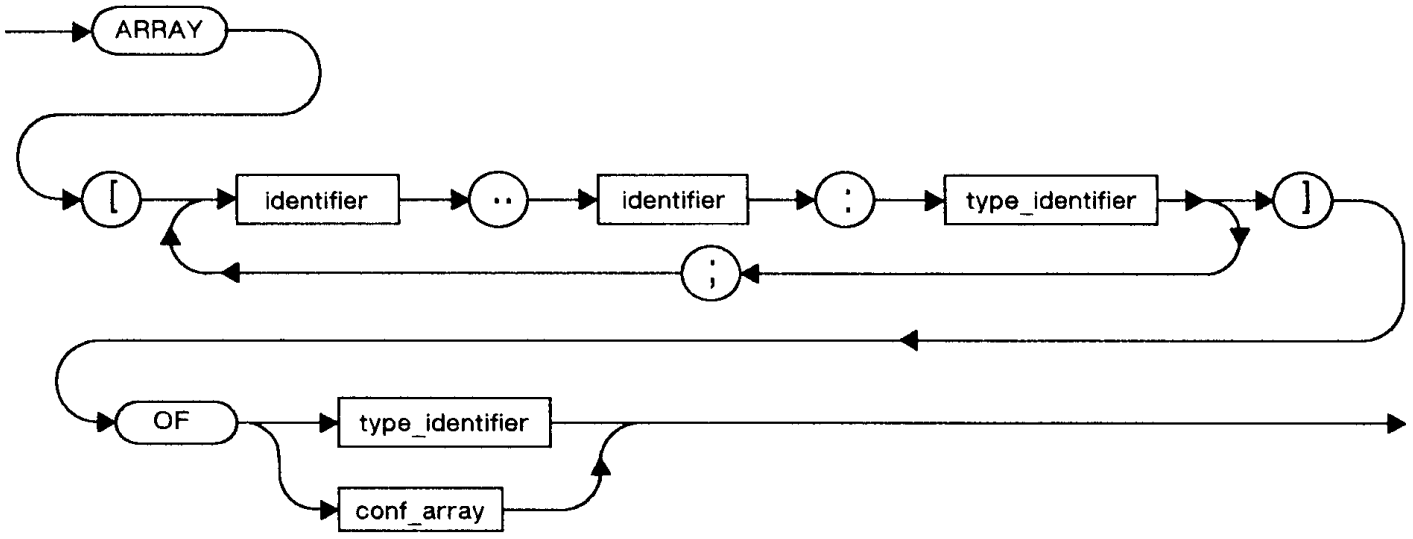
If T1 is an array type with a single index type, and T2 is the type of the index type specification of a conformant array form, then T1 is conformable with the conformant array form if all the following are true:

* The index type of T1 is type compatible with T2.
* You cannot index a value of T1 that does not lie within the bounds of that specified by T2.
* The component type of T1 is identical to the type identifier of the conformant array form, or, if the element type of the conformant array form is a conformant array form, is conformable with the element type conformant array form in the conformant array schema.
* Both T1 and the conformant array form are *packed* or both are *unpacked*.

Syntax

Conf_Array_Schema:

Conf_Array:



Example

```
TYPE
   inxtype = 0..20;

...

PROCEDURE Proc1 (
   P1: ARRAY[L1..H1:inxtype] OF ARRAY[L2..H2:inxtype] OF integer;
   P2: PACKED ARRAY[L3..H3:inxtype] OF integer;
   P3: ARRAY[L4..H4:inxtype] OF integer;
   P4: ARRAY[L5..H5:inxtype;L6..U6:inxtype] OF integer);

...

VAR
   V1: PACKED ARRAY[0..10] of integer;
   V2: ARRAY [3..5,1..10] OF integer;
   V3: ARRAY[1..50] OF integer;
```

V1 is conformable with P2, but not with P1, P3, and P4. V2 is conformable with P1 and P4, but not with P2 or P3. V3 is comformable with P3, but not with P1, P2, or P4.

Directives

A *directive* may replace a block in a procedure or function declaration. In HP Standard Pascal, the only directive is FORWARD. This directive makes it possible to postpone full declaration of a procedure or function. Additional directives may be provided by particular implementations. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for information about those directives. Note that the term FORWARD may appear as an identifier in source code and, at the same time, as a directive.

FORWARD Directive

The FORWARD directive permits the full declaration of a procedure or function to appear after a call to the procedure or function. For example, if procedures A and B are declared on the same level, you must use the FORWARD directive if A and B will call each other.

Example

```
PROCEDURE A; FORWARD;

PROCEDURE B;
BEGIN
   .
   A;    { calls A }
   .
END;

PROCEDURE A;  { full declaration of A }

BEGIN
   .
   B;    { calls B }
   .
END;
```

The body of the function or procedure must be fully declared elsewhere in the same block. Formal parameters, if any, and the function result type must appear with the FORWARD declaration. These formal parameters and the result type may be omitted when making the subsequent full declaration. However, if repeated, they all must be present and identical with the original formal parameters or result type.

The FORWARD directive may appear with a procedure or function at any level.

Example

```
FUNCTION exclusive_or (x,y: Boolean): Boolean;
```

```
FORWARD;
   .
   .
FUNCTION exclusive_or;        { Parameters not repeated. }

BEGIN
   exclusive_or:= (x AND NOT y) OR (NOT x AND y);
END;
```

Recursion

A *recursive* procedure or function is a procedure or function that calls itself. It is also legal for procedure A to call procedure B that in turn calls procedure A. This is *indirect recursion* and is often an instance when the FORWARD directive is useful. Note that when a routine is called recursively, new local variables are created for each invocation of the routine.

Example

```
FUNCTION factorial (n: integer): integer;
{ Calculates factorial recursively }

BEGIN
   IF n = 0 THEN
      factorial := 1
   ELSE
      factorial := n * factorial(n-1);
END;
```

Function Calls

A *function call* invokes the block of a standard or declared function and returns a value to the calling point of the program. Because an operator can perform some action on this value, a function result is an expression.

A *function call* consists of a function identifier and an optional list of actual parameters in parentheses. The actual parameters must match the formal parameters in *number, type,* and *order*. The function result has the type specified in the function heading. Actual value parameters are expressions that must be assignment compatible with the formal value parameters or, in the case of value conformant array parameters, conform with the conformant array schema.

Actual *reference parameters* are variables that must be type identical with the formal variable parameters or, in the case of variable conformant array parameters, conform with the conformant array schema. Components of a packed structure may not appear as actual variable parameters.

Actual *procedural* or *functional parameters* are the names of declared procedures or functions. Standard functions or procedures are not legal actual parameters.

The *parameter list*, if any, of an actual procedural or functional parameter, must be congruent with the parameter list of the formal procedural or functional parameter. For more information, see the section on Procedures in this chapter.

Functions may call themselves recursively. Refer to "Recursion" earlier in this chapter for more details.

If an actual functional or procedural parameter, upon invocation, accesses any entity non-locally, then the entity accessed is one that is accessible to the function or procedure when its identifier is passed. For example, suppose Procedure A uses the non-local variable x. If A is passed as a parameter to Function B, then it still has access to x, even if x is otherwise inaccessible in B.

If the *function result* is a structured type, then the function call may select a particular component as the result. This requires the use of an appropriate selector.

Example

```
PROGRAM show_function (input,output);

VAR
   n,
   coef,
   answer: integer;

FUNCTION fact (p: integer) : integer;

BEGIN
   IF p > 1 THEN
      fact := p * fact (p-1)
   ELSE fact := 1
END;

FUNCTION binomial_coef (n, r: integer) : integer;

BEGIN
   binomial_coef := fact (n) DIV (fact (r) * fact (n-r))
END;

BEGIN { show_function }
   read(n);
   FOR coef := 0 TO n DO
       writeln (binomial_coef (n, coef));
END.  { show_function }
```

Chapter 9 Standard Routines

HP Pascal supplies *predefined procedures* and *functions* that perform various commonly used operations. These are listed below, followed by a description of most in the subsequent pages of this chapter. Any procedure or function that is followed by an asterisk (*) is discussed in Chapter 10 .

Procedures:

append *	overprint *	setstrlen
assert	pack	strappend
associate *	page *	strdelete
close *	prompt *	strinsert
disassociate *	put *	strmove
dispose	read *	strread
get *	readdir *	strwrite
halt	readln *	unpack
mark	release	write *
movebyteswhile	reset *	writedir *
new	rewrite *	writeln *
open *	seek *	

Functions:

abs	lastpos *	sqr
arctan	linepos *	sqrt
baddress	ln	statement_number
binary	maxpos *	str
chr	octal	strlen
cmpbytes	odd	strmax
cos	ord	strltrim
eof *	position *	strpos
eoln *	pred	strrpt
exp	round	strrtrim
fnum *	scanuntil	succ
get_alignment	scanwhile	trunc
hex	sin	waddress

Procedures for Allocation and Deallocation

HP Pascal distinguishes two classes of *variables*. These are *static* and *dynamic*.

A *static variable* is explicitly declared in the declaration part of a block, and may then be referred to by name in the body. The compiler allocates storage for this variable when the block is activated. The system does not deallocate this space until the process closes the scope of the variable.

A *dynamic variable* is not declared and cannot be referred to by name. Instead, a declared pointer references this variable. The system allocates and deallocates storage for a dynamic variable during program execution as a result of calls to the standard procedures new and dispose. HP Pascal also supports the standard procedures mark and release. The area of memory reserved for dynamic variables is called the heap.

Dynamic variables permit the creation of temporary buffer areas in memory. Furthermore, since a pointer may be a component of a structured dynamic variable, it is possible to write programs with dynamic data structures such as linked lists or trees.

new

Usage

```
new(p )
new(p, t1,...,tn )
```

Parameters

p Any pointer variable.

t A case constant value representing tag values for the pointer variable *p*.

Description

The procedure new(*p*) allocates storage for a dynamic variable on the heap and assigns its identifying value to the pointer variable *p*.

If the dynamic variable is a record with variants, then the tag may be used to specify a case constant. This constant determines the amount of storage allocated. For nested variants, the values must be listed contiguously and in order of their declaration. The procedure call does not assign the specified tag values to the tag fields of the dynamic variable.

If new is called for a record with variants and no case constants are specified, the compiler determines storage by the size of the fixed part plus the size of the largest variant.

NOTE You cannot use an entire dynamic record variable allocated with one or more case constants as an actual parameter, or in an assignment statement.

Note that the pointer variable may be a component of a packed structure. Pointer dereferencing accesses the actual values stored in a dynamic variable on the heap.

Example

```
PROGRAM show_new (output);

TYPE
   marital_status = (single, engaged, married, widowed, divorced);
   year = 1900..2100;
   ptr  = ^person_info;
   person_info = RECORD
                    name: string[25];
                    birdate: year;
                    next_person: ptr;
                    CASE status: marital_status OF
                     married..divorced: (when: year;
                                         CASE has_kids: Boolean OF
                                          true: (how_many: 1..50);
                                          false: ();
                                        );
                     engaged: (date: year);
                     single : ();
                 END;

VAR
   p : ptr;

BEGIN                          { Various legal calls of new.           }
   .
   .
   new(p);                     { Allocates record of the largest size. }
```

```
        .
        .
        new(p,engaged);             { Allocates record with variant engaged.}
        .
        .
        new(p,married);             { Allocates record with variant married.}
        .
        .
        new(p,widowed,false);       { Allocates record with variants widowed
        .                             and false.}
        .
     END.
```

dispose

Usage

```
dispose(p)
dispose(p, t1,...,tn)
```

Parameters

p A pointer expression that cannot be NIL or undefined.

t A case constant value whose value matches the case constant value specified in new.

Description

This procedure indicates that the storage allocated for the given dynamic variable is no longer needed. It is an error if the argument to dispose is NIL or undefined. After dispose, the system has closed any files in the disposed storage and *p* is undefined.

If the case constant values are specified when calling new, it is an error if the identical constants do not appear as the parameters in the call to dispose. It is also an error if the pointer argument *p* references a dynamic variable to which another reference exists. This would be the case if it is a reference parameter, part of a reference parameter, or another pointer to it exists elsewhere.

Using dispose may be equivalent to executing an empty statement. For more details, see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, or the compiler options "HEAP_COMPACT" and "HEAP_DISPOSE" .

Example

```
PROGRAM show_dispose (output);

TYPE
   marital_status = (single, engaged, married, widowed, divorced);
   year = 1900..2100;
   ptr  = ^person_info;
   person_info = RECORD
                    name: string[25];
                    birdate: year;
                    next_person: ptr;
                    CASE status: marital_status OF
                       married..divorced: (when: year;
                                           CASE has_kids: boolean OF
                                            true: (how_many:1..50);
                                            false: ();
                                          );
                       engaged: (date: year);
                       single : ();
                 END;
```

```
VAR
   p : ptr;

BEGIN
   .
   .
   new(p);                    { Allocates largest variant.               }
   .
   .
   dispose(p);                { Deallocates record with largest variant. }
   .
   .
   new(p,engaged);            { Allocates record with variant engaged.   }
   .
   .
   dispose(p,engaged);        { Deallocates record with variant engaged. }
   .
   .
   new(p,married,false);      { Allocates record with variants married and false.}
   .
   dispose(p,married,true);   { Error, case constants don't match new.   }
   .
   .
END.
```

mark

Usage

mark*(p)*

Parameter

p A pointer variable.

Description

The procedure mark*(p)* marks the allocation state of the heap and sets the value of *p* to specify that state. In other words, mark saves the allocation state of the heap in *p*, which must not subsequently be altered by assignment. If altered, the corresponding release cannot be performed. mark is used with release.

Example

```
PROGRAM show_markrelease;

VAR
   w,x,y: ^integer;

BEGIN
   .
   mark(w);
   .
   release(w);  { Returns heap to state marked by w.      }
   .
   mark(x);
   .
   mark(y);
   .
   release(x);  { Returns heap to state marked by x. The  }
   .            { pointer y no longer marks a heap state. }
END.            { Release(y) is now an error.             }
```

release

Usage

release*(p)*

Parameter

p A pointer variable that previously appeared as a parameter in a call to mark, and should not have been previously passed to release or altered by assignment.

Description

The procedure release*(p)* returns the heap to its allocation state when *mark* was called with a parameter that has the value of *p*. This has the effect of deallocating any heap variables allocated since the program called mark. The system can then reallocate the released space. The system automatically closes any files in the released area.

It is an error if *p* was not passed as a parameter to mark, or if it was previously passed to *release* explicitly or implicitly. After release, *p* is undefined.

Example

```
PROGRAM show_markrelease;

VAR
   w,x,y: ^integer;

BEGIN
   .
   mark(w);
   .
   release(w);  { Returns heap to state marked by w.      }
   .
   mark(x);
   .
   mark(y);
   .
   release(x);  { Returns heap to state marked by x. The  }
   .            { pointer y no longer marks a heap state. }
END.            { Release(y) is now an error.             }
```

String Procedures

HP Pascal supports a number of *string procedures* that manipulate string expressions, variables, or literals. A string expression may consist of a string literal, a string variable, a string constant, a function result that is a string, or an expression formed with the concatenation operator.

Note that strings must be initialized just like any other variable. The string procedures are setstrlen, strappend, strdelete, strinsert, strmove, strread, and strwrite. These procedures are described in the following pages.

setstrlen

Usage

setstrlen*(s, e)*

Parameters

s A string variable.

e An integer expression. The value of *e* must not be greater than the maximum length of *s*.

Description

The procedure setstrlen*(s, e)* sets the current length of *s* to *e* without modifying the contents of *s*.

If the new length of *s* is greater than the previous length of *s*, the extra components become defined, but no value is given to them. No blank filling occurs. If the new length of *s* is less than the previous length of *s*, previously defined components beyond the new length become undefined.

Example

```
VAR
   alpha: string[80];

BEGIN
   .
   alpha:= 'abcdef';                  { strlen(alpha) = 6       }
   .
   setstrlen(alpha,2*strlen(alpha));  { Doubles current length  }
   .                                  { of alpha. Alpha[7]      }
   .                                  { through alpha[12] have  }
   .                                  { unpredictable values.   }
   .
   setstrlen(alpha,2)                 { Alpha[3] through        }
   .                                  { alpha[80] not undefined.}
END.
```

strappend

Usage

strappend*(s1, s2)*

Parameters

s1 A string variable.

s2 A string expression whose length must be less than the difference between the maximum and actual length of the string variable *s1*.

Description

The procedure strappend*(s1, s2)* appends string *s2* to *s1*. It is an error if the strlen of *s2* is greater than strmax*(s1)* -strlen*(s1)*. That is, it cannot exceed the number of characters left to fill in *s1*. The current length of *s1* is updated to strlen*(s1)* +strlen*(s2)*.

Example

```
VAR
   message: string[132];

BEGIN
   .
   message:= 'Now hear ';
   strappend(message,'this!');   { message is 'Now hear this!' }
   .
END.
```

strdelete

Usage

```
strdelete(s, p, n)
```

Parameters

s A string variable.

p An integer expression representing the starting index of the deletion.

n An integer expression representing the number of characters to be deleted.

Description

The procedure strdelete*(s, p, n)* deletes *n* characters from *s* starting at component *s [p]* , and the current length of *s* is updated to the length of *s-n*. It is an error if *n+p-1* is greater than the current length of *s*.

Example

```
VAR
   uncensored, censored: string[80];

BEGIN
   .
   uncensored:= 'Attack at 6 a.m.!';
   strdelete(uncensored,7,strlen(uncensored)-7);

   censored:= uncensored;  { censored is 'Attack!'. }
   .
   .
END.
```

strinsert

Usage

```
strinsert(s1, s2, p)
```

Parameters

s1 A string expression.

s2 A string variable.

p An integer or an integer expression representing the offset in *s2* where insertion begins.

Description

The procedure strinsert*(s1, s2, p)* inserts string *s1* into *s2* starting at *s2 [p]*. Initially, *s2* must be at least *p* -1 characters in length, or it is an error. The resulting string may not exceed strmax*(s2)*. The current length of *s2* is updated to strlen*(s1)* + strlen*(s2)*.

Example

```
VAR
   remark: string[80];

BEGIN
   .
   remark:= 'There is missing!';
   strinsert(' something',remark,9);{ remark is 'There is something missing! }
```

```
    .
END.
```

strmove

Usage

```
strmove(n, s1, p1, s2, p2)
```

Parameters

n An integer expression indicating the number of characters to be copied.

s1 A string expression or PAC variable.

p1 An integer expression indicating the index in *s1* from which copying starts.

s2 A string or PAC variable.

p2 An integer expression indicating the index in *s2* where copying starts.

Description

The procedure strmove*(n, s1, p1, s2, p2)* copies *n* characters from *s1*, starting at *s1[p1]*, to *s2*, starting at *s2[p2]*. The string length of *s2* is increased, if needed, to (*p2+n* -1) if (*p2+n* -1) > strlen*(s2)*. If *p2* equals strlen*(s2)* +1, strmove is equivalent to appending a subset of *s1* to *s2*. It is an error if *p2* > strlen*(s2)* +1. The value (*p1+n* -1) must not exceed strlen(*s1*).

The strmove procedure may be used to convert PAC's to strings and vice versa. It is also a way of manipulating subsets of PAC's.

NOTE The strmove procedure should not be used to move data into an uninitialized variable, regardless of type.

NOTE The strmove procedure is not appropriate for propagating characters within a string. Use the strrpt function or the fast_fill procedure instead.

Example

```
VAR
   pac: PACKED ARRAY[1..15] OF char;
     s: string[80];

BEGIN
   s:= '';
   pac:= 'Hewlett-Packard';
   strmove(15,pac,1,s,1);   { Converts a PAC to a string. }
END.
```

strread

Usage

```
strread(s, p, t, v)
strread(s, p, t, v1,...,vn)
```

Parameters

s A string expression.

p An integer expression.

t An integer or integer subrange variable.

v A simple, string, or PAC variable. Any number of *v* parameters may appear separated by commas.

Description

The procedure strread*(s, p, t, v)* reads a value from *s*, starting at *s [p]* , into the variable *v*. After the operation, the value of the variable appearing as the *t* parameter will be the index of *s* immediately after the index of the last component read into *v*.

S is treated as a single-line textfile. Strread*(s, p, t, v)* is analogous to read*(f, v)* when *f* is a textfile of one line. Like read, strread implicitly converts a sequence of characters from *s* into the types integer, real, longreal, Boolean, enumerated, PAC, or string.

It is an error if strread attempts to read beyond the current length of *s*.

The call:

```
strread (s,p,t,v1,...,vn);
```

is equivalent to:

```
strread (s,p,t,v1);
strread (s,t,t,v2);
.
.
strread (s,t,t,vn);
```

Example

```
VAR
   s: string[80];
   p,t: 1..80;
   m,n: integer;
BEGIN
   .
   s:= '   12  564   ';
   .
   p:= 1;
   strread(s,p,t,m);      { The value of m will be 12;  }
    .                     { t will be 6.                }
    .
   strread(s,t,t,n);      { The value of n will be 564; }
    .                     { t will be 11.               }
END.
```

strwrite

Usage

```
strwrite(s, p, t, e)
strwrite(s, p, t, e1,...,en)
```

Parameters

s A string variable.

p An integer expression.

t An integer or integer subrange variable.

e A simple or string expression, or a PAC variable. Any number of *e* parameters may appear separated by commas.

Description

The procedure strwrite*(s, p, t, e)* writes the value of *e* on *s* starting at *s [p]* . After the operation, the value of the variable appearing as the *t* parameter is the index of the component of *s*, immediately after the last component of *s* that strwrite has accessed.

S is treated as a single-line textfile. Strwrite*(s, p, t, e)* is analogous to write*(f, e)* when *f* is a one-line textfile. As with write, strwrite also permits you to format the value of *e* as it is written to *s* using the formatting conventions. The same default formatting values hold for strwrite.

Strwrite may write into the middle of a string without affecting the original length. It is an error if strwrite attempts to write beyond the maximum length of *s*, or if *p* is greater than strlen*(s)* + 1. The current length of *s* is updated if the current length is increased.

Example

```
VAR
   s: string[80];
   p,t: 1..80;
   f,g: integer;

BEGIN
   f:= 100;
   g:= 99;
   p:=1;
   s:='';                    { empty string              }
    .
   strwrite(s,p,t,f:3);      { S is now '100'; t is 4    }
   strwrite(s,t,t,' ',g:2);  { S is now '100 99'; t is 7. }
    .
END.
```

String Functions

String functions may be used to manipulate string expressions, variables, or literals. A string expression may consist of a string literal, a string variable, a string constant, a function result that is a string, or an expression formed with the concatenation operator.

Note that strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information. The string functions str, strlen, strltrim, strmax, strpos, strrpt, or strrtrim, are defined by HP Pascal and are described on subsequent pages.

str

Usage

```
str(s, p, e)
```

Arguments

s A string expression.

p An integer expression indicating the index of the starting character.

e An integer expression indicating the length of the substring.

Description

The function str*(s, p, e)* returns the portion of *s* which starts at *s [p]* and is of length *e*. The result is type string, and may be used as a string expression. It is an error if strlen*(s)* is less than *p* + (*e* -1).

Example

```
VAR
   i: integer;
   wish_list: string[132];
   granted: string[5];

BEGIN
   .
   i:= 13;
   wish_list:= 'wish1 wish2 wish3 wish4 wish5';
   granted:= str(wish_list,i,5);        { Selects the 3rd wish. }
                                        { Granted is 'wish3'.   }
END.
```

strlen

Usage

```
strlen(s)
```

Argument

s A string expression.

Description

The function strlen*(s)* returns the current length of the string or PAC expression *s*. If *s* is not initialized, strlen*(s)* is undefined.

Example

```
VAR
   ars, vita: string[132];
   b: boolean;

BEGIN
   .
   ars:= 'HELLO';
   vita:= 'TO YOU';
   IF strlen(ars) > strlen(vita) THEN
      b:= true
   ELSE
      b:=false;
       .
```

```
        writeln (strlen(ars):2,strlen(vita):2);
     END.
```

Output:

```
     5 6
```

strltrim

Usage

```
     strltrim(s)
```

Argument

s A string expression.

Description

The function strltrim*(s)* returns a string consisting of *s* trimmed of all leading blanks. The function strrtrim trims trailing blanks.

Example

```
     VAR
       s: string[80];
     BEGIN
        .
       s:= '          abc';
       s:=strltrim(s);        {s is now 'abc'}
        .                     {strlen(s) = 3 }
     END.
```

strmax

Usage

```
     strmax(s)
```

Argument

s A string variable.

Description

The function strmax*(s)* returns the maximum length of *s*. Strmax is useful for finding the maximum length of VAR string parameters whose maximum is not determined until run time.

Example

```
     VAR
        s: string[15];

     BEGIN
        s:= '     ABCDE     ';
        IF strlen(s) = strmax(s) THEN
           BEGIN
              s:= strltrim(s);
              s:= strrtrim(s);
           END;

        writeln (s,strmax(s):3);
        .
     END.
```

Output:

```
ABCDE 15
```

strpos

Usage

```
strpos(s1, s2)
```

Arguments

s1 A string expression.

s2 A string expression.

Description

The function strpos*(s1, s2)* returns the integer index of the position of the first occurrence of *s2* in *s1*. If *s2* is not found, zero is returned. If the length of *s2* is zero, the result is 1.

NOTE Some HP Pascal implementations have the order of the two parameters reversed. Also, your implementation may have a compiler option that reverses the order of parameters.

Example

```
CONST
   separator = ' ';

VAR
   i: integer;
   names: string[80];

BEGIN
   .
   names:= 'Jon Jill Ruth Marnie Bob Joan Wendy';
   i:= strpos (names,separator);       { i = 4              }
   IF i <> 0 THEN
      strdelete(names,1,i);            { deletes first name }
      .
   i:= (strpos(names,'Ron'));          { i = 0              }
END
```

strrpt

Usage

```
strrpt(s, n)
```

Arguments

s A string expression.

n An integer expression indicating the number of repetitions where *n* must be greater than or equal to zero.

Description

The function strrpt*(s, n)* returns a string composed of *s* repeated *n* times. If *n* is 0, a zero-length string is returned.

Example

```
CONST
   one = '1';

VAR
   b_num: string[12];

BEGIN
   .
   b_num:= strrpt(one,strmax(b_num));     { b_num is '111111111111' }
   b_num:= strrpt ('a',10);               { b_num is 'aaaaaaaaaa' }
   .
END.
```

strrtrim

Usage

strrtrim*(s)*

Argument

s A string expression.

Description

The function strrtrim*(s)* returns a string consisting of *s* trimmed of trailing blanks. Leading blanks are stripped by the function strltrim.

Example

```
VAR
   s: string[80];

BEGIN
   .
   s:= 'abc          ';
   .
   s:= strrtrim(s);        { s is now 'abc' }
   .                       { strlen(s) = 3  }
END.
```

Transfer Procedures

The *transfer procedures* supported by HP Pascal are *pack* and *unpack*. A description of these procedures follows.

pack

Usage

pack*(a, i, z)*

Parameters

a Any ARRAY [m..n] of t.

i An expression that is type compatible with the index of the non-packed array.

z Any PACKED ARRAY [u..v] of t.

Description

The standard procedure pack transfers data from unpacked arrays to packed arrays. For example, assuming that *a* is an ARRAY[m..n] OF t and *z* is a

PACKED ARRAY[u..v] of t; the procedure pack*(a, i, z)* assigns components of the unpacked array *a*, starting at component *i*, to each component of the packed array *z*.

Because all the components of *z* are assigned a value, the normalized value of *i* must be less than or equal to the difference between the lengths of *a* and *z* + 1; for example, *i* -m+1 <= (n-m) - (v-u) + 1. Otherwise, it is an error when pack attempts to access a nonexistent component of *a*.

The component types of arrays *a* and *z* must be type identical. The index types of *a* and *z*, however, may be incompatible.

The call pack*(a, i, z)* is equivalent to:

```
BEGIN
  k:= i;
  FOR j:= u TO v DO
    BEGIN
      z[j]:= a[k];
      IF j <> v THEN k:= succ(k);
    END;
END;
```

where k and j are variables that are type compatible with the index type of *a* and the index type of *z*, respectively.

Example

```
PROGRAM show_pack (input,output);
TYPE
   clothes = (hat, glove, shirt, tie, sock);

VAR
   dis : ARRAY [1..10] OF clothes;
   box : PACKED ARRAY [1..5] of clothes;
   index: integer;
   .
   .

BEGIN
   .
   .
   index:= 1;
   pack(dis,index,box);   { After pack executes, box contains     }
   .                      { the first 5 components of dis.        }
   .
   index:= 8;
   pack(dis,index,box);   { An error results when pack attempts   }
   .                      { to access nonexistent 11th component  }
   .                      { of dis.                               }
END.
```

unpack

Usage

```
unpack(z, a, i)
```

Parameters

z Any PACKED ARRAY [u..v] of t.

a Any ARRAY [m..n] of t.

i An expression that is type compatible with the index of the non-packed array.

Description

This procedure transfers data from a packed array to an unpacked array. For example, assuming that a is an ARRAY[m..n] OF t and *z* is a PACKED ARRAY [u..v] OF t; the procedure unpack*(z,a,i)* successively assigns the components of the packed array **z**, starting at component u, to the components of the unpacked array *a*, starting at *a [i]*.

All the components of *z* are assigned. Also, the normalized value of *i* must be less than or equal to the difference between the lengths of *a* and *z* + 1; for example, *i* -m+1 <= (n-m) - (v-u) + 1. Otherwise, it is an error when unpack attempts to index *a* beyond its upper bound.

The index types of *a* and *z* need not be compatible. The components of the two arrays, however, must be type identical.

The call unpack*(z,a,i)* is equivalent to:

```
BEGIN
   k:= i;
   FOR j:= u TO v DO
      BEGIN
         a[k]:= z[j];
         IF j <> v THEN k:= succ(k);
      END;
  END;
```

where k and j are variables that are type compatible with the indices of *a* and *z* respectively.

Example

```
PROGRAM show_unpack (input,output);

TYPE
   suit_types = (casual, business, leisure, birthday);

VAR
   suit : PACKED ARRAY [1..5] OF suit_types;
   kase : ARRAY [1..10] OF suit_types;
   i : integer;
   .
   .
BEGIN
   .
   .
   i := 1;
   unpack(suit,kase,i);  { After execution, the first 5      }
   .                     { components of kase contain the    }
   .                     { value of suit.                    }
   .
   i := 7
   unpack(suit,kase,i);  { An error results because unpack   }
   .                     { attempts to assign a component of }
   .                     { suit to a component of kase which }
   .                     { is out of range.                  }
END.
```

Program Control Procedures

The only *program control procedures* supported by HP Pascal are halt and assert. The details of these procedures are given below.

halt

Usage

```
halt(n)
halt
```

Parameter

n An integer expression that may be omitted.

Description

This procedure terminates the execution of the program. What this means and what is done with the optional integer expression is implementation defined. For more information, see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

Example

```
halt
halt(int_exp)
```

assert

The predefined procedure *assert* allows your program to test assumptions, specify invariant conditions, and check data structure integrity.

Usage

```
assert (b, i  [, p ])
```

Parameters

b A Boolean expression that *assert* evaluates. If its value is *true,* the program executes the statement following the call to *assert*. If its value is *false,* the program's action depends upon whether *p* is specified and whether the ASSERT_HALT compiler option is OFF or ON (see Figure 11-1).

If the compiler can determine that *b* is a constant expression whose value is *true*, then it does not generate code for the call to *assert*.

i An integer expression. If the value of *b* is *false* and *p* is specified, procedure *p* is called with *i* as the actual value parameter. If *b* is *false* and *p* is not specified, the system issues a run-time error message that includes the value of *i*.

A call to the predefined function *statement_number* is a useful integer expression for *i*. It returns the statement number (as shown on the compiler listing) for the statement from which it is called (in this case, the call to *assert*).

p The name of a procedure whose heading has the syntax

```
PROCEDURE p (parameter_name  : integer);
```

If the value of *b* is *false* and *p* is specified, the system executes the call *p(i)*.

The default for the ASSERT_HALT compiler option is OFF (see Chapter 12 for more information).

Example

```
PROCEDURE my_assert (value : integer);
BEGIN
   writeln('my_assert #', value);
END;
```

```
PROCEDURE x (p : ptrtype; n : integer);
BEGIN
   assert(p <> nil, 80101, my_assert);
   assert(n >= 0, 80102);
END;
```

MPE V Migration Routines

baddress

Usage

baddress*(v)*

Parameters

v A variable, procedure, or function.

Description

The function *baddress(v)* returns the byte address of *v* when *v* is a variable name, and the entry point when *v* is a procedure or function name. This variable may not be type file or a file type component of a structured variable. Also, *v* cannot be a component of a packed structure, except if it is a component of a PAC.

baddress is useful for calling certain intrinsics which require byte addresses for parameters.

baddress returns an integer in the range *minint..maxint*.

NOTE [REV BEG]

baddress does not work correctly with the $OPTIMIZE compiler option for addresses of variables. Use type coercion and addr[REV END] instead. Refer to the *HP Pascal/iX Programmer's Guide* or to the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information on optimizer assumptions.

Example

```
TYPE
  rec_type = RECORD
               f1: integer;
               f2: boolean;
               f3: char;
             END;

VAR
  n: integer;
  r: rec_type;
  p: ^rec_type;
  a: ARRAY [1..10] of 0..255;
  pac:  PACKED ARRAY [1..10] OF char;
  pab:  PACKED ARRAY [1..10] OF boolean;
```

Calls

```
baddress(n)
baddress(r)
baddress(r.f3)
baddress(p)
```

```
baddress(p^)
baddress(p^.f3)
baddress(a)
baddress(a[4])
baddress(pac)
baddress(pac[2])    { Legal since component type is char. }
baddress(pab)
baddress(pab[2])    { Error.                              }
```

cmpbytes

Usage

```
cmpbytes (s1, s2, l )
```

Parameters

s1 A PAC or string variable that contains a byte string to compare.

s2 A PAC or string variable that contains a byte string to compare.

l A shortint or bit16 expression that indicates the number of bytes to be compared.

Result

A shortint indicating the result of the comparison:

```
0 : s1  is less than s2.
1 : s1  is greater than s2.
2 : s1  is equal to s2.
```

Description

The function cmpbytes compares the *s1* and *s2* byte strings for *l* bytes. The result is a shortint value indicating that the *s1* byte string is less than, greater than, or equal to the *s2* byte string.

NOTE This feature requires the compiler option STANDARD_LEVEL 'EXT_MODCAL'.

Example

```
$STANDARD_LEVEL 'EXT_MODCAL'$
program asmb005 (output)
type
   pac20 = packed array[1..20] of char;
var
   pac,pac1 : pac20;
   i : integer;
   s : shortint;
   c,m : char;
   b : boolean;
   result : shortint;

begin

s := 4;
pac := 'abcd';
pac1 := 'abcd';
result := cmpbytes( pac,pac1,s );
writeln(result);  {2}
```

```
pac := 'aacd';
pac1 := 'abcd';
result := cmpbytes( pac,pac1,s );
writeln(result);  {0}

pac := 'abcd';
pac1 ;= 'aacd';
result := cmpbytes( pac,pac1,s );
writeln(result);  {1}

end.
```

movebyteswhile

Usage

```
movebyteswhile (s, t, a, n, u, p)
```

Parameter

s A PAC or string variable that contains the source string to be copied.

t A PAC or string variable to which the source is to be copied.

a An ordinal constant expression whose ordinal value is 0 or 1, indicating whether the copy is to continue while the characters are alphabetic (1).

n An ordinal constant expression whose ordinal value is 0 or 1, indicating whether the copy is to continue while the characters are numeric (1).

u An ordinal constant expression whose ordinal value is 0 or 1, indicating whether the copied characters remain the same (0), or whether all lowercase characters are upshifted (1).

p A shortint variable which will indicate the index in the source array where the test condition, alpha or numeric, failed.

Description

The procedure movebyteswhile moves a byte from the source array to the target array if the byte meets the test conditions set by *a* or *n*. Once the condition fails, the *p* of the byte is returned. If *u* is set, each alphabetic character moved to the target array is upshifted. Either or both of *a* and *n* must evaluate to 1. If neither evaluates to 1, then the results are unpredictable.

The length field of a target string variable is not updated.

NOTE This feature requires the compiler option STANDARD_LEVEL 'EXT_MODCAL'.

Example

```
$STANDARD_LEVEL 'EXT_MODCAL'$
program asmb005(output);
type
   pac20 = packed array[1..20] of char;
const
   apac = pac20[20 of ' '];
var
```

```
   pac,pac1 : pac20
   s : shortint;
   result : shortint;

begin

pac1 := apac;
pac := 'thisoisoaotest56789 ';
movebyteswhile( pac, pac1, true, true, true, s );
writeln (s);              {20}
writeln('"',pac1,'"');   {"THISOISOAOTEST56789 ")

pac1 := apac;
movebyteswhile( pac, pac1, #1, true, false, s );
writeln (s);              {20}
writeln('"',pac1,'"');    {"thisoisoaotest56789 "}

pac1 := apac;
movebyteswhile( pac, pac1, true, #0, false, s );
writeln (s);              {15}
writeln('"',pac1,'"');   {"thisoisoaotest      "}

end.
```

scanuntil

Usage

scanuntil *(s, t1, t2, p)*

Parameters

s A PAC or string variable that contains the source string to be scanned.

t1 An expression whose value is of any char type.

t2 An expression whose value is of any char type.

p A shortint variable which will indicate the position in the source byte string where *t1* or *t2* was found.

Result A boolean value.

true : indicates *t2* was found.
false : indicates *t1* was found.

Description

The function scanuntil scans the source byte string until either the *t1* or *t2* is found. The position at which the *t1* or *t2* was found is returned. The result is a Boolean value indicating whether *t2* or *t1* was found.

NOTE This feature requires the compiler option STANDARD_LEVEL 'EXT_MODCAL'.

Example

```
$STANDARD_LEVEL 'EXT_MODCAL'$
program asmb005(output);
type
   pac20 = packed array[1..20] of char;
var
```

```
   pac : pac20;
   s : shortint;
   c,m : char;
   b : boolean;

begin

pac := 'thisoisoaotest56789 ';
c := '6';
m := ' ';
b := scanuntil( pac, c, m, s );
writeln (s);    {16}
writeln (b);    {false}

b := scanuntil( pac, 'x', m, s );
writeln (s);    {20}
writeln (b);    {true}

b := scanuntil( pac, #101, ' ', s );
writeln (s);    {12}
writeln (b);    {false}

end.
```

scanwhile

Usage

scanwhile *(s, t1, t2, p)*

Parameters

s A PAC or string variable that contains the source string to be scanned.

t1 An expression whose value is of any char type.

t2 An expression whose value is of any char type.

p A shortint variable into which an index is returned which indicates at which position in the source array the *t2* was found or the *t1* was not found.

Result

A boolean value:

true : indicates *t2* was found.
false : indicates *t1* was not found.

Description

The function scanwhile scans the source byte string until a byte is found that does not match the *t1*. The position where the match failed is returned. The result is a boolean value indicating whether *t2* was found or *t1* was not found.

NOTE This feature requires the compiler option STANDARD_LEVEL 'EXT_MODCAL'.

Example

```
$STANDARD_LEVEL 'EXT_MODCAL'$
program asmb005(output);
```

```
type
   pac20 = packed array[1..20] of char;
var
   pac : pac20;
   s : shortint;
   c,m : char;
   b : boolean

begin

pac := 'aaaaaaaaabaaaaaaaaaa';
c := 'a';
m := 'c';
b := scanwhile( pac, c, m, s );
writeln (s);   {10}
writeln(b);    {false}

b := scanwhile( pac, 'a', m, s );
writeln (s);   {10}
writeln(b);    {false}

b := scanwhile( pac,#98 ,'a', s );
writeln (s);   {1}
writeln(b);    {true}

end.
```

waddress

Usage

```
waddress (i)
```

Parameters

i The name of a variable, procedure, or function.

Description

The function *waddress(i)* returns the byte address of *i* when *i* is a variable name, and the entry point when it is a procedure or function name. This variable cannot be type file or a file type component of a structured variable. Also, *i* cannot be a component of a packed structure as an argument, except when this component is an element of a PAC.

The *waddress* function is useful for calling copy text from *baddress*.

waddress returns an integer in the range *minint..maxint*.

NOTE [REV BEG]

waddress does not work correctly with the $OPTIMIZE compiler option for addresses of variables. Use type coercion[REV END] and addr instead. Refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information on optimizer assumption.

```
TYPE
  rec_type = RECORD
               f1: integer;
               f2: boolean;
             END;

VAR
  n: integer;
```

```
  r: rec_type;
  p: ^rec_type;
  a: ARRAY [1..10] OF integer;
  pac: PACKED ARRAY [1..10] OF char;
  pab: PACKED ARRAY [1..10] OF boolean;
PROCEDURE pro;
  BEGIN
  END;
FUNCTION f: integer;
  BEGIN
  END;
```

Calls

```
waddress(n)
waddress(r)
waddress(r.f2)
waddress(p)
waddress(p^)
waddress(p^.f2)
waddress(a)
waddress(a[4])
waddress(pac)
waddress(pac[3])  { Legal since component type is char.   }
waddress(pab)
waddress(pab[3])  { Error.                                }
waddress(pro)
waddress(f)
```

Arithmetic Functions

The eight standard *arithmetic functions* in HP Pascal are abs, arctan, cos, exp, ln, sin, sqr, and sqrt. Details about each of these functions are given in the following pages.

abs

Usage

abs*(x)*

Argument

x A numeric expression.

Description

The abs function computes the absolute value of its argument, which must be an expression with a numeric type. The type of the result is the same as the type of the numeric expression. Note that it may be an error to take the absolute value of minint.

Example

Input	Result
abs(-13)	13 { integer result }
abs(-7.11)	7.110000E+00
abs (true)	error { not a numeric type }

arctan

Usage

arctan*(x)*

Argument

x A numeric expression.

Description

The arctan function returns the principal value of the angle that has the tangent equal to the argument. The result is in radians within the range -pi/2..pi/2. This function returns a real for sub-integer, integer, or real arguments, and longreal for longreal or super-integer arguments. The value used for pi is implementation dependent.

Example

Input	Result
arctan(num_exp)	
arctan(2)	1.107149E+00
arctan(-4.002)	-1.32594E+00

cos

Usage

cos*(x)*

Argument

x A numeric expression.

Description

The cos function returns the cosine of the angle represented by its argument that is interpreted in radians. This function returns a real for sub-integer, integer, or real arguments, and longreal for longreal or super-integer arguments. The range of the returned value is -1.0 through +1.0.

Example

Input	Result
cos(x_rad)	
cos(1.62)	-4.91838E-02

exp

Usage

exp*(x)*

Argument

x A numeric expression.

Description

The exp real function raises **e** to the power of the argument. This function returns a real for sub-integer, integer, or real arguments, and longreal for longreal or super-integer arguments. The value used for Naperian e is implementation dependent.

Example

Input	Result
exp(3)	2.008554E+01
exp(8.8E-3)	1.008839E+00
exp(8.8L-3)	1.00883883382898L+00

ln

Usage

ln*(x)*

Argument

x Any positive numeric expression.

Description

The ln function returns the natural logarithm (base e) of the argument. This function returns a real for sub-integer, integer, or real arguments, and longreal for longreal or super-integer arguments. It is an error if *x* is 0 or less than 0. The value used for Naperian e is implementation dependent.

Example

Input	Result
ln(43)	3.761200E+00
ln(2.121)	7.518877E-01
ln(0)	{ error }

sin

Usage

sin*(x)*

Argument

x A numeric expression.

Description

The sin function returns the sine of the angle interpreted in radians represented by its argument. This function returns a real for sub-integer, integer, or real arguments, and longreal for longreal or super-integer arguments. Note that the argument can be any numeric value.

Example

Input	Result
sin(rad)	
sin(0.024)	2.399769E-02
sin(90)	8.93997E-01

sqr

Usage

sqr*(x)*

Argument

x Any numeric expression.

Description

The sqr function computes the square of its argument that must be an expression with a numeric type. The type of the result is the same as the base type of the numeric expression.

Example

Input	Result
sqr(3)	9
sqr(1.198E3)	1.435204E+06.
sqr(-5)	25
sqr(maxint)	{ error }

sqrt

Usage

sqrt*(x)*

Argument

x Any positive numeric expression.

Description

The sqrt function computes the square root of its argument, which must be an expression with a numeric type. It is an error if the argument is less than 0. This function returns a real for sub-integer, integer, or real arguments, and longreal for longreal or super-integer arguments.

Example

Input	Result
sqrt(64)	8.000000E+00
sqrt(13.5E12)	3.674235E+06
sqrt(0)	0.000000E+00
sqrt(-5)	{ error }

Predicate Functions

There are three predicate functions in HP Pascal. They are odd, eof, and eoln. The functions eof and eoln are described in Chapter 10 of this manual.

odd

Usage

odd*(x)*

Argument

x Any integer expression.

Description

This function returns true if the integer expression is odd, and false otherwise.

Example

Input	Result
odd(int_var)	depends on value of int_var
odd(ord(color))	depends on value of color
odd(2 + 4)	false
odd(-32767)	true
odd(32768)	false
odd(0)	false

Numeric Conversion Functions

binary, hex, and octal are the three *numeric conversion functions* supported in HP Pascal.

binary, hex, and octal return an integer value. Therefore, all bits must be specified if a negative result is desired. Alternatively, the positive representation may be negated.

A description of each of these functions follows.

binary

Usage

binary*(s)*

Argument

s Any string or PAC expression whose range is implementation dependent.

Description

The binary function converts a string or PAC expression that is interpreted as a *binary* value to an integer. Leading and trailing blanks are ignored in the argument. It is an error if any character is not a legal binary digit; for example, 0..1.

Example

Input	Result
binary(strng)	depends on the value of strng
binary('10011')	19
-binary('10011')	-19

NOTE If your particular implementation uses 32-bit 2's complement notation, the following example also works:

```
binary('11111111111111111111111111101101')   =  -19
```

---

hex

Usage

hex*(s)*

Argument

s Any string or PAC expression whose range is implementation dependent.

Description

The hex function converts a string or PAC expression, that is interpreted as a *hexadecimal* value to an integer. Leading and trailing blanks are ignored. It is an error if any character is not a legal hex digit; for example, 0..9, 'A'..'F', or 'a'..'f'.

Example

Input	Result
hex(strng)	depends on the value of strng
hex('FF')	255
-hex('FF')	-255

NOTE If a particular implementation uses 32-bit 2's complement notation, the following example also works:

```
hex('FFFFFF01')        =       -255
```

---

octal

Usage

octal*(s)*

Argument

s Any string or PAC expression whose range is implementation dependent.

Description

The octal function converts a string or PAC expression that is interpreted as an *octal* value to an integer. Leading and trailing blanks in the argument are ignored. It is an error if any other character is not a legal octal digit; for example, 0..7.

Example

Input	Result
octal(strng)	depends on the value of strng
octal('77')	63
-octal('77')	-63

NOTE If your particular implementation uses 32-bit 2's complement notation, the following example also works:

```
octal('37777777701')          -63
```

Transfer Functions

Round and trunc are the *transfer functions* found in HP Pascal. These functions are described on the next two pages.

round

Usage

round*(x)*

Argument

x Any real or longreal expression.

Description

The round function returns the argument rounded to the nearest integer. If *x* is positive or zero, then round (*x*) is equivalent to trunc(*x* + 0.5); otherwise, round (*x*) is equivalent to trunc(*x* - 0.5). It is an error if the result is greater than *maxint* or less than *minint*.

Example

Input	Result
round(3.1+2.4)	6
round(3.1)	3
round(-6.4)	-6
round(-4.6)	-5
round(1.5)	2

trunc

Usage

trunc*(x)*

Argument

x Any real or longreal expression.

Description

The trunc function returns the integer part of a real or longreal expression that is the integral part of its argument. The absolute value of the result is not greater than the absolute value of *x*. It is an error if the result is greater than *maxint* or less than *minint*.

Example

Input	Result
trunc(real_exp)	depends on the value of real_exp
trunc(5.61)	5
trunc(-3.38)	-3
trunc(18.999)	18

Ordinal Functions

The *ordinal functions* found in HP Pascal are chr, ord, pred, and succ. Each of these functions are discussed on the next few pages.

chr

Usage

chr*(x)*

Argument

x An integer expression in the range 0..255.

Description

The chr function converts an integer numeric value into an ASCII character by returning the character value, if any, whose ordinal number is equal to the value of its argument. Note that it is an error if the argument is not within the range 0..255.

Example

Input	Result
chr(x)	depends on the value of x
chr(63)	'?'
chr(82)	'R'
chr(13)	(carriage return)

ord

Usage

ord*(x)*

Argument

x An ordinal expression.

Description

The function ord*(x)* returns the integer representing the ordinal associated with the value of *x*. If *x* is an integer, *x* itself is returned. If *x* is type char, the result is an integer value between 0 and 255 determined by the ASCII order sequence. If *x* is any other ordinal type (such as a predefined or user-defined enumerated type), then the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative integers star ting at zero. For example, since the standard type Boolean is predefined as:

```
TYPE Boolean = (false,true)
```

The call ord (false) returns 0, and the call ord (true) returns 1.

For any character ch, the following is true:

```
chr (ord (ch)) = ch
```

It is an error if the result is greater than *maxint* or less than *minint*.

Example

Input	Result
ord(ord_exp)	depends on the value of ord_exp
ord('a')	97
ord('A')	65
ord(-1)	-1
ord(yellow)	2 {TYPE color=(red,blue,yellow)}
ord(red)	0

NOTE Taking the ORD of short pointer type expressions is permitted at the Standard_Level EXT_MODCAL.

---

pred

Usage

pred*(n)*

Argument

x Any ordinal expression.

Description

The pred function returns the value whose ordinal number is one less than the ordinal number of the argument. The type of the result is identical to the type of the argument. pred(*x*) must exist.

Example

Input	Result
pred(ord_var)	depends on the value of ord_var
pred(1)	0
pred(-5)	-6
pred('B')	'A'
pred(true)	false
pred(false)	{error}

succ

succ*(x)*

Argument

x Any ordinal expression.

Description

The succ function returns the value whose ordinal number is one greater than the ordinal number of the argument. The type of the result is identical with the type of the argument. It is an error if succ*(x)* does not exist.

Example

Input	Result
succ(ord('b'))	99
succ(1)	2
succ(-5)	-4
succ('a')	'b'
succ(false)	true
succ(true)	{ error }

Chapter 10 Input and Output

Files are the means by which a program receives input and produces output. A file is a sequence of components of the same type. This may be any type except a file type or a structured type with a file type component.

Logical files are files declared in a Pascal program. *Physical files* are files that exist independently of a program and are controlled by the operating system. Logical and physical files are associated so that a program manipulates data objects external to itself.

The components of a file are indexed starting at component 1. Each file has a current component and a buffer variable whose contents, if defined, are accessible using a file buffer (^) selector. The standard procedure read(*f,x*) copies the contents of the current component into *x* and advances the current position to the next component. The procedure write(*f,x*) copies *x* into the current component and, like read, advances the current position.

The standard procedures reset, rewrite, append, or open are used to open a file for input or output. Reset opens a file in the *input* state so that writing is prohibited; rewrite and append open a file in the *output* state so that reading is prohibited; and *open* opens a file in the *direct* state so that both reading and writing are legal.

All files are automatically closed on exit from the block in which they are declared whether by a normal exit or a nonlocal GOTO or escape. Files allocated on the heap are automatically closed when the file or structure containing the file is disposed, or the area in which the file resides is released. All files are closed at the end of the program.

Files opened with reset, rewrite, or append are *sequential files*. In sequential files, the current position advances only one component at a time. Files opened with open are direct access files. The current position may be relocated anywhere in the file using the procedure seek. Direct access files have a maximum number of components determinable by the standard function maxpos. The maximum number of components of a sequential file, on the other hand, is not determinable with an HP Pascal function.

Textfiles are special predefined sequential files with char type components. End-of-line markers are used to substructure textfiles into lines. The standard procedure writeln creates these markers. The standard files *input and output* are textfiles. Textfiles cannot be opened for direct access.

Table 10-1 lists each HP Pascal file procedure or function together with a brief description of its action. The third column of the table indicates the permissible categories of files that a procedure or function may reference.

Table 10-1. File Procedures and Functions

Procedure or Function	Action	Permissible Files
append	Opens file in output state. Current position is after last component and eof is true.	any
associate	Associates a logical file with an open physical file.	any
close	Closes a file.	any
disassociate	Disassociates a logical file from it's associated open physical file.	any
eof	Returns true if file opened in output state, if no component exists for sequential input, or if current position in direct access file is greater than lastpos.	any
eoln	Returns true if the current position of a text file is at a line marker.	input textfiles
get	Allows assignment of current component to buffer and, in some cases, advances current position.	input or direct files
lastpos	Returns index of highest written component of direct access file.	direct access files
linepos	Returns number of characters read from or written to a textfile since the last line marker.	textfiles
maxpos	Returns maxint or the maximum component possible to read or write. Check implementation.	direct access files
open	Opens file in direct access state. Current position is 1 and eof is false. Eof is true if file is empty.	any file except a textfile
overprint	A form of write which causes the next line of a textfile to print over the current line.	output textfiles
page	Causes skip to top of new page when a textfile is printed.	output textfiles

Table 10-1. File Procedures and Functions (cont.)

position	Returns integer indicating the current component of a non-text file.	any file except a textfile

Procedure or Function	Action	Permissible Files
prompt	A form of write which assures textfile buffers have been written to the device. No line marker is written.	output textfiles
put	Assigns the value of the buffer variable to the current component and advances the current position.	output or direct access files
read	Copies current component into specified variable parameter and advances current position.	input or direct access files
readdir	Moves current position of a direct access file to designated component and then performs read.	direct access files
readln	Performs read on textfile and then skips to next line.	input textfiles
reset	Opens file in input state. Current position is 1.	any
rewrite	Opens file in output state. Current position is 1 and eof is true. Old components discarded.	any
seek	Places current position of direct access file at specified component number.	direct access files
write	Assigns parameter value to current file component and advances current position.	output or direct access files
writedir	Advances current position in direct access file to designated component and performs a write.	direct access files
writeln	Assigns parameter value to current textfile component, appends a line marker and advances current position.	output textfiles

append

Usage

```
append(f)
append(f, s)
append(f, s, t)
```

Parameters

f A variable of type file. The parameter *f* may not be omitted.

s The name of a physical file associated with f. This can be a string or PAC expression whose range is implementation defined.

t A string or PAC expression whose value is implementation dependent. Refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information. This parameter specifies carriage control and file access.

Description

The procedure append*(f)* opens file *f* in the output state, and places the current position immediately after the last component. All previous contents of *f* remain unchanged. The eof*(f)* function returns true, and the file buffer *f^* is undefined. Data may now be written on *f*.

If *f* is already open, append closes and then reopens it. If a file name is specified, the system closes any physical file previously associated with *f*.

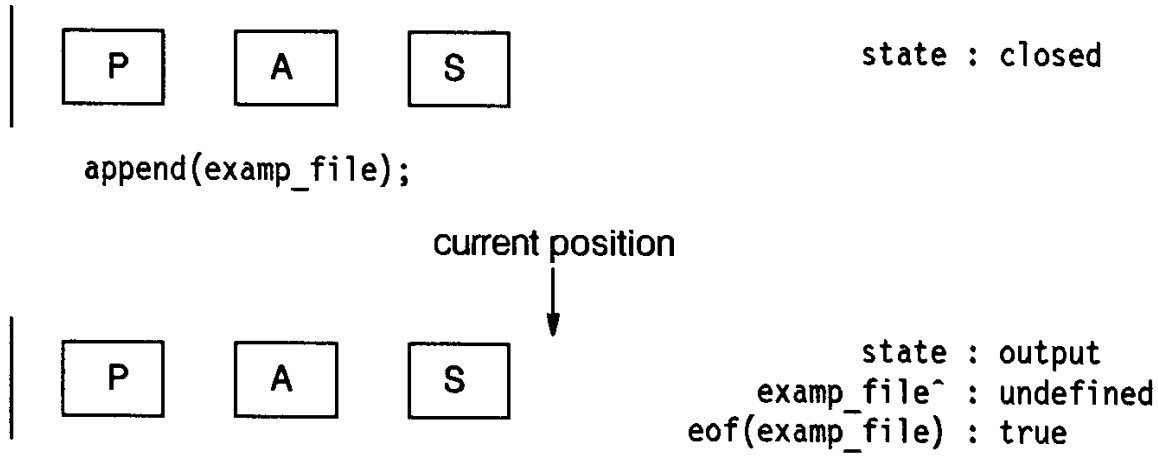
When *f* does not appear as a program parameter and *s* is not specified, the system maintains any previous association of a physical file with *f*. If there is no such association, it opens a temporary nameless file. This file cannot be saved. It becomes inaccessible after the process terminates or the physical-to-logical file association changes. For more information, see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

Example

```
append(file_var)
append(file_var,phy_file_spec)
append(file_var,phy_file_spec,opt_str)
append(fvar,'SHORTFIL')
```

Illustration

Suppose examp_file is a closed logical file of *char* containing three components. In order to open it and write additional material without disturbing its contents, *append* is called.



associate

Usage

associate*(f, num, option_str)*

Parameters

f A variable of type file.

num The system-provided file number of a previously opened file.

option_str Must be one of the following:

READ	associate to sequential access file with read access.
WRITE	associate to sequential access file with write access.
READ, DIRECT	associate to direct access file with read access.
WRITE, DIRECT	associate to direct access file with write access.
READ, WRITE, DIRECT	associate to direct access file with read/write access.
DIRECT	same as READ, WRITE, DIRECT.
NOREWIND	associate to a file without changing the current file position.

Description

The procedure associate*(f,num,option_str)* allows the opened file *num* to be used with Pascal input/output routines through *f*. The file must already be open as the result of a direct call to an operating system routine or as the result of a call to a non-Pascal procedure. The file cannot be opened as a result of a Pascal append, associate, open, reset, or rewrite. Therefore, the Pascal function fnum cannot be used to determine the file number of a file opened by Pascal. The file must also be open.

One of the above-mentioned combinations must appear in *option_str*. It is also an error to specify read or write access if the physical file is not opened for read or write access, respectively.

Other options legal for opening a file, such as those in the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, are ignored.

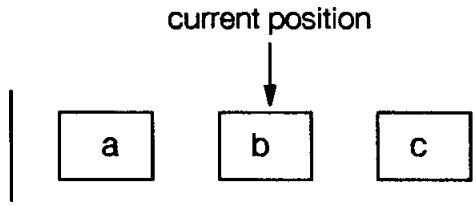
Associate places the current file position at the first component of the file unless NOREWIND is specified. The contents of *f*, if any, are undisturbed, and *f* is undefined. If the *option_str* parameter specifies WRITE, then eof(*f*) returns true, even though the actual end of file remains at the end of any previously existing data in the file. If the *option_str* parameter specifies read access for a sequential file or read or write access for a direct access file, eof(*f*) returns false after the call to associate. If the file is empty and is associated to read access, a subsequent read causes an error.

Example

```
associate(file_var,file_number,option_str)
```

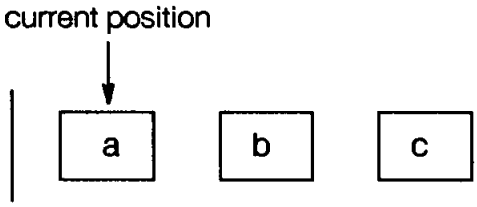
Illustration

Suppose examp_file is an opened logical file of char with three components. To read sequentially from examp_file, we call associate:



state : opened

```
associate(examp_file,file_num,'READ');
```




```
            state : output
examp_file^ : undefined
eof(examp_file) : false
```

close

Usage

close*(f)*
close*(f, t)*

Parameters

f A variable of type file. *f* may not be omitted.

t Options string that may be a string or PAC expression whose value is implementation dependent. Refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information.

Description

The procedure close(*f*) closes the file *f* so that it is no longer accessible. After being closed, any references to the file *f*, except through one of the file-open routines, results in an error, and *f* is not associated with any physical file.

When closing a direct access file, the last component of the file is the highest-indexed component ever written to the file (lastpos(*f*)). The value of maxpos for the file, however, remains unchanged. Once a file is closed, it may be reopened.

The options string specifies the disposition of any physical file associated with the file. The value is implementation defined. The compiler ignores leading and trailing blanks and considers upper and lower case equivalent. If no options string is supplied, the file retains its previous (original) status.

Example

```
close(fil_var)
close(fil_var,opt_str)
```

disassociate

Usage

disassociate*(f)*

Parameter

f A variable of type file.

Description

This procedure removes the logical-physical file association that was previously created with the associate procedure. Consequently, the file *f* is no longer available to Pascal input and output routines.

Normally a file is closed upon exit from the block in which it is declared. A file that has been disassociated will not be closed upon exit, and must be explicitly closed with a direct call to the operating system routines.

The disassociate procedure is useful when a file is passed to a Pascal routine and must remain open when control returns to the routine that passed the procedure to Pascal.

Example

```
disassociate (file_var)
```

eof

Usage

```
eof(f)
eof
```

Parameter

f A variable of type file that must be open. If *f* is omitted, the system uses the standard file *input*.

Description

This Boolean function returns true if the end of a file is reached. If the file *f* is open, the Boolean function eof(*f*) returns *true* when *f* is in the output state, when *f* is in the direct access state, and its current position is greater than the highest-indexed component ever written to *f*, or when no component remains for sequential input. Otherwise, eof(*f*) returns false. If false, the next component is placed in the buffer variable. If *f* is omitted, the system uses the standard file *input*.

When reading non-character values, such as integers or reals, from a textfile, eof may remain false even if no other value of that type exists in the file. This can occur if the remaining components are blanks; for example, eoln is still false.

Example

```
eof
eof(file_var)
```

eoln

Usage

```
eoln(f)
eoln
```

Parameter

f A variable of type TEXT opened in the input state. If f is omitted, the system uses the standard file *input*.

Description

This Boolean function returns true when the end of a line is reached in a textfile. This happens when the current position of textfile *f* is at an end-of-line marker. The function references the buffer variable *f* ^, possibly causing an input operation to occur. For example, after readln, a call to eoln places the first character of the new line in the buffer variable. If *f* is omitted, the system uses the standard file *input*.

Example

```
eoln
eoln(text_file)
```

get

Usage

```
get(f)
get
```

Parameter

f A variable of type file opened in input or direct access state. If *f* is omitted, the system uses the standard file *input*.

Description

The procedure get(*f*) advances the current file position and causes a subsequent reference to the buffer variable *f^* to actually load the buffer with the current component. This definition of get is known as the *deferred get.*

It is an error if *f* is in the output state or if eof(*f*) is true prior to the call to get.

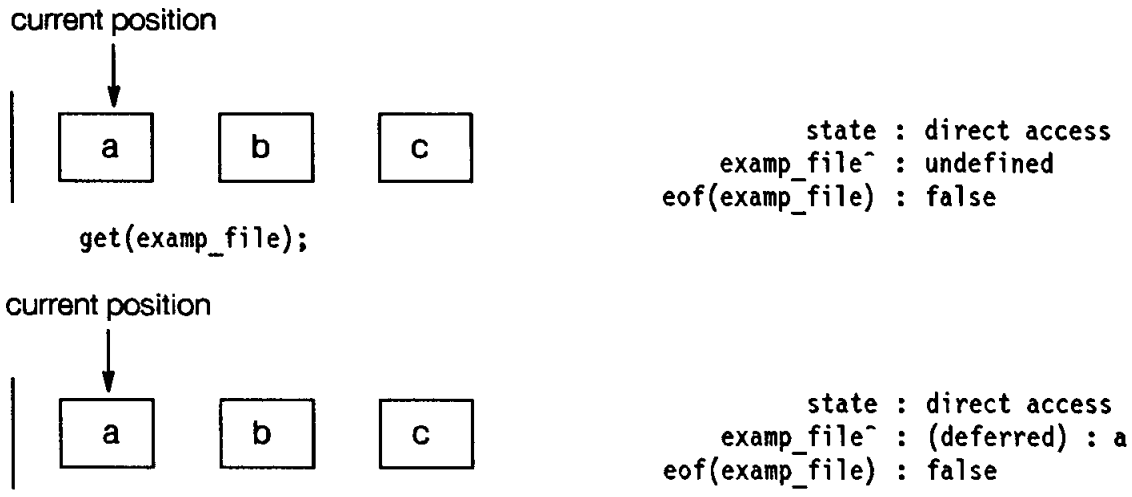
If a file is opened with open, a get must be performed to load the buffer variable with valid data. However, if a file is opened with reset, the buffer variable contains valid data and a get should not be performed until the second component is accessed. If get is called after read, one file component is skipped because read concludes with a get operation.

Example

```
get(file_var)
```

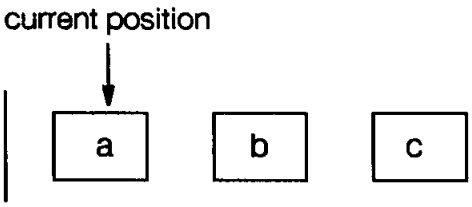
Illustration

Suppose examp_file is a logical file of char with three components which has just been opened in the direct state. The current position is the first component and examp_file^ is undefined. To inspect the first component, *get* is called.



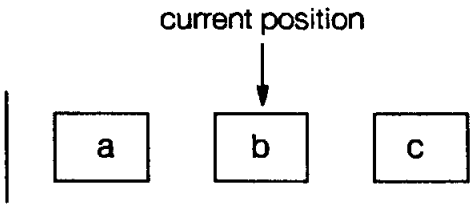
The current position is unchanged. Now, however, a reference to examp_file^ loads the first component into the buffer. We assign the buffer to a variable.

```
char_var := examp_file^
```




```
                state : direct access
    examp_file^ : a
eof(examp_file) : false
```

```
get(examp_file);
```




```
                state : direct access
    examp_file^ : (deferred) : b
eof(examp_file) : false
```

lastpos

Usage

lastpos*(f)*

Parameter

f A variable of type file opened in the direct access state. *f* must be specified.

Description

The function lastpos(*f*) returns the integer index of the last component of *f* that has been accessed while the program has been running, or in the life of the file. It is an error if *f* is not opened as a direct access file.

Example

```
i:=lastpos(file_var)     { File_var is the name of a file type variable }
```

linepos

Usage

linepos*(f)*

Parameter

f A textfile variable that must be opened. *f* may not be omitted. The program must specify the standard files *input* and *output* by name.

Description

The function linepos(*f*) returns the integer number of characters read from or written to the textfile *f* since the last end-of-line marker. This does not include the character in the buffer variable *f* ^. The result is zero after reading a line marker, or immediately after a call to readln, writeln, prompt, or overprint.

Example

```
i:=linepos(text_file)
```

maxpos

Usage

maxpos*(f)*

Parameter

f A file variable that must be opened in the direct access state where *f* may not be omitted.

Description

The function maxpos(*f)* returns the integer index of the last component of *f* that the program could possibly access. An error occurs if *f* is not opened as a direct access file. Note that the value returned is implementation defined.

On implementations that allow direct access files to be extended, maxpos returns the value of maxint or the maximum possible number.

Example

```
i:=maxpos(file_var)     { File_var is the name of a file type variable }
```

open

Usage

```
open(f)
open(f, s)
open(f, s, t)
```

Parameters

f A file variable that is not a textfile.

s The name of a physical file that the system associates with *f*.

t A string or PAC expression whose value is implementation dependent. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide,* depending on your implementation, for more details.

Description

The procedure open(*f)* opens *f* in the direct state and places the current position at the beginning of the file. The function eof returns false, unless the file is empty. The buffer variable *f^* is undefined.

After a call to open, *f* is said to be a direct access file. Data may be read or written using the procedures read, write, readdir, writedir, get or put. The procedure seek and the functions lastpos and maxpos are also legal. eof*(f)* becomes true when the current position is greater than the highest-indexed component ever written to *f*.

Direct access files have a maximum number of components. The function maxpos returns this number. The lastpos function returns the index of the highest-written component of a direct access file.

A textfile cannot be opened for direct access since its format is incompatible with direct access operations.

When the physical file specifier parameter is specified, the system closes any physical file previously associated with *f*.

When *f* does not appear as a program parameter and *s* is not specified, the system maintains any previous association of a physical file with *f*. If there is no such association, it opens a temporary, nameless file. This file cannot be saved. It becomes inaccessible after the process terminates or the physical-to-logical file association changes. For more information, see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

Example

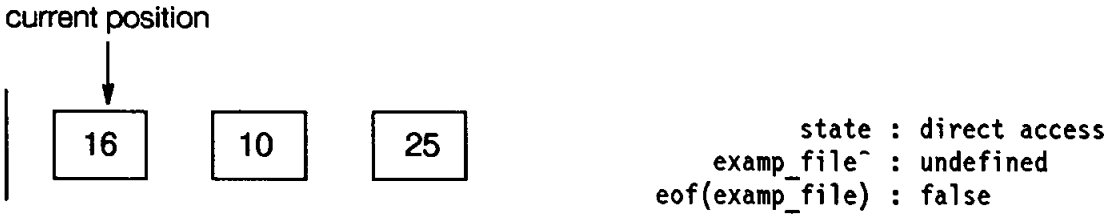
```
open(file_var)
open(file_var,phys_file_string)
open(file_var,phys_file_string,opt_str)
open(file_var,'TESTFILE')
```

Illustration

Suppose examp_file is a file of *integer* with three components. To

perform both input and output, we call open:

```
open (examp_file) ;
```



overprint

Usage

```
overprint(f)
overprint(f, e)
overprint(f, e1, ..., en)
overprint
overprint(e)
overprint(e1, ..., en)
```

Parameters

f A textfile variable that must be opened. If *f* is omitted, the system uses the standard file *output*.

e An expression of simple, string, or PAC type, or a string literal. The system writes the value of *e* on *f* according to the formatting conventions described for the procedure *write*.

Description

The procedure overprint has the same function as writeln, except that it does not terminate the line with a line feed. This causes the next write or overprint to overlay the line written by the original overprint. Several successive overprints all write to the same line, and printing advances to the next line after the first writeln.

NOTE Some printers do not support the overprint procedure. Refer to the manual for your particular printer.

After the execution of overprint*(f)*, the buffer variable *f* ^ is undefined and eoln*(f)* is false. The expression parameter, *e*, behaves exactly like the equivalent parameter for the procedure write.

If the output device is not a printer, overprint will be ignored.

Examples

```
overprint(file_var)
overprint(file_var,exp)
overprint(file_var,exp1,...,expn)
overprint(exp)
overprint(exp1,...,expn)
overprint
```

or

```
writeln('def');
overprint('___');
```

def

page

Usage

```
page(f)
page
```

Parameter

f A textfile variable that must be open. If *f* is omitted, the system uses the standard file *output*.

Description

The procedure page*(f)* writes a special character to the text file *f*, which causes the printer to skip to the top of the form when *f* is printed. The current position in *f* advances, and the buffer variable *f^* is undefined.

Example

```
page(text_file)
page
```

position

Usage

```
position(f)
```

Parameter

f A file variable that must not be a textfile.

Description

The function position*(f)* returns the integer index of the current component of *f*, starting from 1. Input or output operations references this component. The parameter *f* must not be a textfile.

Example

```
i:=position(file_var)
```

prompt

Usage

```
prompt(f)
prompt(f, e)
prompt(f, e1, ..., en)
prompt
prompt(e)
prompt(e1, ..., en)
```

Parameters

f A textfile variable. The system uses the standard file *output* if *f* is omitted.

e The expression of any simple, string, or PAC type or string literal.

Description

The procedure prompt*(f)* causes the system to write any buffers associated with textfile f to the device. prompt does not write a line marker on *f*. The current position is not advanced, and the buffer variable *f^* becomes undefined.

prompt is normally used when directing output to a terminal. prompt causes the cursor to remain on the same line after output to the screen is complete. The user may then respond with input on the same line.

The expression parameter, *e*, behaves exactly like the equivalent parameters in the procedure write.

Example

```
prompt(file_var)
prompt(file_var,exp)
prompt(file_var,exp1,...,expn)
prompt(exp)
prompt(exp1,...,expn)
prompt
```

put

Usage

```
put(f)
put
```

Parameter

f A file variable opened in the output or direct access state. The system uses the standard file output if *f* is omitted.

Description

The procedure put*(f)* assigns the value of the buffer variable *f^* to the current component and advances the current position. Following the call, *f* ^ is undefined.

It is an error if *f* is open in the input state.

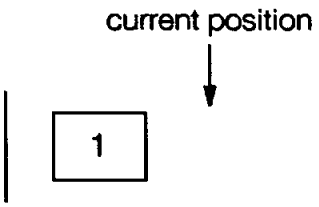
Example

```
put(file_var)
```

Illustration

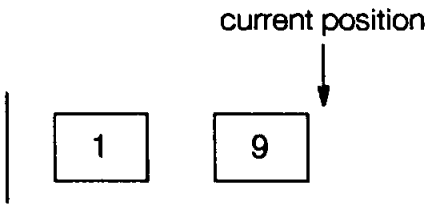
Suppose examp_file is a file of integer with a single component opened in the output state by append. Furthermore, 9 has been assigned to the buffer variable examp_file^. To place this value in the second component, put is called.

```
append ( examp_file ) ;  { for open }
examp_file^ := 9
```




```
          state : direct
    examp_file^ : 9
eof(examp_file) : true
```

```
put(examp_file);
```




```
          state : output
    examp_file^ : undefined
eof(examp_file) : true
```

read

Usage

```
read(f,v)
read(f, v1, ..., vn)
read(v)
read(v1, ..., vn)
```

Parameters

f A file variable opened in the input or direct access state. If *f* is omitted, the system uses the standard file *input*.

v The name of a variable or component of a structure whose type is not FILE and does not contain a component of type FILE.

Description

The procedure read*(f, v)* assigns the value of the current component of *f* to the variable *v*, according to the rules below, advances the current position, and causes any subsequent reference to the buffer variable *f* ^ to actually load the buffer with the current component.

If the file is a textfile, the read variables can be simple, string, or PAC variable. If the file is not a textfile, its components must be assignment compatible with the variable.

The following statement:

```
read(f,v)
```

is equivalent to accessing the file variable and establishing a reference to that file variable for the remaining execution of the statement (denoted by ff) and then calling get on ff.

```
v := ff^
get(ff);
```

For example, the call

```
read(f,v1,...,vn);
```

establishes a reference, ff, to the file variable, *f*. It is equivalent to:

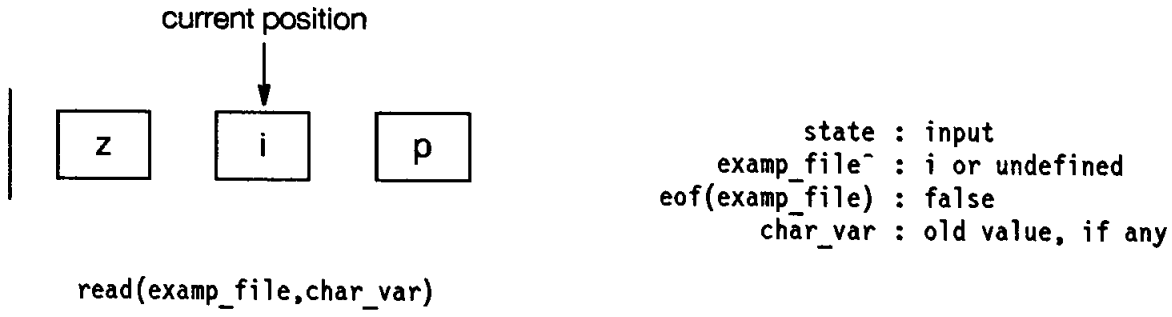
```
read(ff,v1);
read(ff,v2);
  .
  .
  .
read(ff,vn);
```

Example

```
read(file_var,variable)
read(file,variable1,...,variablen)
read(variable)
read(variable1,...,variablen)
```

Illustration

Suppose examp_file is a file of char opened in the input state. The current position is at the second component. To read the value of this component into char_var, we call read:



Implicit Data Conversion.

If *f* is a textfile, its components are type char. The parameter, *v*, however, need not be of type char. It may be any simple, string, or PAC type, which is an HP extension. The read procedure performs an implicit conversion from the ASCII form that appears in the textfile *f* to actual form stored in the variable *v*.

If *v* is type real, longreal, integer, or an integer subrange, the read(*f,v*) operation searches *f* for a sequence of characters which satisfies the syntax below for these types. The search skips preceding blanks or end-of-line markers. If *v* is longreal, the result is independent of the letter preceding the scale factor.

It is an error if the read operation finds no non-blank characters or a

faulty sequence of characters, or if the value is outside the range of *v*. After *read*, a subsequent reference to the buffer variable *f* ^ actually loads the buffer with the character immediately following the number previously read. Also note that eof is false if a file has more blanks or line markers, even though it contains no more numeric values.

If *v* is a variable of type string or PAC, then read(*f, v*) fills *v* with characters from *f* up to the number of elements of *v*. When *v* is type PAC and eoln(*f*) becomes true before *v* is filled, the operation puts blanks in the rest of *v*. If *v* is type string and eoln(*f*) becomes true before *v* is filled to its maximum length, no blank padding occurs. Strlen(*v*) then returns the actual number of characters in *v*. If eoln(*f*) is true when the call is made, no additional characters are read from *f*. The length of a string variable is set to zero, and PAC variables are filled with blanks. Readln must be used to proceed to the next line.

If *v* is a variable of an enumerated type, read(*f, v*) searches *f* for a sequence of characters satisfying the syntax of an HP Pascal identifier. The search skips preceding blanks and line markers. Then the operation compares the identifier from *f* with the identifiers which are values of the type of *v*, ignoring upper and lower case distinctions. Finally, it assigns an appropriate value to *v*. It is an error if the search finds no non-blank characters, if the string from *f* is not a valid HP Pascal identifier, or if the identifier does not match one of the identifiers of the type of *v*.

Table 10-2 shows the results of calls to read with various sequences of characters for different types of *v*.

Table 10-2. Implicit Data Conversion

Sequence of Characters in f Following Current Position	Type of v	Result Stored in v
(space)(space)1.850	real	1.850
(space)(linemarker)(space)1.850	longreal	1.850
10000(space)10	integer	10000
8135(end-of-line)	integer	8135
54(end-of-line)36	integer	54
1.583E7	real	1.583x10(7)
1.583E+7	longreal	1.583x10(7)
(space)Pascal	string[5]	'_Pasc'
(space)Pas(end-of-line)cal	string[9]	'_Pas'

(space)Pas(end-of-line)cal	PAC {length 9}	'_Pas_____'
(end-of-line)Pascal	PAC {length 5}	'_____'
(space)Monday(space)	enumerated	MONDAY

readdir

Usage

```
readdir(f, k, v)
readdir(f, k, v1, ..., vn)
```

Parameters

f A file variable open to read that is not a textfile.

k The index of a component in *f*.

v The name of a variable or component of a structure whose type is not FILE and does not contain a component of type FILE.

Description

The procedure readdir(*f, k, v)* places the current position at component *k*, and then reads the value of that component into *v*. The index, *k*, is relative to the beginning of the file. Formally, this is equivalent to:

```
seek(f,k);
read(f,v);
```

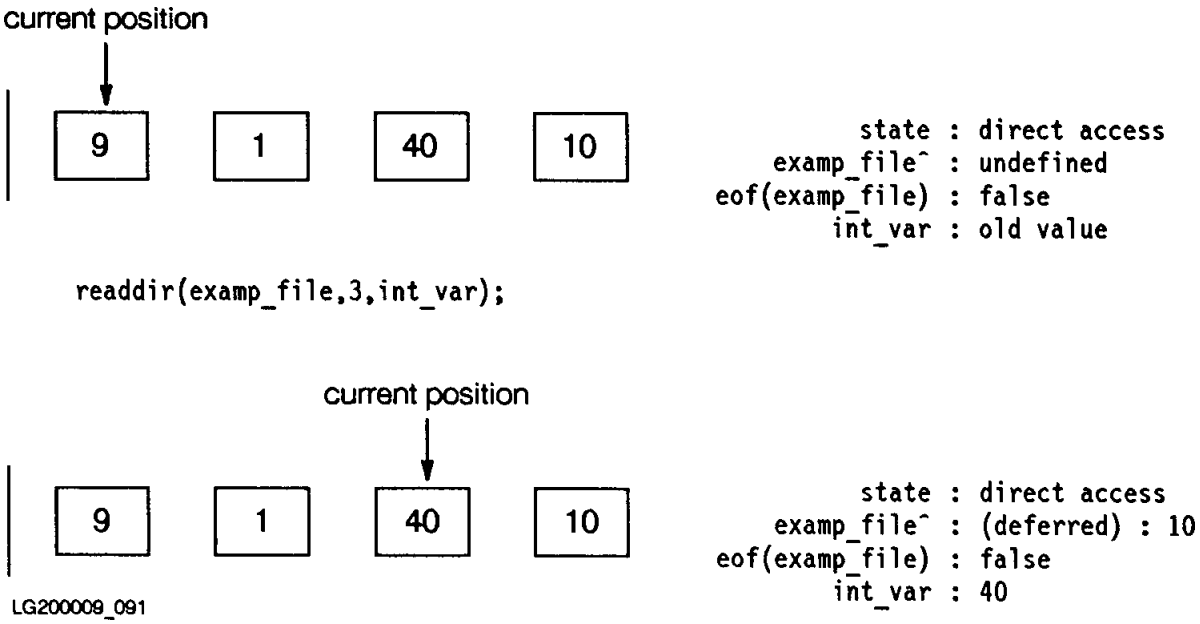
The call get(*f*) is not required between seek and read because of the definition of read. The procedure readdir can be used only with files opened for direct access. Therefore, a textfile cannot appear as a parameter for readdir.

Example

```
readdir(file_var,indx,variable)
readdir(file_var,indx,variable1,...,variablen)
```

Illustration

Suppose examp_file is a file of integer with four components just opened in the direct access state. The current position is the first component. To read the third component into int_var, readdir is called. After readdir executes, the current position is the fourth component.



readln

Usage

```
readln(f)
readln(f, v)
readln(f, v1, ..., vn)
readln
readln(v)
readln(v1, ..., vn)
```

Parameters

f A textfile variable. The system uses the standard file input if *f* is omitted.

v The name of a variable or component of a structure whose type is not FILE and does not contain a component of type FILE.

Description

The procedure readln*(f)* reads zero or more values from a textfile and then advances the current position to the beginning of the next line. The operation performs implicit data conversion if *v* is not type char, string, or PAC. The call readln(f,v1,...,vn) is equivalent to:

```
read(f,v1,...,vn);
readln(f);
```

If the parameter, *v*, is omitted, readln simply advances the current position to the beginning of the next line.

Example

```
readln(file)
readln(file,variable)
readln(file,variable1,...,variablen)
readln(variable)
readln(variable1,...,variablen)
readln
```

reset

Usage

```
reset(f)
reset(f, s)
reset(f, s, t)
```

Parameters

f A file variable that may not be omitted.

s The name of a physical file that the system associates with *f*. *s* may be a string or PAC expression.

t An options string that may be a string or PAC expression whose value is implementation dependent.

Description

The procedure reset*(f)* opens the file *f* in the input state and places the current position at the first component. The contents of *f*, if any, are undisturbed. The file *f* may then be read sequentially.

If *f* is not empty, eof*(f)* is false, and a subsequent reference to the buffer variable *f^* actually loads the buffer with the first component. The components of *f* may now be read in sequence. If *f* is empty, however, eof*(f)* is true and *f^* is undefined, then subsequent calls to read are errors.

If *f* is already open at the time reset is called, the system automatically closes and then reopens it, retaining the contents of the file. If the parameter *s* is specified, the system closes any physical file previously associated with *f*.

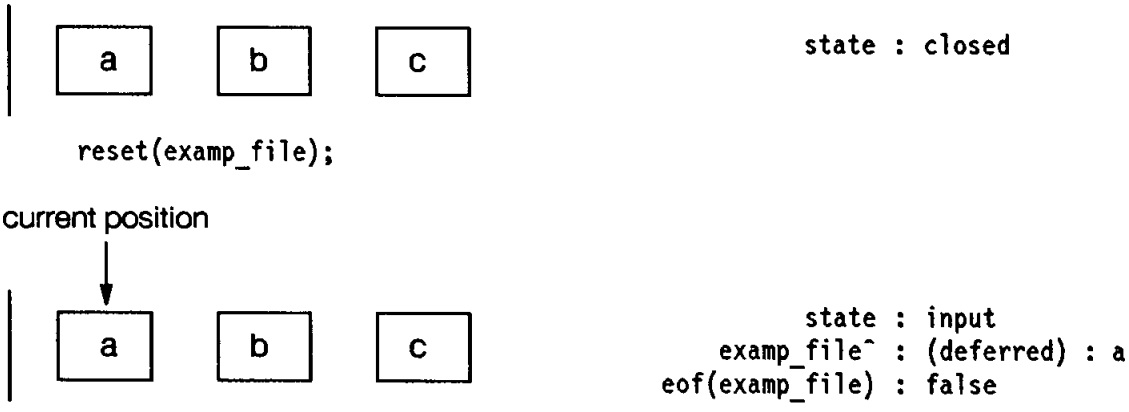
When *f* does not appear as a program parameter and *s* is not specified, the system maintains any previous association of a physical file with *f*. For more information on opening files, see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

Example

```
reset(file_var)
reset(file_var,file_name)
reset(file_var,file_name,opt_str)
```

Illustration

Suppose examp_file is a closed file of char with three components. To read sequentially from examp_file, we call reset:



rewrite

Usage

```
rewrite(f)
rewrite(f, s)
rewrite(f, s, t)
```

Parameters

f A file variable that may not be omitted.

s The name of a physical file the system associates with *f*.

t May be a string or PAC expression whose value is implementation dependent.

Description

The procedure rewrite*(f)* opens the file *f* in the output state and places the current position at the first component. The system discards any previously existing components of *f*. The function eof*(f)* returns true and the buffer variable *f* ^ is undefined. The file *f* may now be written sequentially.

If *f* is already open at the time rewrite is called, the system closes it automatically, flushes the buffers, and then reopens it, losing the contents of the file. If *s* is specified, the system closes any physical file previously associated with *f* and associates *s* with *f*.

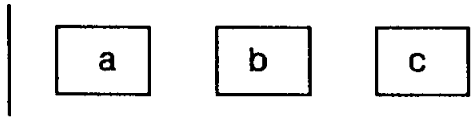
When *f* does not appear as a program parameter and *s* is not specified, the system maintains any previous association of a physical file with *f*. If there is no such association, it opens a temporary, nameless file. This file cannot be saved. It becomes inaccessible after the process terminates or the physical-to-logical file association changes. For more information, see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

Example

```
rewrite(file)
rewrite(file,file_name)
rewrite(file,file_name,opt_str)
```

Illustration

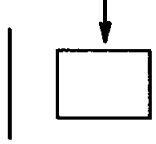
Suppose examp_file is a closed file of char with three components. To discard these components and write sequentially to examp_file, rewrite is called.



state : closed

rewrite(examp_file);

current position



state : output
examp_file^ : undefined
eof(examp_file) : true

seek

Usage

```
seek(f, k)
```

Parameters

f A file variable that must be opened in the direct access state. It may not be a textfile.

k The integer index of a component of *f*. This must be an integer expression >0.

Description

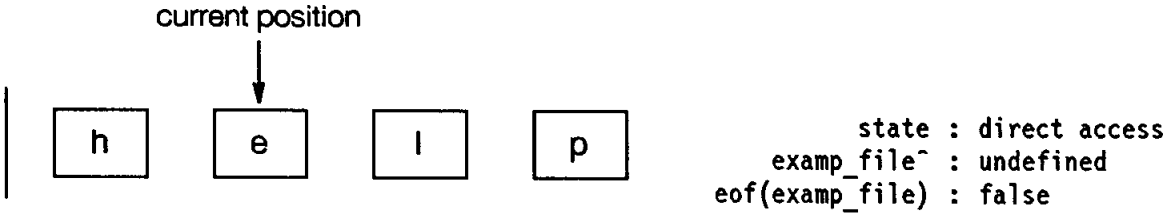
The procedure seek*(f, k)* places the current position of *f* at component *k*. If *k* is greater than the index of the highest-indexed component ever written to *f*, the function eof*(f)* returns true, otherwise false. The buffer variable *f* ^ is undefined following the call to seek. It is an error if *f* is not open in the direct access state, or *k* is greater than maxpos(*f*). The index, *k*, is relative to the beginning of the file.

Example

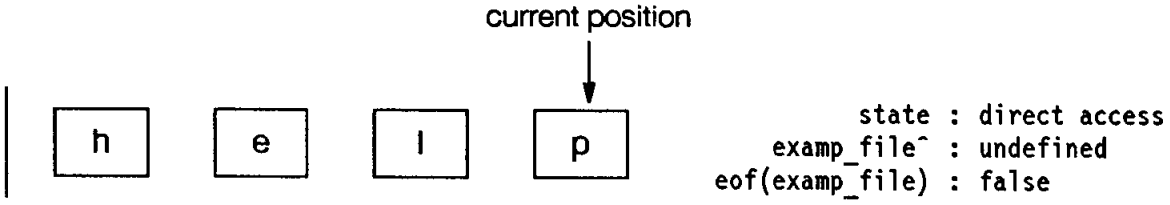
```
seek(file_var,indx)
```

Illustration

Suppose examp_file is a file of char with four components opened for direct access. The current position is the second component. To change it to the fourth component, seek is called.




```
seek(examp_file,4);
```



write

Usage

```
write(f, e)
write(f, e1, ..., en)
write(e)
write(e1, ..., en)
```

Parameters

f A file variable that must be open in the output or direct access state.

e A variable or expression whose type is not FILE and which does not contain a component of type FILE.

Description

The procedure write*(f, e)* assigns the value of *e* to the current component of *f* and then advances the current position. After the call to write, the buffer variable *f* ^ is undefined. It is an error if *f* is not open in the output or direct access state. It is also an error if the current position of a direct access file is greater than maxpos (*f*).

If *f* is not a textfile, *e* must be an expression whose result type is assignment compatible with the components of *f*. If *f* is a textfile, *e* may be an expression whose result type is any simple, string, or PAC type. Also, the value of *e* may be formatted as it is written to a textfile as described later in this chapter.

The call write*(f, e)* is equivalent to accessing the file variable, *f*, and establishing a reference to that file variable for the remaining execution of the statement denoted by *ff*.

The call write*(f,e1,...en)* is equivalent to:

```
write(ff,e1);
write(ff,e2);
 .
 .
write(ff,en);
```

Example

```
write(file_var,exp:5)
write(file_var,exp1,...,expn)
write(exp)
write(exp1,...,expn)
```

Illustration

Suppose examp_file is a file of integer opened in the output state, and that one number has been written to it. To write another number, write is called again:


```
write(examp_file,19);
```

Formatting of Output to Textfiles

When *f* is a textfile, the result type of *e* need not be char. It may be any simple, string, or PAC type, or a string literal. The value of *e* may be formatted as it is written to *f* using the integer field-width parameters *m* and, for real or longreal values, *n*. If *m* and *n* are omitted, the system uses default formatting values. Therefore, three forms of *e* are possible:

```
e      {default formatting}
e:m    {when e is any type}
e:m:n  {when e is real or longreal}
```

Table 10-3 shows the system default values for *m*.

Table 10-3. Default Field Widths

Type of e	Default Field Width (m)
char	1
integer	12
real	12
longreal	20
bit16	12
bit32	12
bit52	12
longint	12
shortint	12
boolean	5 *
enumerated	length of identifier
string	current length of string
PAC	length of PAC

string literal	length of string literal

* If $STANDARD_LEVEL$ is not ANSI or ISO, then the default width of TRUE is 4.

NOTE If *e* is Boolean or an enumerated type, the case of the letters written is implementation defined.

When *m* is specified and the value of *e* requires less than *m* characters for its representation, the operation writes *e* on *f* preceded by an appropriate number of blanks. If the value of *e* is longer than *m*, it is written on *f* without loss of significance; such that *m* is defeated, provided that *e* is a numeric type. Otherwise, the operation writes only the leftmost *m* characters. *m* may be 0 if *e* is not a numeric type.

When *e* is type real or longreal, you may specify *n* as well as *m*. In this case, the operation writes *e* in fixed-point format with *n* digits after the decimal point. If *n* is 0, the decimal point and subsequent digits are omitted. If *n* is not specified, the operation writes *e* in floating-point format consisting of a coefficient and a scale factor. In no case is it possible to write more significant digits than the internal representation contains. This means write may change a fixed-point format to a floating-point format in certain circumstances.

Example

```
PROGRAM show_formats (output);
VAR
  x: real;
  lr: longreal;
  george: boolean;
  list: (yes, no, maybe);
BEGIN
  writeln(999);              {default formatting}
  writeln(999:1);            {format defeated}
  writeln('abc');
  writeln('abc':2);          {string literal truncated}
  x:= 10.999;
  writeln(x);                {default formatting}
  writeln(x:25);
  writeln(x:25:5);
  writeln(x:25:1);
  writeln(x:25:0);
  lr:= 19.1111;
  writeln(lr);
  george:= true;
  writeln(george);           {default format}
  writeln(george:2);
  list:= maybe;
  writeln(list);               {default formatting}
END.
```

Output:

```
          999
999
abc
ab
 1.099900E+01
             1.099900E+01
                 10.99900
```

```
                        11.0
                          11
 1.9111099243164L+01
TRUE
TR
MAYBE
```

writedir

Usage

```
writedir(f, k, e)
writedir(f, k, e1, ..., en)
```

Parameter

f A file variable opened in direct access state.

k The integer index of a component of *f*.

e An expression whose result type must be assignment compatible with the components of *f*.

Description

The procedure writedir*(f, k, e)* places the current position at the component of *f* specified by *k*, and then writes the value of *e* to that component. It is equivalent to:

```
seek(f,k);
write(f,e)
```

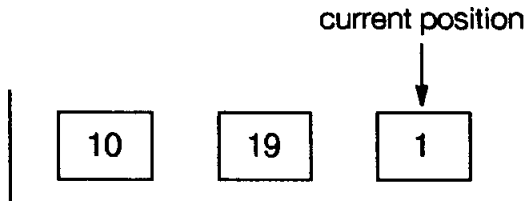
An error occurs if *f* has not been opened in the direct-access state or if *k* is greater than maxpos*(f)*. After writedir executes, the buffer variable *f* ^ is undefined, and the current position is *k* + n, where n is from en.

Example

```
writedir(fil_var,indx,exp)
writedir(fil_var,indx,exp1,....,expn)
```

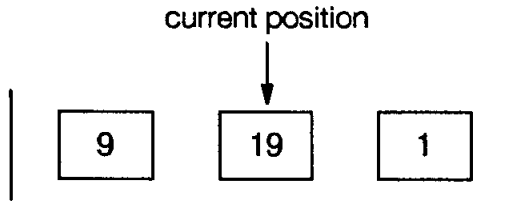
Illustration

Suppose file examp_file is a file of integer opened for direct access. The current position is the third component. To write a number to the first component, we call writedir:




```
                                  state : direct
examp_file^ (deferred) : undefined
       eof(examp_file) : false
```

```
writedir(examp_file,1,9);
```




```
                                  state : direct
                       examp_file^ : undefined
                  eof(examp_file) : false
```

writeln

Usage

```
writeln(f)
writeln(f, e)
writeln(f, e1, ..., en)
writeln
writeln(e)
writeln(e1, ..., en)
```

Parameters

f A file variable for a text file opened in the output state. The system uses the standard file output if *f* is omitted.

e A variable or expression whose type is not FILE and does not contain a component of type FILE.

Description

The procedure writeln*(f, e)* writes the value of the expression *e* to the textfile *f*, appends an end-of-line marker, and places the current position immediately after this marker. After execution, the file buffer *f^* is undefined, and eof*(f)* is true. You may write the value of *e* with the formatting conventions described for the procedure write.

The call writeln*(f, e1,..., en)* is equivalent to

```
write(f,e1);
write(f,e2);
  .
  .
  .
write(f,en);
writeln(f)
```

The call writeln without the file or expression parameters effectively inserts an end-of-line marker in the standard file output.

Example

```
writeln(fil_var)
writeln(fil_var,exp:4)
```

```
writeln(fil_var,exp1,...,expn)
writeln(exp)
writeln(exp1,...,expn)
writeln
```

Chapter 11 System Programming Extensions

This chapter describes extensions to HP Pascal for systems programming. The following subjects are covered:

* pointers
* type coercion
* error handling
* parameter mechanisms
* crunched packing
* routine mechanisms
* predefined routine

Some HP implementations of Pascal do not support all of these features. Any implementation that has system programming extensions support the following:

* anyptr type
* the form of sizeof that accepts variables
* type coercion
* ANYVAR parameters
* TRY-RECOVER statement
* PROCEDURE and FUNCTION variables
* the predefined function addr

The motivations for providing the system programming extensions are:

* Pascal is very strict with regard to type checking. Although this eases the burden on the user by permitting the compiler to check the validity of an operation, it is sometimes necessary to bypass this strict type checking.
* Pascal was originally designed as a language for teaching programming. Because of this, serious attention is not paid to such issues as recovery from run-time errors and the creation and maintenance of large software systems.
* Pascal was not originally defined to be an efficient systems programming language.

This chapter covers the HP_MODCAL and EXT_MODCAL standard levels.

Figure 11-1 illustrates the relationship between the STANDARD_LEVEL parameters.

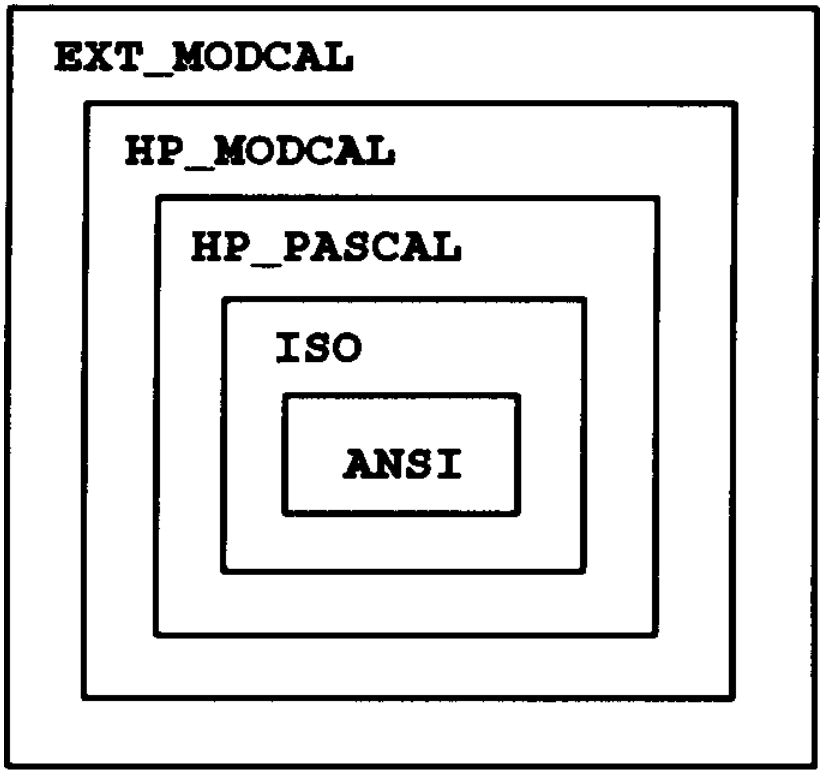


Figure 11-1. Relationship of STANDARD_LEVEL Compiler Option Parameters

The STANDARD_LEVEL compiler option allows the user to choose one of five options which specifies what features or extensions are to be allowed in a given program. The five options correspond to sets which have the relationship depicted in Figure 11-1 above.

If a STANDARD_LEVEL option is not specified, the default feature set is HP_PASCAL. At this level, the compiler does not recognize system programming extension reserved words, and will issue warnings about standard level violations whenever a predefined identifier is encountered.

The list on the following pages delineates the language features that are available for a given STANDARD_LEVEL. ANSI is taken as the base set.

ISO

Conformant Arrays

HP_PASCAL

* Blank padding of PACs and string literals.
* Compiler Directives:

  EXTERNAL INTRINSIC

* Command line parameter handling.
* Compiler Options:

ALIAS	HP_DESTINATION **	RLFILE *
ALIGNMENT	IF	RLINIT *
ANSI *	INCLUDE * **	S300_EXTNAMES
ARG_RELOCATION	INCLUDE_SEARCH * **	SEARCH *
ASSERT_HALT	INLINE	SET
ASSUME	INTR_NAME	SHLIB_CODE * **
BUILDINT	KEEPASMB	SHLIB_VERSION * **
CALL_PRIVILEGE * **	LINES *	SKIP_TEXT
CHECK_ACTUAL_PARM	LIST *	SPLINTR

```
CHECK_FORMAL_PARM          LIST_CODE                  STANDARD_LEVEL  *
CODE                       LISTINTR                   STATEMENT_NUMBER  *  **
CODE_OFFSETS               LITERAL_ALIAS              STDPASCAL_WARN
CONVERT_MPE_NAMES  **      LOCALITY                   STRINGTEMPLIMIT
COPYRIGHT                  MAPINFO                    SUBPROGRAM
COPYRIGHT_DATE             LONG_CALLS                 SYMDEBUG  *  **
ELSE                       MLIBRARY  *  **            SYSINTR  *  **
ENDIF                      NLS_SOURCE  *  **          SYSPROG
EXEC_PRIVILEGE  *  **      NOTES                      TABLES
EXTERNAL                   OPTIMIZE                   TITLE
EXTNADDR                   OS                         TYPE_COERCION
FONT  *                    OVFLCHECK                  UPPERCASE
GLOBAL                     PAGE  *                    VERSION
GPROF  **                  PAGEWIDTH                  VOLATILE
HEAP_COMPACT               PARTIAL_EVAL  *            WARN
HEAP_DISPOSE               POP                        WIDTH
HP3000_16  *               PUSH                       XREF
HP3000_32  *               RANGE  *
```

* Feature is part of standard HP Pascal.

* Feature is MPE/iX system dependent.

** Feature is HP-UX system dependent.

HP_PASCAL (continued)

* Constant expressions.

* Enumerated type, string, PAC I/O. *

* File attribute options to:

append close open reset rewrite

* Functions and procedures returning structured types. *

* Libraries.

* Literal control characters delimited by #. *

* Modules. *

* OTHERWISE in CASE statement. *

* Predefined I/O functions and procedures: *

append lastpos linepos maxpos overprint
position prompt readdir seek writedir

* Predefined string functions and procedures: *

setstrlen str strappend strdelete strinsert
strlen strltrim strmax strmove strpos
strread strrpt strrtrim strwrite

* Ranges in case constants in CASE and variant records. *

* Relaxation in order of declaration section. *

* Special functions and procedures:

assert associate baddress binary*
disassociate getheap halt* hex* mark*
octal* release* rtnheap sizeof waddress

HP-UX:

argc argn argv

MPE/iX, HP-UX:

ccode fnum get_alignment p_getheap p_rtnheap

MPE/iX:

setconvert strconvert

* Structured Constants. *

* Types:

  anyptr bit16 bit32 bit52
  globalanyptr localanyptr longint
  longreal* shortint string*

HP_MODCAL

* ANYVAR parameters.
* Compiler Options:

  TYPE_COERCION (MPE/iX,HP-UX)

* Error handling with:

  escape escapecode TRY-RECOVER

* Procedure and Function Types and Variables.
* Special Predefined Routines:

  addr call fcall statement_number

EXT_MODCAL

* CRUNCHED packing.
* Predefined functions and procedures:

  addtopointer bitsizeof buildpointer cmpbytes
  fast_fill haveextension haveoptvarparm movebyteswhile
  move_fast move_l_to_r move_r_to_l scanuntil scanwhile

* Routine Options:

  DEFAULT_PARMS EXTENSIBLE INLINE UNCHECKABLE_ANYVAR UNRESOLVED

* READONLY parameters.

Language Elements

Reserved Words

The following words are added to the HP Pascal list of reserved words when the system programming extensions are enabled:

Table 11-1. System Programming Extension Reserved Words

Reserved Word	Description
ANYVAR	Routine formal parameter.
CRUNCHED	Structure packing type parameter.
READONLY	Routine formal parameter.
RECOVER	Error recovery statement keyword.
TRY	Error recovery statement keyword.
OPTION	Routine option attribute header.

Note that with the STANDARD_LEVEL set below HP_MODCAL, these identifiers

are not reserved and can be defined by the user.

Predefined Identifiers

The system programming extensions add the following identifiers to the HP Pascal list of predefined identifiers. The compiler issues warning messages if it encounters these identifiers and the standard level is too low.

Like any predefined identifiers, these identifiers may be redefined by the user.

Table 11-2. System Programming Extension Predefined Identifiers

Predefined Identifiers	Description
addtopointer	Address arithmetic
anyptr	Predefined pointer type
bitsizeof	Predefined size function
buildpointer	Address arithmetic
call	Procedure variables
escape	Error recovery
escapecode	Error handling
fcall	Function variables
fast_fill	Predefined move procedure
globalanyptr	Predefined pointer type
haveextension	Parameter mechanism
haveoptvarparm	Parameter mechanism
localanyptr	Predefined pointer type
move_fast	Predefined move procedure

move_l_to_r	Predefined move procedure
move_r_to_l	Predefined move procedure
sizeof	Predefined size function

Data Types

Figure 11-2 summarizes the types that are supplied by the system programming extensions. A detailed discussion of the data types follows in this chapter. This figure augments the HP Pascal data types summarized in Figure 11-1 . Note that the HP Pascal predefined data types are highlighted.

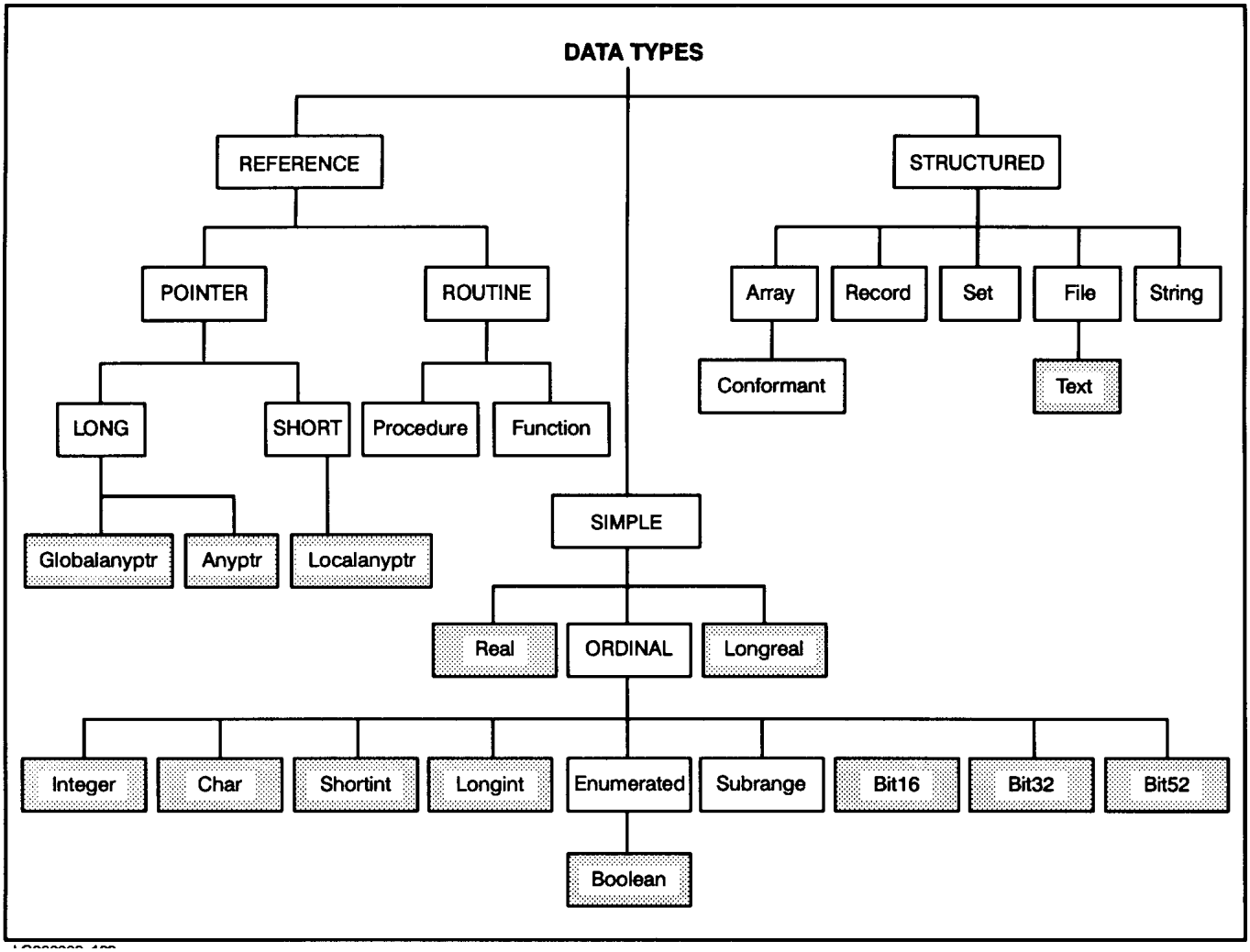


Figure 11-2. Extended Data Types

Structured Types

CRUNCHED.

In Pascal, a structure (array, record, or set) can be unpacked or packed. Packed structures are declared by specifying the reserved word PACKED at

the start of a structured type declaration.

The system programming extensions define a third type of packing in addition to unpacked and packed, namely CRUNCHED.

The reserved word CRUNCHED indicates that the components of a structured type (array, record, or set) are allocated contiguously, first to last, in a bit-aligned fashion with no intervening unused bits. Syntactically, the word CRUNCHED may be substituted for the word PACKED.

The primary purpose of crunched packing is to provide a map from data item type to data representation that is independent of the implementation and the packing algorithm. For that reason, machine dependent types such as real, longreal, and file are not allowed in crunched structures.

Example

```
TYPE
   rec = RECORD
      a : type_a;
      b : type_b;
      c : type_c;
   END;

   crec = CRUNCHED RECORD
      a : type_a;
      b : type_b;
      c : type_c;
   END;
```

In a crunched structure, each component is allocated the minimum number of bits required to represent that type, and each component is aligned in such a way that there are no unused bits between it and the previous component.

The first declaration for rec in the previous example may lead to the following storage allocation for an arbitrary processor:

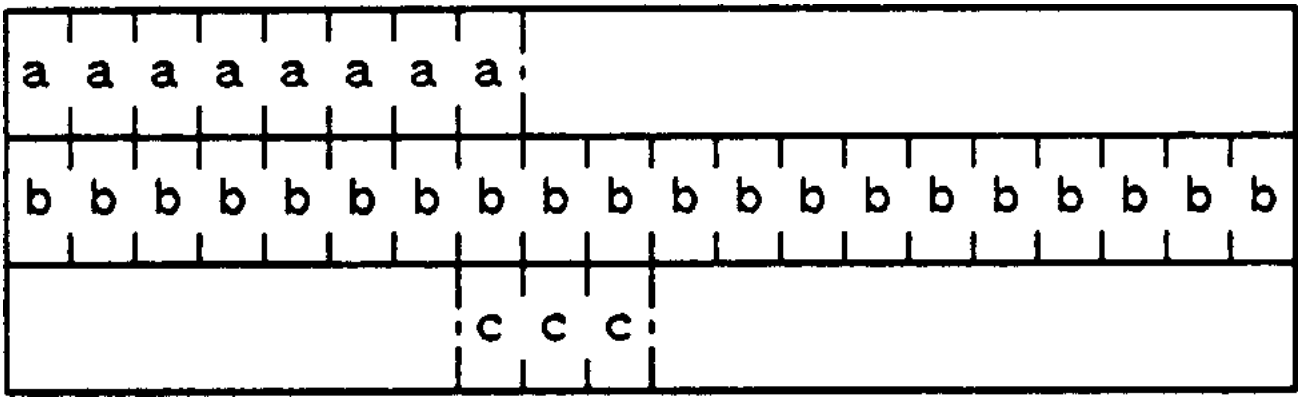


Figure 11-3. Layout of a Record

Note that there are unused bits between the fields a and b, and between the fields b and c.

The crunched record declaration for crec, that is identical to the uncrunched record rec with the addition of the reserved word CRUNCHED, would lead to the following storage allocation:

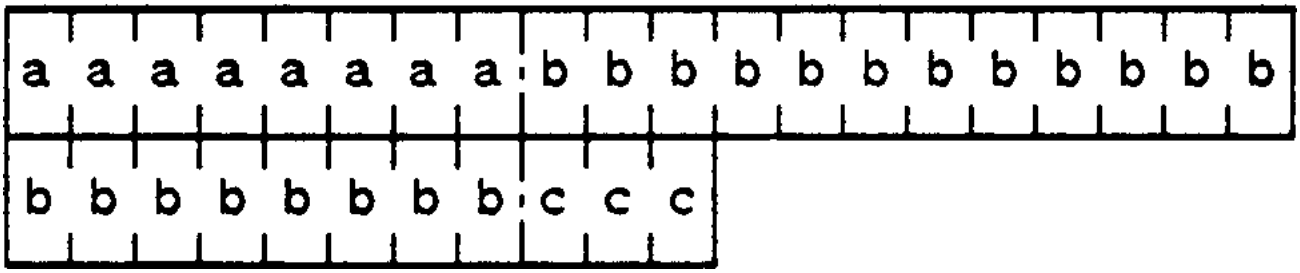


Figure 11-4. Layout of a Crunched Record

Note that there are no wasted bits between fields in the crunched record.

The number of bits used to represent each component of a crunched structured type is the minimum needed to represent the values associated with that component. The calculation of the minimum number of bits for various types is:

* Record, Array Types.

  The sum of the minimum number of bits required to represent each component. If the record has variants, consider the size of the largest variant.

* Set Types (of the form set of low .. high).

  The ordinal value of high minus the ordinal value of low plus one:

  ord(high) - ord(low) + 1

* Char and Enumeration Based Types (of the form low .. high).

  The next larger integer (the ceiling) of the logarithm base 2 of the successor of the ordinal value of the upper bound, or one, whichever is greater:

  max(ceil [log2(ord(high) + 1)], 1)

  Integer Based Types (of the form low .. high)

  The next larger integer (the ceiling) of the logarithm base 2 of the maximum of the absolute value of the ordinal value of the lower bound, and the successor of the absolute value of the ordinal value of the upper bound, or one, whichever is greater:

  max(ceil [log2(max(|low|, |high| + 1))], 1)

  If the type is signed (the lower bound is less than zero), then add one to the size.

Table 11-3 shows the lower and upper bound ranges and number of bits allocated for unsigned subranges. Table 11-4 gives the same information for signed subranges.

Table 11-3. Number of Bits Allocated for Unsigned Subranges

Lower Bound Range	Upper Bound Range	# Bits Allocated
>= 0	0..1	1
>= 0	2..3	2
>= 0	4..7	3
>= 0	8..15	4
>= 0	16..31	5
>= 0	32..63	6
>= 0	64..127	7
>= 0	128..255	8
>= 0	256..511	9
>= 0	512..1023	10
>= 0	1024..2047	11
>= 0	2048..4095	12
>= 0	4096..8191	13
>= 0	8192..16383	14
>= 0	16384..32767	15
>= 0	32768..65535	16
>= 0	65536..131071	17
>= 0	131072..262143	18
>= 0	262144..524287	19
>= 0	524288..1048575	20
>= 0	1048576..2097151	21
>= 0	2097152..4194303	22
>= 0	4194304..8388607	23
>= 0	8388608..16777215	24
>= 0	16777216..33554431	25
>= 0	33554432..67108863	26
>= 0	67108864..134217727	27
>= 0	134217728..268435455	28
>= 0	268435456..536870911	29
>= 0	536870912..1073741823	30
>= 0	1073741824..2147483647	31

Table 11-4. Number of Bits Allocated for Signed Subranges

Lower Bound Range	Upper Bound Range	#Bits Allocated
-1	0	1
-2	1	2
-4..-3	2..3	3
-8..-5	4..7	4
-16..-9	8..15	5
-32..-17	16..31	6
-64..-33	32..63	7
-128..-65	64..127	8
-256..-129	128..255	9
-512..-257	256..511	10
-1024..-513	512..1023	11
-2048..-1025	1024..2047	12
-4096..-2049	2048..4095	13
-8192..-4097	4096..8191	14
-16384..-8193	8192..16383	15
-32768..-16385	16384..32767	16
-65536..-32769	32768..65535	17
-131072..-65537	65536..131071	18
-262144..-131073	131072..262143	19
-524288..-262145	262144..524287	20
-1048576..-524289	524288..1048575	21
-2097152..-1048577	1048576..2097151	22
-4194304..-2097153	2097152..4194303	23
-8388608..-4194305	4194304..8388607	24
-16777216..-8388609	8388608..16777215	25
-33554432..-16777217	16777216..33554431	26
-67108864..-33554433	33554432..67108863	27

-134217728..-67108865	67108864..134217727	28
-268435456..-134217729	134217728..268435455	29
-536870912..-268435457	268435456..536870911	30
-1073741824..-536870913	536870912..1073741823	31
-2147483648..-1073741825	1073741824..2147483647	32

Example

```
TYPE
   cr1_t = CRUNCHED RECORD
      f1 :   0..15;                    { takes 4 bits               }
      f2 :  -1..15;                    { takes 5 bits               }
      f3 : -16..15;                    { takes 5 bits               }
      f4 :  13..15;                    { takes 4 bits               }
      END;                             {              total: 18 bits }

   cr2_t = CRUNCHED RECORD
      f1 : CRUNCHED set of  0..15;     { takes 16 bits              }
      f2 : CRUNCHED set of 13..15;     { takes 3 bits               }
      f3 : CRUNCHED set of -5..5;      { takes 11 bits              }
      END;                             {              total: 30 bits }

   cr3_t = CRUNCHED RECORD
      f1 : integer;                    { takes 32 bits              }
      CASE tag : Boolean OF            { takes 1 bit                }
         true: ( v1 : cr1_t );         { takes 18 bits              }
         false:( v2 : cr2_t );         { takes 30 bits              }
         END;                          {              total: 63 bits }
```

The restrictions that apply to packed types also apply to crunched types. In particular, it is illegal:

* To pass a component of a crunched structure as a reference parameter.
* To take the address of a component of a crunched structure.

In addition:

* File types cannot be crunched.
* Structured types that contain file, real, longreal, string, or pointer types cannot be crunched.
* All structured types contained in a crunched structured type must also be crunched.
* All integer based types and enumeration based types are represented with the most significant bit first through least significant bit last. Byte swapping is not permitted.

Pointer Types

In HP Pascal, *pointers* are designators that point only to a *specific class of objects*, namely *objects on the heap*.

When using the system programming extensions, pointers can point to any data; that is, objects on the heap, as well as global and local variables. In this sense pointers truly are addresses.

In HP Pascal, the only way to create a pointer is by calling the predefined procedure NEW or the intrinsic getheap to dynamically allocate a heap object. In order to create pointers, the system programming extensions define the addr function that returns the address of its argument, and the functions buildpointer and addtopointer that perform address arithmetic.

There are three predefined pointer types defined in the system programming extensions that allow relaxed type checking of pointers. These are anyptr, localanyptr, and globalanyptr.

Short and Long Pointers.

The system programming extensions define two classes of pointers: *short*

and *long* pointers.

Long pointers can point to any addressable object on the system (in this sense addressable in terms of the representability of an address, as opposed to allowed access rights).

Short pointers can point to a subset of the objects addressable by long pointers. A subset of the object addressable by short pointers are objects in the heap. By default, all user declared pointers are short pointers.

The following diagram explains the relationship between these classes of pointers.

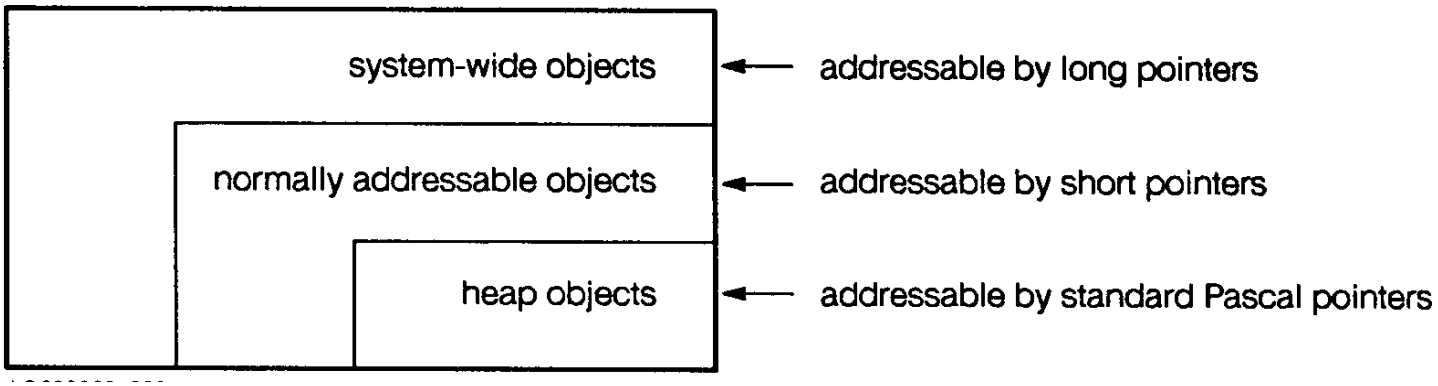


Figure 11-5. Pointer Class Relationship

Note that in some implementations, long and short pointers may be identical; in other words, the collection of objects that long and short pointers can point to may be the same.

The compiler option EXTNADDR can be used to specify that a given user defined pointer type is to be a long pointer.

Localanyptr.

The predefined type localanyptr is a pointer type that is assignment compatible with any other pointer type. It can be used to defeat type checking on pointers.

A pointer of any type can be assigned to a pointer of type localanyptr, and a pointer of type localanyptr can be assigned to any pointer type. However, since pointers of type localanyptr are not bound to a base type, they cannot be dereferenced. (In order to dereference a pointer of type localanyptr, it must first be type coerced or assigned to a proper pointer type).

localanyptr takes the form of a short pointer. It may only be able to represent a subset of the addresses on a machine. On implementations where short and long pointers are not the same, localanyptr is more efficient than globalanyptr.

Permissible Operators

assignment	:=
relational	=, <>

Example

```
VAR
   ptr1   : pointer_type_1;
   ptr2   : pointer_type_2;
   anyp   : localanyptr;

BEGIN
   ...
   anyp := ptr1;
```

```
   anyp := ptr2;
   ...
   ptr1 := anyp;
   ...
END;
```

Globalanyptr.

The predefined type globalanyptr is a pointer type that is assignment compatible with any other pointer type. It can be used to defeat type checking on pointers.

A pointer of any type can be assigned to a pointer of type globalanyptr, and a pointer of type globalanyptr can be assigned to any pointer type. However, since pointers of type globalanyptr are not bound to a base type, they cannot be dereferenced. (In order to dereference a pointer of type globalanyptr, it must first be type coerced or assigned to a proper pointer type.)

Globalanyptr takes the form of a *long pointer*. It can represent any address on the machine. A more efficient type of pointer called localanyptr can be used in a program that has no need for long pointers.

Permissible Operators

assignment	:=
relational	=, <>

Example

```
VAR
   ptr1   : pointer_type_1;
   ptr2   : pointer_type_2;
   anyp   : globalanyptr;

BEGIN
   ...
   anyp := ptr1;
   anyp := ptr2;
   ...
   ptr1 := anyp;
   ...
END;
```

Anyptr.

The predefined type anyptr is a pointer type that is assignment compatible with any other pointer type. It can be used to defeat type checking on pointers.

A pointer of any type can be assigned to a pointer of type anyptr, and a pointer of type anyptr can be assigned to any pointer type. However, since pointers of type anyptr are not bound to a base type, they cannot be dereferenced. In order to dereference a pointer of type anyptr, it must first be type coerced or assigned to a proper pointer type.

Anyptr takes the form of a *long pointer*. It can represent any address on the machine. A more efficient type of pointer called localanyptr can be used in a program that has no need for long pointers.

Anyptr is equivalent to globalanyptr; however, globalanyptr and localanyptr are the recommended types to use.

Permissible Operators

assignment	:=
relational	=, <>

Example

```
VAR
   ptr1   : pointer_type_1;
   ptr2   : pointer_type_2;
   anyp   : anyptr;
```

```
BEGIN
   ...
   anyp := ptr1;
   anyp := ptr2;
   ...
   ptr1 := anyp;
   ...
END;
```

The above example illustrates that a variable of type anyptr is assignment compatible with any other pointer type.

Example

```
VAR
   ptr1   : pointer_type_1;
   ptr2   : pointer_type_2;

PROCEDURE proc( ptr : anyptr );

BEGIN
  ...
END;

BEGIN
   proc( ptr1 );
   proc( ptr2 );
END;
```

In the above example, the routine proc can accept any pointer as an actual parameter because the type of the formal parameter is anyptr. anyptr is assignment compatible with any pointer type.

Example

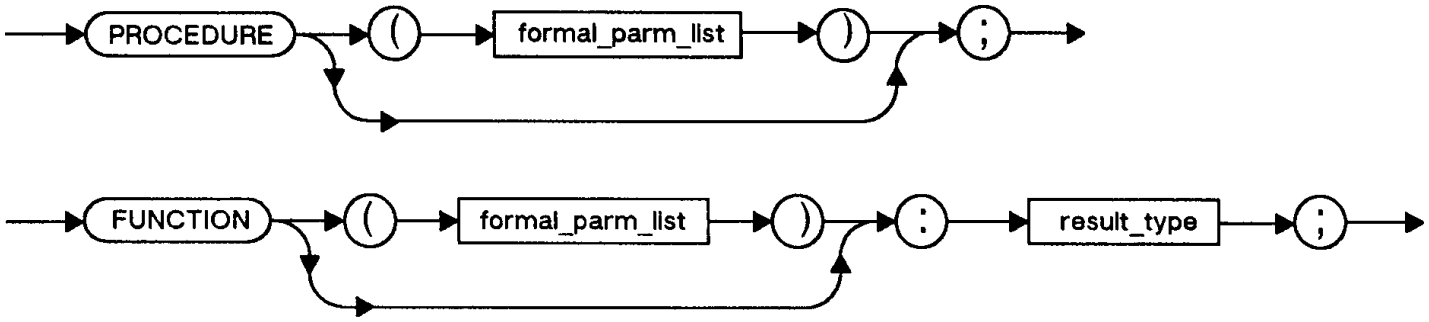
```
TYPE
   pointer_type = ^record_type;
   record_type = RECORD
          int : integer;
   END;

VAR
   i    : integer;
   anyp : anyptr;

BEGIN
   i := pointer_type( anyptr )^.int;
END;
```

In the above example, the pointer anyp is dereferenced to access a field in a record. Since an anyptr is not bound to a base type, the pointer must first be type-coerced to a pointer type corresponding to the structure to which anyp is pointing.

PROCEDURE and FUNCTION Types

In Pascal, PROCEDURE and FUNCTION parameters allow dynamic reference to procedures and functions where the exact instance of the procedure or function is not known until run-time. The system programming extensions extend this concept to allow variables as well as parameters that refer to procedures and functions.

Syntax



A parallel can be drawn between pointers and PROCEDURE/FUNCTION variables. While pointers are variables that reference data, PROCEDURE/FUNCTION variables reference code.

Variables of PROCEDURE/FUNCTION types may be assigned procedures and functions that have congruent parameter lists, as defined in HP Pascal. See chapter 8 for more information on parameters. To assign a procedure or function to a PROCEDURE/FUNCTION variable, the routine name is used as a parameter to the addr function. See the section on predefined routines in this chapter for more information on addr.

Any procedure or function assigned must have the same or wider scope than the variable or value parameter to which it is assigned. Any PROCEDURE/FUNCTION variable passed as a reference parameter must have the same or wider scope than the formal parameter to which it is bound.

A PROCEDURE/FUNCTION variable can be assigned NIL.

The procedure or function referenced by a PROCEDURE/FUNCTION variable, may be invoked by calling the predefined procedure call for a PROCEDURE variable or the predefined function fcall for a FUNCTION variable. See the section on predefined routines in this chapter for more information on call and fcall.

Permissible Operators

assignment	:=
relational	=, <>

Standard Procedures

argument	CALL

Standard Functions

argument	FCALL
return	ADDR

Example

```
TYPE
   proc_0_type = PROCEDURE;
   func_0_type = FUNCTION: integer;
   proc_1_type = PROCEDURE( ANYVAR i : integer );
   func_2_type = FUNCTION( VAR s : string;
                               i : integer ): boolean;
VAR
   proc_0 : proc_0_type;
   func_0 : func_0_type;
   proc_1 : proc_1_type;
   func_2 : func_2_type;

PROCEDURE p1; external;
PROCEDURE p2( n : shortint ); external;
PROCEDURE p3( VAR i : integer ); external;

BEGIN
   func_0 := nil;                    { initialized to nil }
```

```
   func_2 := nil;                    { initialized to nil }

   proc_0 := addr( p1 );             { proc_0 now 'points to' p1 }
   proc_1 := addr( p2 );             { illegal: parameters don't match }
   proc_1 := addr( p3 );             { illegal: parameters don't match }
   func_0 := addr( p1 );             { illegal: must be a function }
END.
```

Example

```
TYPE
   proc_type = PROCEDURE;

VAR
   proc_var_0 : proc_type;

PROCEDURE proc_1;
VAR
   proc_var_1 : proc_type;

   PROCEDURE proc_2;
   BEGIN {PROCEDURE proc_2}
      ...
   END; {PROCEDURE proc_2}

BEGIN {PROCEDURE proc_1}
   proc_var_0 := addr( proc_1 );
   proc_var_1 := addr( proc_1 );
   proc_var_0 := addr( proc_2 );      { illegal: scoping violation }
   proc_var_1 := addr( proc_2 );
END; {PROCEDURE proc_1}
```

Expressions

Type Coercion

Pascal is very strict with respect to type checking. In any operation such as assignment, binary operations, passing parameter, or indexing, relevant types must be compatible according to the HP Pascal rules of compatible types. Refer to "Type Compatibility" for more information.

Type coercion allows the user to selectively circumvent the normally strong type checking. The system programming extensions support several forms of type coercion including ANYVAR, reference, and value. ANYVAR type coercion (using the formal parameter mechanism ANYVAR) is described in this chapter under "Procedures and Functions" .

Reference type coercion consists of type coercion of an actual parameter that is being passed to a reference formal parameter, or type coercing a pointer to a different pointer type before a dereference.

Value type coercion consists of type coercion of a constant, variable, function result, or expression to a different type.

The syntax for type coercion looks like the application of a function to an expression, where the name of the function is the name of the target type of the coercion.

Syntax

Expression:

The expression being coerced may be a constant, variable, function result, or expression involving unary and binary operators.

Syntactically, value type coercion is allowed:

* In an expression.
* On the right-hand side of an assignment statement.
* On an actual parameter.

By default, the compiler does not allow value type coercion. The compiler option TYPE_COERCION allows the user to enable a certain level of type coercion. There are three classes of type coercion based on the source and target types: ordinal, pointer, and free union type coercion. Ordinal and pointer type coercions are enabled by specifying the conversion level of type coercion. Instances of free union type coercion are enabled by specifying one of *structural*, *representation*, *storage*, or *noncompatible* type coercion.

Ordinal Type Coercion.

The *ordinal* types are viewed as different sets of names for the points on the integer number line. Given this view of ordinals, value type coercion of one ordinal type to another is simply a renaming operation.

A type coercion expression is considered an *ordinal coercion*, if both the source expression (expression being coerced) and the target type (type to which the expression is being coerced) are any of the following types:

* The predefined types integer, shortint, char, Boolean, and Bit16.
* A user-declared enumerated type.
* A user-declared subrange type.

If the value of the source expression is out of range with respect to the allowed values of the target type, a subrange violation occurs. If range checking is on, this causes a run-time error.

Example

```
TYPE
   color_t = (red,orange,yellow,green,chartreuse,blue,indigo,violet);

VAR
   i     : integer;
   color : color_t;

BEGIN
   ...
   color := chartreuse;
   i := integer( color );    { i has the value 4 }
   i := 3;
   color := color_t( i );    { color has the value green }
   i := 12;
   color := color_t( i );    { will cause a run-time error }
   ...
END;
```

Pointer Type Coercion.

The pointer types are viewed as virtual addresses. Given this view, value type coercion from one pointer type to another is a mapping from one virtual address to another.

On implementations that have alignment restrictions, it is an error if the alignment of the type that the source expression points to is smaller than the alignment of the type that the target type points to. If range checking is on, this causes a run-time error. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information on alignment.

Coercion from long-to-long and short-to-short pointers is a one-to-one mapping and involves no actual run-time conversion operations.

Coercion from a short to a long pointer, in implementations where long pointers point to a wider class of objects than short pointers, may involve amending the value of the short pointer with additional address

information that a long pointer requires.

Coercion from a long to a short pointer, in implementations where long pointers point to a wider class of objects than short pointers, may involve truncating the value of the long pointer. If this occurs, the short pointer may not be able to address the original object pointed to by the long pointer because of the short pointer's limited addressing. This is an error. If range checking is on, this causes a run-time error.

Example

```
TYPE
   integer_pointer = ^ integer;
   real_pointer    = ^ real;

VAR
   ip : integer_pointer;
   rp : real_pointer;

BEGIN
   ...
   ip := integer_pointer( rp );
   ...
END;
```

Other Type Coercion.

All type coercions that do not fall under the categories of ordinal and pointer type coercion can be viewed as the use of a free union, or tagless variant record, where the implementation overlays the record variants onto the same storage area.

The model for value type coercion:

```
type_1(expression)
```

is equivalent to the function call:

```
f_type_1(expression)
```

where f_type_1 is defined as:

```
FUNCTION f_type_1 (e: type_of_expression):type_1 ;

VAR
   coerce_record : RECORD CASE Boolean OF
      true:  ( source_variant : type_of_expression );
      false: ( target_variant : type_1 );
      END;

BEGIN
   coerce_record.source_variant := e;
   f_type_1 := coerce_record.target_variant;
END;
```

The model for reference type coercion:

```
target_type ( source );
```

is equivalent to:

```
pointer_to_target_type ( addr ( source ) )
```

Whereas both ordinal and pointer type coercions may cause run-time errors if the source values are not representable in the target type, the free union form of type coercion never causes run-time errors.

Depending upon the source and target types, free union type coercion consists of the following levels, listed in increasing order of freedom:

* Structural
* Representation
* Storage
* Noncompatible

Structural. A type coercion expression is considered to be *structural* if the following are true:

* The bitsizes of the source and target types are the same.
* The alignment of the source and target types are the same.
* The source and target types are compatible.
* If the source and target are structured, then the corresponding component in the two structures obey the above three rules of bitsizes, alignment and compatibility.

Structural type coercions are enabled by specifying 'STRUCTURAL' in the compiler option TYPE_COERCION.

Structural type coercion is essentially a renaming of the components of a structure. Because the component types are guaranteed to be the same, the storage allocated for the source and target types is also the same, and reinterpreting the storage of the source as if it was of the target type will produce correct results.

Example

```
$TYPE_COERCION 'STRUCTURAL'$
...
TYPE
   source_t = RECORD
      i : integer;
      b : false..true;
      end;
   target_t = RECORD
      j : minint..maxint;
      c : Boolean;
      END;

VAR
   source : source_t;
   target : target_t;
...
BEGIN
   ...
   target := target_t(source);
   ...
END;
```

In the above example, the two record types are the same: their bitsizes are identical and their corresponding components are the same.

Representation. A type coercion expression is considered to be *representation* type coercion if the bitsizes of the source and target types are the same. The internal structure of structured types for either the source or target does not matter.

Representation type coercions are enabled by specifying 'REPRESENTATION' in the compiler option TYPE_COERCION.

Example

```
$STANDARD_LEVEL 'HP_MODCAL'$
PROCEDURE write_hex( n : integer );

TYPE
   nibble_array = PACKED ARRAY[0..7] OF 0..15;
   hex_digit_t  = array [0..15] OF char;

CONST
   hex_digit = hex_digit_t[ '0', '1', '2', '3', '4', '5', '6', '7',
                            '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'  ];
VAR
   i : 0..7;

BEGIN
   FOR i := 0 to 7 DO
```

```
            $PUSH, TYPE_COERCION 'REPRESENTATION'$
            WRITE( hex_digit[ nibble_array( n )[i] ] );
            $POP$
      END;
```

In the above example, the integer n is treated as an array of nibbles in order to extract each nibble sequentially and write out its value in hexadecimal. Since representation type coercion guarantees that the source and target types are identical in size, the compiler can guarantee that the entire integer is covered by the nibble array: there are no bits missed.

Storage. A type coercion expression is considered to be *storage* type coercion, if the size of the storage allocated for the source is greater than the size of the storage allocated for the target type.

Storage type coercion guarantees that no nonexistent memory is accessed and that no undefined bits are accessed.

The following illustrates storage type coercion. The compiler guarantees that PROC never accesses a part of its formal parameter that is not actually part of the actual parameter. This is because the actual parameter is guaranteed to be larger than or the same size as the formal parameter.

Example

```
      TYPE
         string_1 = STRING [255];
         string_2 = STRING [80]

      VAR
         s1 : string_1;
         s2 : string_2;
      ...

      PROCEDURE PROC (VAR S : STRING_2);

         BEGIN
            ...
         END;
      ...

      $PUSH, TYPE_COERCION 'STORAGE'$
      PROC ( string_2 (s1) );
      $POP$
```

Noncompatible. Noncompatible type coercion permits anything to be coerced to anything. There is no guarantee that the accessed storage exists, nor that there is any accessible storage.

Example

```
      FUNCTION non_protected_space: integer;

      TYPE
         big_index = 0..max_array_size-1;
         big_array = array[big_index] of integer;

      VAR
         idx : big_index;
         int : integer;

      BEGIN
         idx := 0;
         TRY
            WHILE (idx <= max_array_size-1) DO BEGIN
               $PUSH, TYPE_COERCION 'NONCOMPATIBLE'$
               int := big_array( int )[idx];
               $POP$
               idx := idx + 1;
               end;
            non_protected_space := max_array_size-1;
         RECOVER
```

```
        non_protected_space := idx - 1;
    END;
```

The previous example coerces an integer to an array of integers and keeps accessing farther out into the array until it cannot access any further. Note that this code assumes that:

* TRY-RECOVER traps the error condition that occurs when the array access grows beyond the limits of the available space.
* The value of the variable idx is updated correctly when execution is transferred to the RECOVER statement.

Declaration Section

Constant Definition

NIL.

The definition of the predefined constant NIL is expanded for the system programming extensions.

The predefined constant NIL is compatible with any long or short pointer type. When NIL is used in a comparison or assignment, it assumes the pointer class (short or long) of the pointer with which it is being compared, or to which it is being assigned.

The predefined constant NIL is compatible with any PROCEDURE/FUNCTION type. A PROCEDURE/FUNCTION variable that has been assigned the value NIL refers to no procedure or function.

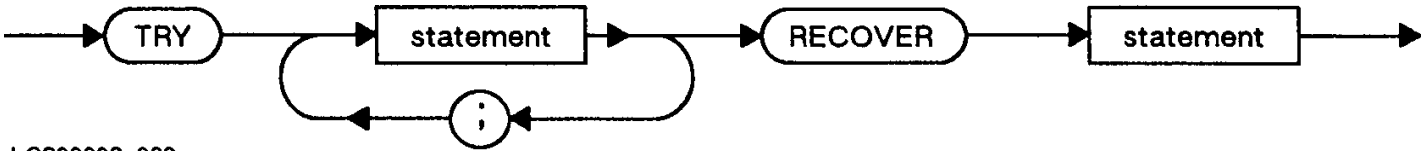
Statements

TRY-RECOVER

A Pascal program that encounters a run-time error is aborted. Because this is not always acceptable, the system programming extensions define the TRY-RECOVER structured statement that allows the user to trap all run-time errors.

The predefined procedure escape allows the user to cause a run-time error to occur, and the predefined function escapecode allows the user to determine the last type of error that occurred. See the section "Error Handling Routines" for more information on escape and escapecode.

Syntax



The statement following the reserved word RECOVER may have a statement label. One can jump to such a label only from within the RECOVER statement itself.

The types of errors that are trapped by TRY-RECOVER are:

* All Pascal run-time errors (defined in Appendix A).
* An implementation defined set of hardware errors.
* An implementation defined set of operating system detected errors.
* All user-generated error conditions (generated by calling escape).

Upon detecting an error in the execution of the body of a TRY-RECOVER statement (the statements between the reserved words TRY and RECOVER, as well as any procedures and functions called from such statements), the following sequence of events occurs:

* The escape code, indicating the type of error that occurred, is saved for later retrieval by the predefined function escapecode.

* The run-time environment is restored to the environment of the most recent TRY-RECOVER statement. This may involve prematurely exiting any nested procedure and function calls and closing any open files local to those routines.
* Execution is transferred to the statement following the reserved word RECOVER.

If no errors are detected within the body of the TRY-RECOVER statement, the recover statement is skipped, and execution continues at the first statement following the TRY-RECOVER statement.

The TRY-RECOVER statement does not trap errors in its recover part (the statement following the reserved word RECOVER). If an error occurs in the execution of the recover part, execution is transferred to the recover part of an enclosing TRY-RECOVER statement. If there is no enclosing TRY-RECOVER statement the program aborts.

The semantics of the TRY-RECOVER statement do not guarantee that the effects of any statements executed in the body of the TRY are valid when executing the RECOVER statement. Certain implementations, however, may guarantee that the effects of any executed statements are valid. Certain other implementations may provide the user with a method of indicating that certain variables preserve their value. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more details. Also see the compiler option "VOLATILE" .

Note that when execution is transferred to the RECOVER statement, the environment in which the error occurred no longer exists. If that environment is required to perform error handling, then trap handlers are required. See the chapter on Error Recovery in the *HP Pascal/iX Programmer's Guide* or in the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information.

Example

```
TRY
   open( f, 'filename' );
RECOVER BEGIN
   writeln( 'open failed' );
   ...
   END;
```

The above code fragment prevents a program from aborting if a file cannot be opened.

Example

```
PROCEDURE proc1;
BEGIN
   ...                  { errors will be trapped in try 0 }
   TRY {try 1}
      ...               { errors will be trapped here in try 1 }
   RECOVER BEGIN
      ...               { errors will be trapped in try 0 }
      END;
   ...                  { errors will be trapped in try 0 }
END;

...

BEGIN
   ...                  { errors will abort the program }
   TRY {try 0}
      ...               { errors will be trapped here in try 0 }
      proc1;
      ...               { errors will be trapped here in try 0 }
   RECOVER BEGIN
      ...               { errors will abort the program }
      END;
   ...                  { errors will abort the program }
```

```
END.
```

In the previous example, any errors occurring in the TRY-RECOVER in proc1 cause execution to be transferred to the recover part of the try statement in proc1. Any errors occurring in the TRY-RECOVER in the outer block, in the recover statement in proc1, and outside of the TRY-RECOVER in proc1 cause execution to be transferred to the recover part of the try statement in the outer block. Any error occurring in the recover statement in the outer block and outside of the TRY-RECOVER statement in outer block, aborts the program because there is no TRY-RECOVER to catch the error.

Example

```
VAR
   int : integer;
...
int := 0;

TRY
   ...
   int := 1;
   ...
   int := 2;
   ...
   int := 3;
   ...

RECOVER BEGIN
   ...
   END;
```

If execution is transferred to the recover statement, there is no guarantee that the variable int has a value other than zero for the following reasons:

* The error could have occurred anywhere within the try body. The first assignment to int may not have been executed yet.
* Even if an assignment statement was executed, the semantics do not guarantee that the actual location of int was updated. If the new value of int was stored in a location other than its memory location, then the transfer of execution to the RECOVER statement does not update the memory location of int.

Procedures and Functions

The system programming extensions define two new formal parameter mechanisms in addition to Pascal value and VAR formal parameters. These mechanisms are ANYVAR and READONLY.

The system programming extensions also define an extension to the procedure and function header syntax for specifying additional attributes of a procedure or function. This extension is *routine options*.

Formal Parameters

The reserved words ANYVAR and READONLY can syntactically replace the reserved word VAR in a formal parameter list specification.

ANYVAR.

This formal parameter mechanism implicitly type coerces the actual parameter to the type of the formal parameter.

A formal ANYVAR parameter represents the actual parameter during execution of the procedure. Any changes in the value of the formal ANYVAR parameter alters the value of the actual parameter. Therefore, it must be a variable-access parameter. The actual parameter may have any type. The formal-ANYVAR parameter, however, is treated within the body of the procedure as a variable of the type specified in its definition.

An additional hidden parameter is passed along with each actual parameter passed to a formal ANYVAR parameter. This hidden parameter is the length in bytes of the actual parameter. This size value can be accessed

through the use of the predefined functions sizeof and bitsizeof. This additional size parameter is not passed when the routine option UNCHECKABLE_ANYVAR is used.

This implicit reference type coercion is independent of the level of type coercion selected when the actual parameter is used.

Example

```
TYPE
   byte = 0..255;
   byte_array = PACKED ARRAY [1..max_bound] OF byte;

VAR
   int : integer;
   rec : record_type;

PROCEDURE zero_bytes( ANYVAR arr : byte_array );
VAR
   i : 0..max_bound;
   limit : 1..max_bound;

BEGIN
   IF (sizeof(arr) > max_bound) THEN
      limit := max_bound
   ELSE
      limit := sizeof(arr);
   FOR i := 1 TO limit DO
      arr[i] := 0;
END;
BEGIN
   zero_bytes( int );
   zero_bytes( rec );
END;
```

READONLY.

This formal parameter mechanism protects the actual parameter from modification within the procedure or function.

A formal READONLY parameter may not be:

* The target of an assignment statement.
* Passed as an argument to a VAR or ANYVAR parameter.
* Passed as an argument to the addr predefined function.
* Passed as an argument to any predefined routine that modifies that argument.

In this way, modification of a variable passed as a READONLY parameter is an error between the call to and return from the procedure or function by modifying the formal parameter itself.

The actual parameter is passed by reference. If the actual parameter is an expression or a constant, then a reference to a copy of the value is passed.

Example

```
PROCEDURE proc( READONLY parm : integer );
VAR
   pint : ^ integer;

   PROCEDURE procx( VAR i : integer );
   external;

BEGIN
   ...
   parm := 0;                 { illegal : cannot assign to a READONLY }
   procx( parm );             { illegal : cannot pass to a VAR parameter }
   pint := addr( parm );      { illegal : cannot take its address }
   ...
END;
```

The above example creates detected errors.

NOTE The mechanism does not detect a modification of a READONLY parameter by another reference parameter or an uplevel reference. The results of such a modification are unpredictable.

Example

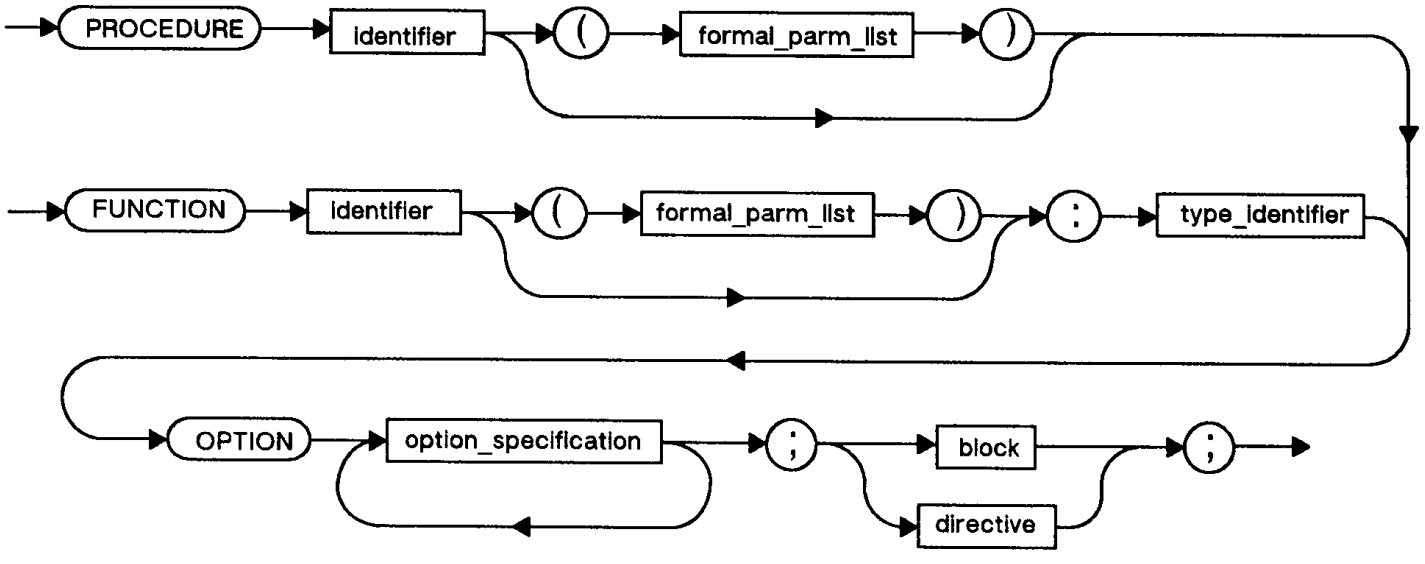
```
PROCEDURE proc1;

VAR
   j : integer;
   PROCEDURE proc2 ( READONLY j : integer
                     VAR m : integer );

   BEGIN

      j := 0;          { modification by an uplevel reference }
      m := 0;          { modification by another reference parameter }
   END;

BEGIN
   proc2 ( j,j );
END;
```

The above example creates undetected errors.

Routine Options

The routine options specify additional attributes of a procedure or function. The routine options follow the parameter list in the declaration of a procedure or function header. $STANDARD_LEVEL 'EXT_MODCAL'$ must be specified when using routine options.

Syntax



The *option-specification* for each option is described in the following pages.

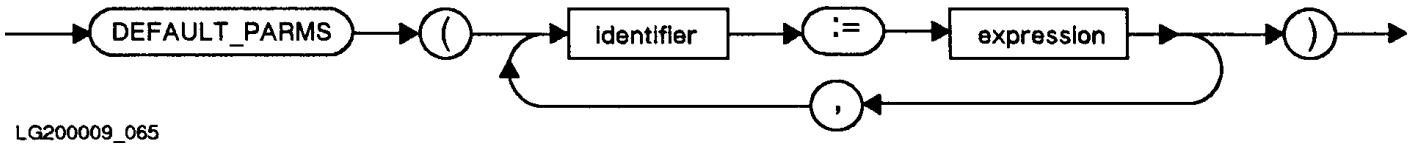
For forward and external declarations of routines, the options specified on the forward or external routine declaration must match the options on the formal declaration.

DEFAULT_PARMS.

Normally, all parameters appearing in a formal parameter list must be present in every corresponding actual parameter list.

The routine option DEFAULT_PARMS allows parameters to be omitted from the actual parameter list. The option specifies which parameters may be omitted, and the default values that the omitted parameters will assume.

Syntax



LG200009_065

The expression supplied in the DEFAULT_PARMS option must be assignment compatible with the corresponding formal parameter type. The expression must also be a constant expression. The only default value permitted for VAR, ANYVAR, and PROCEDURE/FUNCTION parameters is NIL.

Because defaulted reference parameters (VAR, ANYVAR, PROCEDURE/ FUNCTION parameters) cannot be examined (their value is NIL, which cannot be 'dereferenced'), the predefined function haveoptvarparm can be used to determine if a reference parameter was supplied by the caller. See the section "Predefined Routines" for more information.

Example

```
PROCEDURE proc( i : integer )
OPTION DEFAULT_PARMS( i := -1 );
BEGIN
   ...
END;

...
proc( 1 );      { value of parameter is 1 }
proc( );        { value of parameter defaulted to -1 }
proc;           { value of parameter defaulted to -1 }
```

See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more details on OPTION DEFAULT_PARMS.

EXTENSIBLE.

Normally, all parameters appearing in a formal parameter list must be present in a corresponding actual parameter list.

The routine option EXTENSIBLE allows parameters to be omitted from the *end* of an actual parameter list. The option specifies the number of non-extension parameters (those that *must* be supplied) in the actual parameter list. The remaining trailing parameters may be omitted.

Syntax

Note that if a particular extension parameter is supplied in an actual parameter list, then all EXTENSIBLE (and non-defaulted) parameters to the left of the supplied parameter must also be supplied.

It is an error to access a formal parameter whose corresponding actual parameter was not passed. An EXTENSIBLE parameter list, therefore, is always passed with a hidden parameter describing the number of parameters actually passed. The predefined function haveextension can be used to determine if an EXTENSIBLE parameter is present. See the section "Predefined Routines" for more details.

Example

```
PROCEDURE proc( i,j : integer )
OPTION EXTENSIBLE 0;
BEGIN
...
END;

...
proc;              { both parameters not supplied }
proc( 1 );         { second parameter not supplied }
proc( 1,2 );       { both parameters passed }

proc( );           { illegal: implies a defaulted parameter }
proc( ,2 );        { illegal: only trailing parameters can be omitted }
proc( 1, );        { illegal: implies a defaulted second parameter }
```

Refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information on OPTION EXTENSIBLE.

INLINE.

The option INLINE specifies that the code for a procedure or function be expanded in line wherever it is invoked. This expansion removes most procedure call overhead and increases the amount of object code generated. Value parameters work the same with INLINE, that is, an assignment to a value parameter inside an inlined routine does not result in the modification of the actual parameter.

INLINE procedures and functions cannot invoke themselves or any other mutually recursive inline procedures or functions. The body of a procedure or function must be supplied when INLINE is used.

Syntax

Example

```
PROCEDURE proc( x,y : integer ) OPTION INLINE;
...

BEGIN
   ...
END;
```

For more information about INLINE, refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

UNCHECKABLE_ANYVAR.

By default, every ANYVAR parameter is accompanied by a hidden size parameter that indicates the size of the actual parameter. The purpose of this parameter is to allow the routine with the formal ANYVAR parameter to verify that a reference to the formal parameter is within the bounds of the actual parameter by predefined size functions such as sizeof.

Syntax

The option UNCHECKABLE_ANYVAR specifies that the hidden size parameter is not passed by the caller and is not expected by the callee. Its primary use is to interface with non-Pascal procedures and functions that do not support the hidden size parameter.

A routine with the option UNCHECKABLE_ANYVAR must have at least one ANYVAR parameter in its formal parameter list.

Calling the predefined size functions sizeof or bitsizeof for a formal ANYVAR parameter with option UNCHECKABLE_ANYVAR, returns the size of the formal parameter as opposed to the size of the actual parameter.

Example

```
PROCEDURE proc( ANYVAR arr : array_type )
   OPTION UNCHECKABLE_ANYVAR;

BEGIN

END;
```

The routine above can be called from languages that do not support the hidden size parameter because it has been declared with the option UNCHECKABLE_ANYVAR.

UNRESOLVED.

Procedure option UNRESOLVED denotes a procedure or function that is left unresolved by both the linker and the loader. The resolution of the symbolic name to its reference part is delayed until the procedure or function is used.

The suggested way to use this kind of procedure or function is to use the predefined function addr to determine if it can be resolved. NIL is

returned if it cannot. This procedure option can be specified only on level one procedures or functions.

NOTE On implementations that do not support dynamic loading, taking the address of an unresolved routine always produces NIL, while calling an unresolved routine is an error.

A procedure or function declared with option UNRESOLVED must not have a body, and must be declared with the directive EXTERNAL.

Syntax

Example

```
PROCEDURE product_x
   OPTION UNRESOLVED; external;

BEGIN
   ...
   IF (addr( product_x ) <> nil) THEN  {  }
   ...
END;
```

The code above performs a check at run time for the existence of a hypothetical product that provides the entry point product_x. If the product does not exist at run time, and, therefore, does not have any of its entry points installed, then the predefined function addr returns NIL.

Predefined Routines

The system programming extensions define the following additional predefined procedures and functions.

Addressing and Pointers

Addr.

The predefined function addr allows the user to create references to routines or data.

Usage

addr*(variable)*
addr*(variable,offset)*
addr*(routine-name)*

Parameters

variable A variable or reference parameter, or a component of an unpacked structured variable or reference parameter. You can take the address of a component of a *packed* or *crunched* structure, if the component begins on a byte-aligned boundary.

offset A signed integer expression.

routine-name The name of a procedure or function.

Description

The predefined function addr returns a pointer value that is the address of the argument. The type of the pointer returned by addr is assignment compatible with any pointer type. *Addr* returns a *short* or *long pointer* depending on the context in which it is called, the context being the type of the target variable of an assignment, the type of a formal parameter, or the target type of a type coercion.

If the type coercion target type is not a pointer type, addr returns a globalanypointer.

If an integer argument is supplied, the pointer returned is offset by the integer number of bytes from the original variable whose address was taken.

It is illegal to take the address of a formal value or READONLY parameter.

It is illegal to take the address of a component of a PACKED or CRUNCHED structure, if the component does not begin on a byte-aligned boundary.

If addr is called with the name of a procedure or function, the value returned is a reference to that procedure or function. The function result type is assignment compatible with a PROCEDURE or FUNCTION type whose parameter list is congruent with the parameter list of the routine passed to addr.

If the name passed to addr cannot be resolved, the value NIL is returned.

Example

```
$STANDARD_LEVEL 'HP_MODCAL', TYPE_COERCION 'CONVERSION'$
TYPE
   p_to_p_type = ^ p_to_p_type;

VAR
   p_to_p : p_to_p_type;

BEGIN
   p_to_p := addr( p_to_p );
   p_to_p := p_to_p_type( addr( p_to_p^, sizeof( p_to_p^ ) ) )^;
END
```

The first assignment points the pointer p_to_p to itself. The second assignment takes the address of the data referenced by p_to_p (which is itself), offset by the size of the data that p_to_p points to, treats the value at that location as a pointer, and assigns the value pointed to by that pointer back to p_to_p.

Addtopointer.

The predefined function addtopointer allows the user to perform address arithmetic with pointers.

Usage

addtopointer*(pointer, delta)*

Parameters

pointer A pointer expression.

delta A signed integer expression whose range restriction

is implementation dependent.

Description

Addtopointer returns a pointer value that points *delta* bytes away from where the argument *pointer* pointed. The type of the pointer returned by addtopointer is the same as the type of the parameter *pointer*.

The results of an overflow are implementation dependent.

Example

```
TYPE
   intptr = ^integer;

VAR
   ptr1: intptr;
   ptr2: intptr;
   i:    integer;

BEGIN
   ptr2 := addtopointer (ptr1, i);
   ptr1 := addtopointer (ptr1, sizeof(integer));
END
```

Buildpointer.

The predefined function buildpointer allows the user to construct pointer values.

Usage

buildpointer*(space,offset)*

Parameters

space	A space identifier whose range restriction and semantics are implementation dependent.
offset	A bit32 expression whose range restriction is implementation dependent.

Description

buildpointer returns a pointer of type globalanyptr whose value is the address *offset* bytes into *space*.

Example

```
CONST
   Global_Known_Space = 4916;

VAR
   Ptr1 : GlobalAnyPtr;
   Ptr2 : GlobalAnyPtr;
   SID  : Integer;
   Off  : Integer;

BEGIN
Ptr1 := BuildPointer (Global_Known_Space, 0);
off := 4;
Ptr2 := BuildPointer (SID, Off);
END.
```

In the above example, the constant Global_Known_Space represents the value of a known space.

The first use of buildpointer creates a pointer to the location with an

offset of zero in the space whose space id is Global_Known_Space.

The second use of buildpointer creates a pointer to the location four bytes from the beginning of the space whose space has been assigned to the variable SID.

Move Routines

The system programming extensions provide the predefined procedures move_l_to_r, move_r_to_l, fast_fill, and move_fast for generalized and efficient data copying.

Move_L_to_R.

The predefined procedure move_l_to_r provides a generalized array copying mechanism.

Usage

move_l_to_r*(count,source,source_index,target,target_index)*

Parameters

count	A positive integer expression whose value is the number of elements to move.
source	The source array from where elements will be moved.
source_index	An integer expression whose value is the index into the source array of the leftmost element to be moved. The value must be greater than or equal to the index of the first element in the source array, and less than or equal to the index of the last element in the source array minus the move count.
target	The target array to where elements are moved.
target_index	An integer expression whose value is the index into the target array to where the move begins. The value must be greater than or equal to the index of the first element in the target array, and less than or equal to the index of the last element in the target array minus the move count.

Description

The syntax of the procedure is identical to the syntax of the predefined procedure strmove.

move_l_to_r moves elements from left to right. In a left to right move, the first element to be moved is the left-most (lowest indexed) element, and the last element to be moved is the right-most (highest indexed) element.

Even if the elements of the array to be moved are arrays themselves, the array will be moved as a single item.

The following diagram shows the order of copying elements for move_l_to_r.

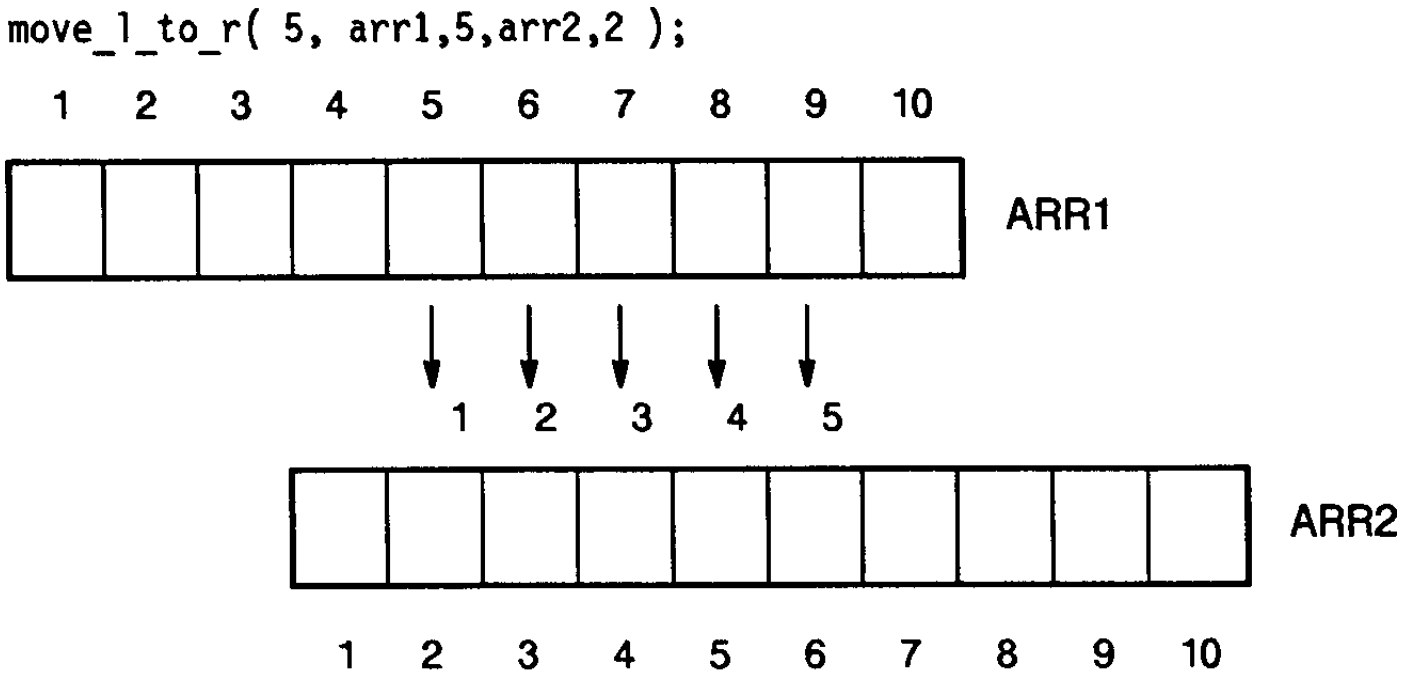


Figure 11-6. Copying Order for move_l_to_r

Example

```
TYPE
   Index_Type_1 =  0..20;
   Index_Type_2 = -3..17;

   Array_Type_1 = PACKED ARRAY [Index_Type_1] of SHORTINT;
   Array_Type_2 = ARRAY [Index_Type_2] of SHORTINT;

VAR
   Array_1 = Array_Type_1;
   Array_2 = Array_Type_2;
   Index   = Integer;

BEGIN

   Move_L_to_R ( 5, Array_1, 3,  Array_2, -3 )

   {  is equivalent to:  }

   FOR Index := 0 TO 4 DO

      Array_2[Index-3] := Array_1[3+Index]

   {  is equivalent to:  }

   Array_2[-3] := Array_1[3]
   Array_2[-2] := Array_1[4]
   Array_2[-1] := Array_1[5]
   Array_2[0]  := Array_1[6]
   Array_2[1]  := Array_1[7]
```

Move_R_to_L.

The predefined procedure move_r_to_l provides a generalized array copying

mechanism.

Usage

move_r_to_l*(count,source,source_index,target,target_index)*

Parameters

count	A positive integer expression whose value is the number of elements to move.
source	The source array from where elements will be moved.
source_index	An integer expression whose value is the index into the source array from where the move will begin. The value must be greater than or equal to the index of the first element in the source array, and less than or equal to the index of the last element in the source array minus the move count.
target	The target array to where elements will be moved.
target_index	An integer expression whose value is the index into the target array to where the move will begin. The value must be greater than or equal to the index of the first element in the target array, and less than or equal to the index of the last element in the target array minus the move count.

Description

The syntax of the procedure is identical to the syntax of the predefined procedure strmove.

move_r_to_l moves the elements from right to left. In a right to left move, the first element to be moved is the right-most (highest indexed) element, and the last element to be moved is the left-most (lowest indexed) element.

Even if the elements of the array to be moved are arrays themselves, the array will be moved as a single item.

The following diagram shows the order of copying for move_r_to_l.

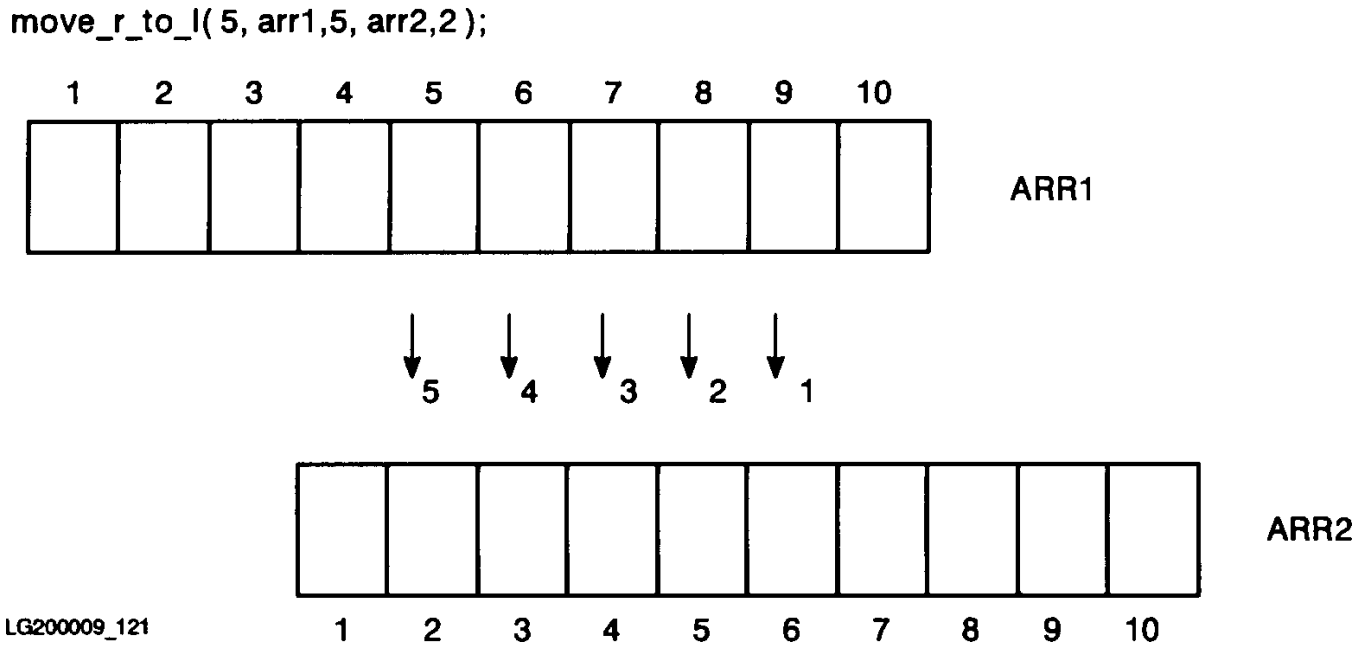


Figure 11-7. Copying Order for move_r_to_l

Fast_Fill.

The predefined procedure fast_fill provides a generalized method of initializing an array to a single 8 bit constant.

Usage

fast_fill *(ptr,fill_char,count)*;

Parameters

ptr	A pointer expression.
fill_char	A constant expression.
count	A positive integer expression that contains the number of bytes to fill with *fill_char*.

Description

fast_fill provides a fast alternative to for loops or assignment statements for initializing each element of a structure or an array to the same 8 bit value.

fill_char and ptr should of compatible types. fill_char must also satisfy the following requirement:

```
0 <= ord(fill_char) <=255
```

Example

```
$standard_level 'ext_modcal'$
program fill;
var p100    : packed array [1..100] of char;
var i1000   : packed array [1..1000] of integer;
type heap_p = array [0..9] of integer;
var p       : ^heap_p;
begin
  fast_fill(addr(p100),' ',sizeof(p100)); {fill a string array}
```

```
                                                {with spaces.}
  fast_fill(addr(i1000),0,sizeof(i1000)); {fill an integer array}
                                                {with 0s.          }
new(p);
fast_fill(p,hex('ff'),sizeof(p^));        {fill an array in the }
                                                {heap to -1 (all bits on).}
end.
```

Move_Fast.

The predefined procedure move_fast provides another generalized array copying mechanism.

Usage

move_fast*(count,source,source_index,target,target_index)*

Parameters

count	A positive integer expression whose value is the number of elements to move.
source	The source array from where elements will be moved.
source_index	An integer expression whose value is the index into the source array of the leftmost element to be moved. The value must be greater than or equal to the index of the first element in the source array, and less than or equal to the index of the last element in the source array minus the move count.
target	The target array to where elements will be moved.
target_index	An integer expression whose value is the index into the target array to where the move begins. The value must be greater than or equal to the index of the first element in the target array, and less than or equal to the index of the last element in the target array minus the move count.

Description

The syntax of the procedure is also identical to the syntax of the predefined procedure, strmove.

Move_fast provides an alternative to move_l_to_r or move_r_to_l for generating simpler and faster code when certain restrictions are met by the parameters.

These restrictions are:

* The source and target arrays must not overlap.
* The source and the target must have elements with the same sizes. The size of each element must be greater than or equal to one byte.
* If the source or the target array is packed, then the packing should be such that the wasted space per word; for example, space left between elements, should be the same for both arrays.
* Both the source and the target arrays must be aligned on byte boundaries. Therefore, one of the following must be true:
    * All elements of the source and the target arrays must each be aligned on byte boundaries.
    * The leftmost source and target element must be aligned on

byte boundaries, and the total size of the elements to be moved must be an integral multiple of one byte.

Example

```
{ This example assumes certain packing which may not apply to your
implementation.  }

     TYPE
        IxType1 =  0..20;
        IxType2 = -3..17;

        Array1 = PACKED ARRAY [IxType1] of SHORTINT;
        Array2 = ARRAY [IxType2] of SHORTINT;
        Array3 = PACKED ARRAY [1..20] of -256..255;
        Array4 = CRUNCHED ARRAY [1..20] of -256..255;

     VAR
        Avar1 : Array1;
        Avar2 : Array2;
        Avar3 : Array3;
        Avar4 : Array4;
        Ix    : Integer;

     BEGIN

        Move_Fast (5, AVar2, -3, AVar1, 3);             { legal  }
        FOR Ix := 0 TO 4 DO                { equivalent FOR loop }
           AVar1[Ix+3] := AVar2[Ix-3];

        Move_Fast (5, AVar3, 2, AVar4, 9);

           { illegal, because                                    }
           { - AVar4 does not have byte-aligned elements         }
           { - AVar4[9] starts on the 27th bit of a word         }
           {      (also not byte-aligned)                        }
           { besides, the number of bits to be moved is not a    }
           { multiple of eight, anyway                           }

        Move_Fast (8, AVar4, 1, AVar4, 9)

           { legal, because                                    }
           { - even though the individual elements of AVar4 are  }
           {      not byte-aligned,                              }
           { - AVar4[1] and AVar4[9] are each byte-aligned, and  }
           { - The total size of the elements to be moved is an  }
           {   integral multiple of eight.                       }
     END;
```

Error Handling Routines

Escape.

Usage

escape*(escape_value)*

Parameters

escape_value An integer expression whose value will be available through the predefined function escapecode.

Description

Calling this predefined procedure indicates that a software error has been detected. Execution passes to the statement following the reserved word RECOVER of the first enclosing TRY-RECOVER statement.

The parameter is evaluated before control is passed and, its value is available to the escapecode function.

If escape is called with no surrounding TRY-RECOVER the program aborts.

Example

```
PROCEDURE proc;
...
BEGIN
   ...
   IF ( {something has gone wrong} ) THEN
      ESCAPE( 0 );
   ...
END;
...
BEGIN
  TRY
     ...
     proc;
     ...
  RECOVER
     WRITELN( 'fatal error. program terminates' );
END.
```

Escapecode.

The predefined function escapecode returns the last execution error number.

Usage

```
escapecode
```

The function returns the value passed to the last implicit or explicit call to the predefined procedure escape.

An explicit call to escape is a call that was made by the user. In this case escapecode returns the value of the escape code passed by the user.

An implicit call to escape is a call that was made by a subsystem on the user's behalf or by the run-time library. In this case, escapecode returns a predefined value based on the type of error detected. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more details about the escape code values.

If escape has never been called (implicitly or explicitly), the value returned by escapecode is undefined. If escapecode is called outside of the recover part of a TRY-RECOVER statement, the value returned is undefined.

Example

```
TRY
   ...
   { perform normal processing }

RECOVER
   CASE escapecode OF
      ...
      { fix-up after an error that can be handled }
      OTHERWISE

         { send errors that cannot be handled }
         escape( escapecode );
   END;
```

The example above shows a possible control structure for trapping software errors. Within the recover section of the TRY-RECOVER statement, escapecode is used to recover information about the nature of the error that caused the trap to the recover section. Note the use of escapecode to pass certain errors on to a next enclosing TRY statement with an explicit call to escape.

Parameter Mechanisms

Haveextension.

The predefined Boolean function haveextension determines if an extension parameter is accessible.

Usage

haveextension*(parameter_name)*

Parameters

parameter_name The name of a formal parameter in the current scope or a containing scope that is EXTENSIBLE.

In a routine with extension parameters it may be necessary to check a formal parameter to ensure that an actual parameter was supplied, as a result of a parameter being passed by the user or being defaulted. The predefined function haveextension indicates, for a formal extension parameter name, whether that parameter exists and can be accessed.

Example

```
PROCEDURE proc_with_opt_parms(     parm1 : type1;
                                VAR parm2 : type2;
                                    parm3 : type3;
                                VAR parm4 : type4  )
   OPTION EXTENSIBLE 2;

BEGIN
   ...
   IF (haveextension( parm4 )) THEN

      { implies that parm4 and parm3 have values }
      ...

   IF (haveextension( parm3 )) THEN

      { implies that parm3 has values }
      ...

   ...
END;

proc_with_opt_parms( var1, var2 );
...
proc_with_opt_parms( var1, var2, var3 );
```

In the previous example, haveextension is used to determine whether either or both of the EXTENSIBLE parameters are passed in the call to the procedure. Note that if parm4 is present, then by definition parm3 must also be present. See the description of routine OPTION EXTENSIBLE for more information.

In the first call to proc_with_opt_parms, both calls to haveextension return false because none of the extension parameters are passed. In the second call, haveextension returns true for the third parameter only.

```
PROCEDURE proc_with_opt_parms(     parm1 : type1;
                                VAR parm2 : type2;
```

```
                                    parm3 : type3;
                                VAR parm4 : type4  )
          OPTION EXTENSIBLE 2
                 DEFAULT_PARMS ( parm3 := 0,
                                 parm4 := nil  );

     BEGIN
        ...
        IF (haveextension( parm3 )) THEN
           ...
        IF (haveextension( parm4 )) THEN
           ...
        ...
     END;
```

In the above example, haveextension(parm4) returns true only if the fourth parameter was actually supplied by the user. If it was not supplied, then the default value is ignored, and the parameter is *not* passed.

Haveextension(parm3) is true if either of the following conditions are true:

* The third or fourth parameters are supplied by the user.

* The fourth parameter is supplied and the third is defaulted. Because the fourth parameter is EXTENSIBLE, and, therefore, by definition all parameters to its left must be passed, the default value for the third parameter is passed even though it was not supplied by the user.

Haveoptvarparm.

The predefined Boolean function haveoptvarparm determines if a default reference parameter is accessible.

Usage

haveoptvarparm*(parameter_name)*

Parameters

parameter_name The name of a default formal parameter of this or a containing scope.

In a routine with default reference parameters, it may be necessary to check a formal parameter to ensure that its actual parameter was supplied by the user. The predefined function haveoptvarparm indicates for a formal reference parameter name whether the corresponding actual parameter was supplied by the user.

The argument to haveoptvarparm must be the name of a formal parameter that:

* Is a VAR, ANYVAR, or PROCEDURE/FUNCTION parameter.

* Specifies a default value of NIL. See the routine OPTION DEFAULT_PARMS.

Example

```
     PROCEDURE proc_with_opt_parms( VAR parm1 : type1;
                                    VAR parm2 : type2;
                                    VAR parm3 : type3  )
              OPTION DEFAULT_PARMS( parm2 := nil,
                                    parm3 := nil  );

     BEGIN
```

```
   ...
   IF (haveoptvarparm( parm2 )) THEN        { ok to use parm2 }
      ...

   IF (haveoptvarparm( parm3 )) THEN        { ok to use parm3 }
      ...
END;
```

The procedure proc_with_opt_parms in the previous example has three VAR parameters, two of which are optional. Before using one of the two parameters within proc_with_opt_parms, a check is made to ensure that the parameters were supplied by the user. This check is accomplished by calling haveoptvarparm with the name of the parameter in question as its argument.

Routine Mechanisms

Call.

The predefined procedure call invokes a procedure.

Usage

call*(procedure_expression)*
call*(procedure_expression,parameter ...)*

Parameters

procedure_expression An expression whose value is a reference to a procedure whose formal parameter list is congruent with the parameters specified in the call.

parameter An actual parameter that is compatible with the corresponding formal parameter of the PROCEDURE type of procedure_expression, that is passed to the invoked procedure.

Description

The predefined procedure call causes the indicated procedure to be called with the indicated parameters.

If, during the execution of the procedure, call accesses any non-local variables, the variables accessed are the variables that were accessible at the time the reference to the procedure was made, when it was passed as an argument to the predefined function, addr. It is an error if the procedure expression has the value NIL or is undefined. It may not be possible to detect an undefined procedure reference.

Example

```
TYPE
   procedure_type = PROCEDURE( i : integer );

VAR
   int : integer;
   proc_var : procedure_type;

PROCEDURE proc( int : integer );
BEGIN
   ...
END;

BEGIN
   proc( int );

   proc_var := addr( proc );
   call( proc_var, int );
```

```
END;
```

In the above example, the two calls to the routine proc are effectively identical.

Fcall.

The predefined function fcall invokes a function.

Usage

```
fcall(function_expression)
fcall( function_expression, parameter ... )
```

Parameters

function_expression An expression whose value is a reference to a function whose formal parameter list is congruent with the parameters specified in the call.

parameter An actual parameter that is compatible with the corresponding formal parameter of the FUNCTION type of *function_expression*, that is passed to the invoked function.

The predefined function fcall causes the function referenced by the first FUNCTION variable parameter to be invoked with the supplied parameters.

The type returned by fcall is the same as the type returned by the FUNCTION expression.

See the description of call for more information.

Size Functions

Bitsizeof.

The predefined function bitsizeof returns an integer representing the size of its argument in bits.

Usage

```
bitsizeof(variable)
bitsizeof(record_variable,tag_value ...)
bitsizeof(type_name)
bitsizeof(record_type_name,tag_value ...)
bitsizeof(struc_constant)
bitsizeof(string_literal)
```

Parameters

variable The name of a variable.

record_variable The name of a record variable with variants.

tag_value The name of a case constant in the variant part of a record declaration. Case constants for nested variants may appear separated by commas.

type_name The name of a type.

record_type_name The name of a record type with variants.

struc_constant The name of an array, record, set, or string constructor.

string_literal A string literal.

The bitsizeof function returns the number of bits needed to represent the data value part of a data item of the given type, or the actual allocated size of a variable. If the first parameter is a record type or variable with variants, a variant may be selected by specifying a case constant with the subsequent parameters. Otherwise, the size with the largest variant is used.

bitsizeof*(type)* returns the minimum number of bits of storage for the type, and bitsizeof*(variable)* returns the number of bits of storage for the variable.

For an ANYVAR parameter, two cases exist: If an additional hidden size parameter is passed along with the ANYVAR parameter, bitsizeof gives the number of bits in the number of bytes allocated to represent the actual parameter. If the hidden length parameter is not passed, bitsizeof gives the number of bits required to represent the formal parameter as a given type.

Example

```
TYPE
   int_type = integer;
   rec_type = RECORD
      int : integer;
      CASE flag: Boolean OF
        true: ( r  : real );
        false:( lr : longreal );
        end;
VAR
   int  : int_type;
   rec  : rec_type;
   size : integer;

BEGIN
   ...
   size := bitsizeof( int );
   size := bitsizeof( int_type );

   size := bitsizeof( rec, true );
   ...
END;
```

NOTE bitsizeof is allowed in CONST declarations except for ANYVAR, VAR string, and conformant array parameters.

Sizeof.

The predefined function sizeof returns an integer representing the size of its argument in bytes.

Usage

```
sizeof(variable)
sizeof(record_variable,tag_value ...)
sizeof(type_name)
sizeof(record_type_name,tag_value ...)
sizeof(struct_constant)
sizeof(string_literal)
```

Parameters

variable The name of a variable.

record_variable The name of a record variable with variants.

tag_value The name of a case constant in the variant part of a record declaration. Case constants for nested variants may appear separated by commas.

type_name The name of a type.

record_type_name The name of a record type with variants.

struct_constant The name of an array, record, set, or string constructor.

string_literal A string literal.

The predefined function sizeof returns the number of bytes of storage required to represent the data value part of a data item of the given type, or the actual allocated size of a variable. If the first parameter is a record type or variable with variants, a variant may be selected by specifying a case constant with the subsequent parameters. sizeof*(type)* returns the minimum number of bytes for the type. sizeof*(variable)* returns the number of bytes of storage for the variable. Otherwise, the size of the largest variant is returned.

For a variable of a simple data type, the number returned by sizeof is equivalent to the storage required for the variable in the unpacked context. For example, if the variable is type char or Boolean, sizeof returns 1.

For an ANYVAR parameter, two cases exist: If an additional hidden size parameter is passed along with the ANYVAR parameter, sizeof gives the actual number of bytes allocated to represent the actual parameter. If the hidden length parameter is not passed, sizeof gives the number of bytes required to represent the formal parameter.

For conformant array parameters, the function sizeof is the actual size of the parameter.

Example

```
TYPE
   byte = 0..255;
   big_record = RECORD CASE Boolean OF
      true:  ( arr : array [ 1..200 ] of byte ):
      false: ( f1  : integer;
                 ...
               f99 : char  );

BEGIN
   ...
   IF (sizeof(big_record,true) <> sizeof(big_record,false)) THEN
       BEGIN
          writeln ( 'variant size mismatch by',
                 abs(sizeof(big_record,true)-sizeof(big_record,false)):1,
                'bytes' );
          HALT (1);
       END;
   ...
END.
```

NOTE sizeof is allowed in CONST sections except for ANYVAR, VAR s, and conformant array parameters.

Chapter 12 Compiler Options

Introduction

This chapter explains every HP Pascal compiler option. Compiler options fall into two categories: system-independent and system-dependent. System-independent options work the same way whether HP Pascal is running on the MPE/iX operating system or the HP-UX operating system. System-dependent options either work on only one operating system, or they work differently on HP-UX and MPE/iX. The following table categorizes the compiler options.

System-Independent Options		System-Dependent Options
ALIAS	MLIBRARY	CONVERT_MPE_NAMES
ALIGNMENT	NOTES	CALL_PRIVILEGE
ANSI	OPTIMIZE	EXEC_PRIVILEGE
ARG_RELOCATION	OS	FONT
ASSERT_HALT	OVFLCHECK	GPROF
ASSUME	PAGE	HP3000_16[REV BEG]
BUILDINT	PAGEWIDTH	HP3000_32
CHECK_ACTUAL_PARM	PARTIAL_EVAL	HP_DESTINATION
CHECK_FORMAL_PARM	POP	INCLUDE[REV END]
CODE	PUSH	INCLUDE_SEARCH
CODE_OFFSETS	RANGE	NLS_SOURCE
COPYRIGHT	S300_EXTNAMES	RLFILE
COPYRIGHT_DATE	SEARCH	RLINIT
ELSE	SET	SHLIB_CODE
ENDIF	SKIP_TEXT	SHLIB_VERSION
EXTERNAL	SPLINTR	SYMDEBUG
EXTNADDR	STANDARD_LEVEL	
GLOBAL	STATEMENT_NUMBER	
HEAP_COMPACT	STDPASCAL_WARN	
HEAP_DISPOSE	STRINGTEMPLIMIT	
IF	SUBPROGRAM	
INLINE	SYSINTR	
INTR_NAME	SYSPROG	
KEEPASMB	TABLES	
LINES	TITLE	
LIST	TYPE_COERCION	
LIST_CODE	UPPERCASE	
LISTINTR	VERSION	
LITERAL_ALIAS	VOLATILE	
LOCALITY	WARN	
LONG_CALLS	WIDTH	
MAPINFO	XREF	

Each compiler option entry in this chapter gives the option's default value (if any) and location. Table 12-1 defines the terms that describe option location (in terms of both option location and scope).

Table 12-1. Compiler Option Locations and Scopes

Location Term	Option Location	Option Scope
Anywhere.	Anywhere in the program.	Depends upon the option.
At front.	Before PROGRAM or MODULE in the source file.	Applies to the entire source file.
Not in body.	Not between BEGIN and END. (preferably immediately before BEGIN or the procedure heading).	Applies to the routine that contains it.
Statement.	Anywhere in the program.	Applies to the statements following it.
Heading.	In a routine heading, after PROCEDURE or FUNCTION, but before the body or directive.	Applies to the routine that contains it.

A compiler option list begins with a dollar sign ($), contains one or more compiler options, and ends with a dollar sign. It must fit on a single line.

Syntax

```
$ option  [{,} option ]...$
        [{;}       ]
```

Parameter

option Any compiler option described in this chapter; however, options with incompatible locations cannot appear in the same list.

Example

```
$LIST OFF$
$ANSI OFF, LIST ON$
$PARTIAL_EVAL ON, ASSUME 'PASCAL_FEATURES', LINES 50$
```

NOTE Unrecognized compiler options do not cause compilation errors.

System-Independent Options

System-independent options work the same way whether HP Pascal is running on the MPE/iX operating system or the HP-UX operating system. These options fall into the following three categories:

Category	Associated With
HP Standard options	HP Standard Pascal
HP Pascal options	HP Pascal
System programming options	System programming extensions

Figure 12-1 shows the relationship between ANSI Standard Pascal and

HP Pascal (with and without system programming extensions).

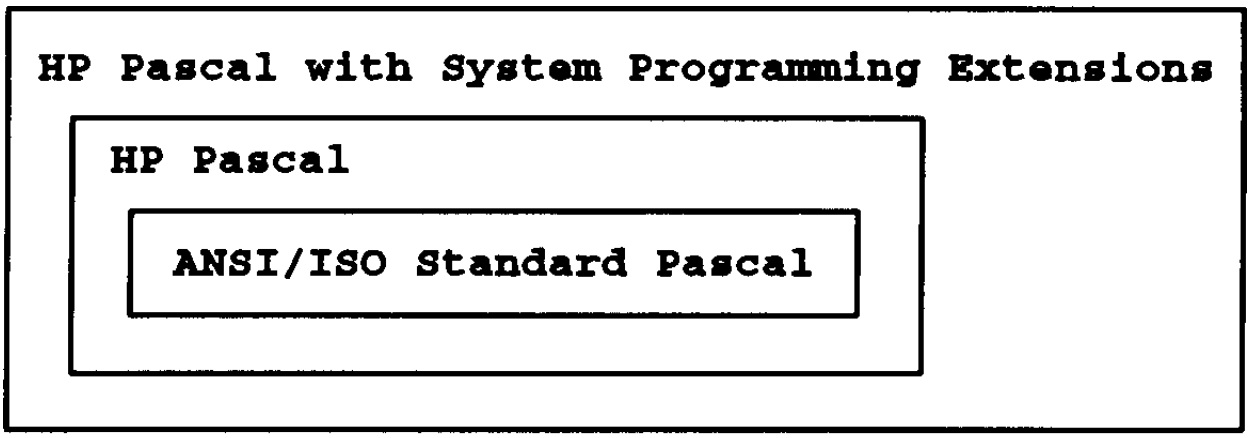


Figure 12-1. Relationship Between HP Pascal and ANSI Standard Pascal

The following table categorizes the system-independent compiler options.

HP Pascal Options

ALIAS
ALIGNMENT
ARG_RELOCATION
ASSERT_HALT
ASSUME
BUILDINT
CHECK_ACTUAL_PARM
CHECK_FORMAL_PARM
CODE
CODE_OFFSETS
COPYRIGHT
COPYRIGHT_DATE
ELSE
ENDIF
EXTERNAL
EXTNADDR
GLOBAL
HEAP_COMPACT
HEAP_DISPOSE
IF
INLINE
INTR_NAME
KEEPASMB
LIST_CODE
LISTINTR
LITERAL_ALIAS
LOCALITY
LONG_CALLS
MAPINFO
MLIBRARY
NOTES
OPTIMIZE
OS
OVFLCHECK
PAGEWIDTH
POP
PUSH
S300_EXTNAMES
SEARCH
SET
SKIP_TEXT
SPLINTR
STATEMENT_NUMBER
STDPASCAL_WARN
STRINGTEMPLIMIT
SUBPROGRAM
SYSINTR
SYSPROG
TABLES
TITLE
TYPE_COERCION
UPPERCASE
VERSION
VOLATILE
WARN
WIDTH
XREF

HP Standard Options

ANSI
LINES
LIST
PAGE
PARTIAL_EVAL
RANGE
STANDARD_LEVEL

System Programming Options

EXTNADDR
TYPE_COERCION

NOTE File name parameters have different syntax on the HP-UX and MPE/iX operating systems. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*.

HP Standard Options

HP Standard compiler options are available on all versions of Pascal that run on HP computers. They are part of the HP Standard. An HP Pascal program containing only HP Standard options can be compiled by any Pascal compiler that runs on an HP computer.

HP Pascal Options

HP Pascal compiler options are not required by the HP Standard, but are available in HP Pascal. An HP Pascal program containing HP Pascal options must be compiled by the HP Pascal compiler.

System Programming Options

System programming compiler options are only available if the compiler specifies $STANDARD_LEVEL 'EXT_MODCAL'$ or $STANDARD_LEVEL 'HP_MODCAL'$ (see "STANDARD_LEVEL" compiler option for more information).

System-Dependent Options

System-dependent options either work on only one operating system, or they work differently on HP-UX and MPE/iX. Figure 12-2 diagrams the three categories of system-dependent options.

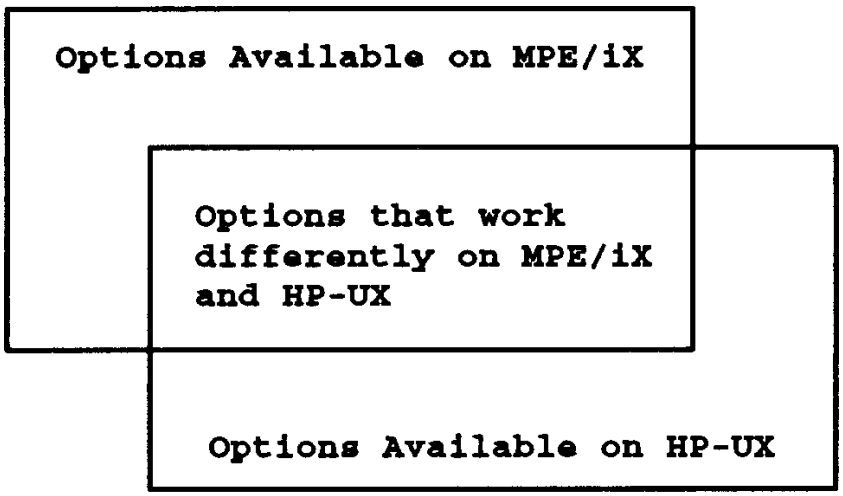


Figure 12-2. Categories of System-Dependent Compiler Options

The system-dependent options are:

MPE/iX Only	MPE/iX and HP-UX	HP-UX Only
CALL_PRIVILEGE	INCLUDE	CONVERT_MPE_NAMES
EXEC_PRIVILEGE	INCLUDE_SEARCH	GPROF
FONT	NLS_SOURCE	HP_DESTINATION 'ARCHITECTURE
HP3000_16	SYMDEBUG	HP_DESTINATION 'SCHEDULER
HP3000_32		SHLIB_CODE
RLFILE		SHLIB_VERSION
RLINIT		

MPE/iX Options

MPE/iX compiler options are available only in HP Pascal running on the MPE/iX operating system.[REV BEG] See the $OS compiler option later in this chapter.[REV END]

HP-UX Options

HP-UX compiler options are available only in HP Pascal running on the HP-UX operating system.[REV BEG] See the $OS compiler option later in this chapter.[REV END]

Options That Work Differently on HP-UX and MPE/iX

The compiler options that this section explains are available in HP Pascal running on either the MPE/iX or HP-UX operating system; however, the options work differently on the two systems. A program that contains these options can be compiled by a program that specifies either $OS 'MPEXL'$ or $OS 'HPUX'$.

System-wide File

The compiler looks for a system-wide file called PASCNTL.PUB.SYS on MPE/iX or /usr/lib/pasopts on HP-UX. If the file exists and is not empty, the compiler opens and reads the file. The file should[REV BEG] contain only compiler options[REV END] and comments. If there is anything else in the file, the compiler emits an error message.

On MPE/iX the message is:
[REV BEG]

```
     ONLY COMMENTS AND COMPILER OPTIONS ARE ALLOWED IN ‘PASCNTL.PUB.SYS' (045)
```

[REV END]

On HP-UX the message is:
[REV BEG]

```
     ONLY COMMENTS AND COMPILER OPTIONS ARE ALLOWED IN /usr/lib/pasopts (045)
```

[REV END]

The file is shipped empty and does not need to contain anything. If the file is empty, the compiler does not attempt to open it.

However, if compiler[REV BEG] options have been added to the file, the compiler processes these options[REV END] before anything else, even the info string. Therefore, you can override the[REV BEG] options in the file because later options take[REV END] precedence over earlier options.

Compiler Option Description

This section contains the descriptions of each of the HP PASCAL compiler options. Each description contains the syntax, location, and default value of the option. They are arranged alphabetically.

ALIAS

ALIAS is an **HP Pascal** Option.

The ALIAS compiler option specifies an external name for a procedure, function, or global variable.

Syntax

     $ALIAS *string* $

Parameter

string The external name. The compiler does not distinguish

between uppercase and lowercase letters. By default, the external name is downshifted. The LITERAL_ALIAS compiler option allows the external name to remain as it is. The UPPERCASE compiler option upshifts the external name.

Default The internal name, downshifted (or upshifted if UPPERCASE is ON).

Location

Routine:	Heading.	
Global variable:	Immediately after the variable name in the variable declaration.	

When a routine has both internal and external names, the program recognizes its internal name and the operating system recognizes its external name.

NOTE For global variables, the HP Pascal options EXTERNAL or GLOBAL must be included or the ALIAS option is ignored.

Also, routines must be level 1 or the ALIAS option is ignored.

The reasons to use the ALIAS option are:

* To define multiple internal names for a single external procedure.

* To access a library or system routine that has an illegal (external) name, by giving it a legal internal name.

Example 1

```
$GLOBAL$
PROGRAM p (input,output);

VAR
   global_var $ALIAS 'gvar'$ : integer;             {global variable}

PROCEDURE $ALIAS 'write'$ Writefile; EXTERNAL;      {procedure}

FUNCTION $ALIAS 'read'$ Readfile : char; EXTERNAL;  {function}

BEGIN
   .
   .
   .
END.
```

Example 2

```
PROGRAM show_alias;
  .
  .
PROCEDURE $ALIAS 'intrinname'$ A; INTRINSIC;  {One intrinsic   }
PROCEDURE $ALIAS 'intrinname'$ B; INTRINSIC;  {has two internal}
                                              {names, A and B  }
  .
  .
PROCEDURE $ALIAS 'x''x'$ xx; INTRINSIC;  {The intrinsic name}
                                         {x'x is illegal in Pascal}
  .
  .
PROCEDURE proc1;
   FUNCTION $ALIAS 'D1'$ do_it (n : INTEGER): BOOLEAN;
   BEGIN {do_it}
```

```
        .
        .
     END; {do_it}
  BEGIN {proc1}
    .
    .
  END; {proc1}

  PROCEDURE proc2;
     FUNCTION $ALIAS 'D2'$ do_it (a,b : INTEGER): INTEGER;
     BEGIN {do_it}
       .
       .
     END; {do_it}
  BEGIN {proc2}
    .
    .
  END; {proc2}

  BEGIN {show_alias}
    .
    .
  END. {show_alias}
```

Example 3

```
  PROGRAM show_alias;

  FUNCTION $ALIAS 'f'$ f1 (p1 : integer); EXTERNAL;

  FUNCTION $ALIAS 'f'$ f2 (p1,p2 : integer); EXTERNAL;

  BEGIN
     .
     .
     .
  END.
```

Notice that the function f1 declares one parameter of the function f, while the function f2 declares two.

ALIGNMENT

ALIGNMENT is an **HP Pascal** Option.

The ALIGNMENT compiler option specifies the alignment requirements for a type (for the definition of *alignment* see Chapter 5). It cannot be used with string or file types. The alignment of a record or array must be at least as large as its largest field or element.

ALIGNMENT does not support alignments greater than 8 bytes for variables. Only fields are aligned greater than 8 bytes. However, you can align a record or array with more than 8 bytes through a call to P_GETHEAP with the appropriate alignment parameter.

Syntax

```
  $ALIGNMENT integer $
```

Parameter

integer In the range 1..2048. The following values for *integer* specify the alignments indicated. Other values are illegal.

Value	Alignment
1	Byte aligned
2	Half-word aligned
4	Word aligned
8	Double-word aligned
16	Cache aligned
32	Cache aligned
64	Cache aligned
2048	Page Aligned

Default Depends upon packing algorithm.

Location After the symbol = in a type definition.

Example

```
TYPE
   Rec = $ALIGNMENT 16$
         RECORD
         F1 : Integer;
         F2 : ShortInt;
         F3 : Real;
         END;

   Integer_ = $ALIGNMENT 2$ Integer;
   Ptr = ^Integer_;
```

ANSI

ANSI is an **HP Standard** Option.

When the ANSI compiler option is ON, the compiler issues an error whenever it encounters a feature in the source code that is illegal in ANSI Standard Pascal. The compiler compiles the illegal feature if possible; otherwise it is a syntax error. The error appears in the listing.

The command line option -A also specifies this option.

Syntax

```
$ANSI {ON }$
      {OFF}
```

Default OFF

Location Anywhere.

The options $ANSI ON$ and $STANDARD_LEVEL 'ANSI'$ are equivalent.

Example

```
PAGE   1 HEWLETT-PACKARD  ...  (C) HEWLETT-PACKARD CO. 1986 ...
    0    1.000   0    $ANSI ON, OS 'MPEXL'$
    0    2.000   0    PROGRAM t;
    0    3.000   0
```

```
     0    4.000   0    BEGIN
     0    5.000   0       assert(false,0);
     0    6.000   0          ^

**** ERROR #  1 THIS FEATURE REQUIRES $STANDARD_LEVEL "HP_PASCAL" (539)
     1    7.000   0    END.
```

ARG_RELOCATION

ARG_RELOCATION is an **HP Pascal** Option.

The ARG_RELOCATION option can be used to suppress parameter relocation information for all procedure or function definitions and calls. This option is only useful for REAL and LONGREAL data types.

Syntax

```
$ARG_RELOCATION {ON }$
                {OFF}
```

Parameters

ON — Relocation information is generated for parameters and function returns. For dynamic calls (FCALL, CALL and calls to procedural and functional parameters), REAL and LONGREAL parameters and function results are put into or assumed to be in general registers.

OFF — Relocation information is suppressed. Additionally, for dynamic calls, REAL and LONGREAL parameters and function returns are put into or assumed to be in floating point registers.

Default ON

Location At front.

Parameter relocation information is used by the linker to make sure the arguments and the function return are in the correct register type. (General versus floating point.)

ARG_RELOCATION OFF might be useful for performance if the called procedure is in a shared library (HP-UX) or executable library (MPE/iX), or if dynamic calls are used with REAL or LONGREAL parameters or function returns.

When ARG_RELOCATION ON is used, linker-supplied stubs copy floating point registers to general registers and then back again in the library. The same thing is done for the function return.

When ARG_RELOCATION OFF is used, the linker assumes that everything is in the correct register and generates no extra stubs. For a dynamic call, the compiler puts floating point parameters in floating point registers.

See *Procedure Calling Conventions Reference Manual* for more details on parameter relocation stubs.

Example

```
$ARG_RELOCATION OFF$
program args;
procedure p_r(x : real); external;
begin
  p_r(1.5);
end.
```

CAUTION If the ARG_RELOCATION is used improperly, unexpected results may

occur. Refer to the *Procedure Calling Conventions Reference Manual* for detailed information on how to use this option.

ASSERT_HALT

ASSERT_HALT is an **HP Pascal** Option.

The ASSERT_HALT compiler option works with the predefined function *assert*. IF an *assert* fails (that is, if its Boolean expression is false), and ASSERT_HALT is ON, the program terminates. If ASSERT_HALT is OFF, the program continues to execute. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*.

Syntax

```
$ASSERT_HALT {ON }$
             {OFF}
```

Default OFF

Location Anywhere.

Example

This program stops if i is greater than 10.

```
     $ASSERT_HALT ON$
     $OS 'MPE/iX'$
     PROGRAM show_asserthalt (input,output);
     VAR
        i: integer;
     BEGIN
        write('Please enter an integer: ');
        prompt;
        read(i);
        assert(i<10,99);
        writeln('Good show! You didn''t abort the program.');
     END.
```

ASSUME

ASSUME is an **HP Pascal** Option.

The ASSUME compiler option specifies optimizer assumptions beyond those implied by the STANDARD_LEVEL compiler option. It determines what the optimizer does, but it does not determine what the compiler accepts. If your program violates its optimizer assumptions, you can compile it with or without optimization; however, the optimized version may fail.

Syntax

```
          {NOTHING             }
          {PASCAL_FEATURES     }
          {PASCAL_POINTERS     }
          {NO_PARM_ADDRESSED   }
          {PARM_TYPES_MATCH    }
          {NO_PARMS_OVERLAP    }
$ASSUME '{LOCAL_GOTOS_ONLY    }'$
          {LOCAL_ACCESSES_ONLY}
          {NO_SIDE_EFFECTS     }
          {NO_HEAP_CHANGES     }
          {NORMAL_RETURN       }
          {LOCAL_ESCAPES_ONLY  }
          {FLOAT_TRAPS_ON      }
```

Parameters

NOTHING	The optimizer assumes nothing, overriding any previous assumptions.
PASCAL_FEATURES	The optimizer assumes that routines are defined and called with Pascal features only. PASCAL_FEATURES implies PASCAL_POINTERS, PARM_TYPES_MATCH, NO_PARM_ADDRESSED, and LOCAL_ESCAPES_ONLY.
PASCAL_POINTERS	The optimizer assumes that no operation except the function *new* creates a pointer, and no operation except an assignment statement modifies its value. This precludes the functions *addr*, *addtopointer*, and *buildpointer*, type coercing to a pointer type, and reference parameters and function return values that violate the assumption. PASCAL_POINTERS implies NO_PARM_ADDRESSED.
NO_PARM_ADDRESSED	The optimizer assumes that no reference parameter is passed to the function *addr*.
PARM_TYPES_MATCH	The optimizer assumes that every formal reference parameter and its corresponding actual parameter are of the same type; that is, no actual parameter is type-coerced (except in the case of ANYVAR parameters).
NO_PARMS_OVERLAP	The optimizer assumes that the actual parameters passed to the formal reference parameters do not overlap; that is, two formal parameters do not get the same actual parameter or the same field of a record. (This is always true if the scope defines only one reference parameter.) NO_PARMS_OVERLAP has no effect without LOCAL_ACCESSES_ONLY.
LOCAL_GOTOS_ONLY	The optimizer assumes that no routine jumps to a label in a surrounding scope.
LOCAL_ACCESSES_ONLY	The optimizer assumes that only parameters and local variables are accessed or modified (directly or indirectly). Input, output, and global and nonlocal variables are not accessed or modified. LOCAL_ACCESSES_ONLY implies NO_SIDE_EFFECTS.
NO_SIDE_EFFECTS	The optimizer assumes that only parameters and local variables are modified (directly or indirectly). Input, output, and global and nonlocal variables are not modified (but they can be accessed). NO_SIDE_EFFECTS implies NO_HEAP_CHANGES.
NO_HEAP_CHANGES	The optimizer assumes that no item currently on the heap is modified (but it can be accessed).
NORMAL_RETURN	The optimizer assumes that routines are exited only in the normal way. NORMAL_RETURN implies LOCAL_GOTOS_ONLY and LOCAL_ESCAPES_ONLY.
LOCAL_ESCAPES_ONLY	The optimizer assumes that no routine escapes to a calling routine; that is, all calls to the predefined procedure *escape* are within TRY-RECOVER constructs.
FLOAT_TRAPS_ON	The optimizer assumes that the IEEE floating point traps are on and does not move loop

invariant expressions (that are conditioned by an IF) out of loops. This parameter can be used in conjunction with any of the other parameters. Refer to the +FP compiler option in the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, as well as the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*.

Default NOTHING (assuming that FLOAT_TRAPS_ON is not specified).

Location Anywhere, but in order to be effective, it must appear before the place in the code where label declarations or directives can appear. If FLOAT_TRAPS_ON is specified, the location must be at the front.

Scope All following source code, until overridden by another ASSUME option.

Figure 12-3 shows how the parameters of the ASSUME compiler option are related.

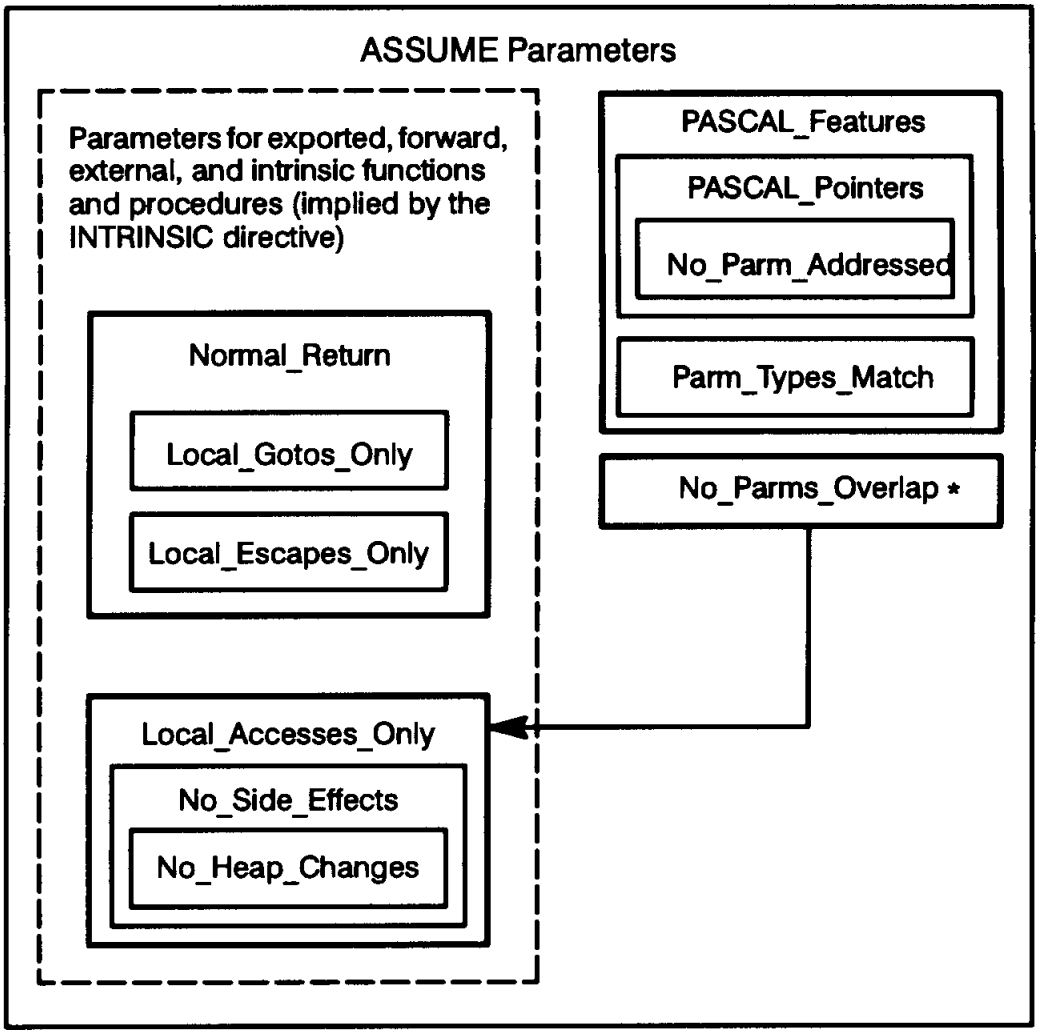


Figure 12-3. Relationships Between ASSUME Compiler Option Parameters

* NO_PARMS_OVERLAP is ineffective without LOCAL_ACCESSES.

After compiling a routine, the compiler knows what it accesses and modifies, so the optimizer can derive the appropriate assumptions. Only exported, forward, and external routines require that you specify LOCAL_GOTOS_ONLY, LOCAL_ACCESSES_ONLY, NO_SIDE_EFFECTS, or NO_HEAP_CHANGES. These assumptions are valid for intrinsic functions and procedures, but you must specify them in the routine.

Example 1

The following program skeleton demonstrates how to nest ASSUME options, using the PUSH and POP compiler options.

```
     $ASSUME 'PASCAL_FEATURES'$
     PROGRAM prog ;
     LABEL
        999 ;  { Possible target for nonlocal GOTO }

     $PUSH$
     $ASSUME 'NO_SIDE_EFFECTS'$
     PROCEDURE extnl ; EXTERNAL ;
          { Optimizer assumes:
          {      PASCAL_FEATURES (inherited)
          {      NO_SIDE_EFFECTS (specified)
          }
     $POP$

     $PUSH$
     $ASSUME 'LOCAL_ACCESSES'$
     $ASSUME 'LOCAL_GOTOS_ONLY'$
     PROCEDURE intnl ;

        $PUSH$
        $ASSUME 'NOTHING'$
             { Optimizer assumes nothing.
             { This overrides inherited assumptions.
             }
        $ASSUME 'PARM_TYPES_MATCH'$
        PROCEDURE nested ;
        VAR
           i : integer ;

           $PUSH$
           $ASSUME 'NO_SIDE_EFFECTS'$
           $ASSUME 'NO_PARMS_OVERLAP'$
           PROCEDURE furthernested ;
```

(Example is continued on next page)

```
           BEGIN {furthernested}

                { Modifying i violates NO_SIDE_EFFECTS }

                { Optimizer assumes:
                {      PARM_TYPES_MATCH (inherited),
                {      NO_SIDE_EFFECTS (specified)
                {      NO_PARMS_OVERLAP (specified)
                {      LOCAL_GOTOS_ONLY (known after compilation)
                }
           END ; {furthernested}
           $POP$

        BEGIN {nested}
             { Optimizer assumes:
             {      PARM_TYPES_MATCH (specified)
             {      LOCAL_GOTOS_ONLY (known after compilation)
             }

        furthernested ;
```

```
   END ; {nested}
   $POP$

BEGIN {intnl}
     { Optimizer assumes:
     {      PASCAL_POINTERS (inherited)
     {      PARM_TYPES_MATCH (inherited)
     {      LOCAL_GOTOS_ONLY (specified)
     {      NO_SIDE_EFFECTS (known after compilation)
     }

nested ;

END ; {intnl}
$POP$

BEGIN { main program }
     { Optimizer assumes:
     {      PASCAL_POINTERS (implied)
     {      PARM_TYPES_MATCH (implied)
     }

intnl ;

999:
END .
```

Example 2

The following example turns on the IEEE floating-point traps. (On HP-UX, the +FPZ option can be used instead of the call to HPENBLTRAP). This program would have aborted on the divide by 0 if the loop invariant expression was moved out of the loop.

```
$ASSUME 'FLOAT_TRAPS_ON'$ $OPTIMIZE ON$
program trap;
var
   r,s : real;
   i : integer;
   oldmask : integer;

procedure hpenbltrap; intrinsic;

begin
   hpenbltrap(hex('ffffffff'),oldmask);
   s := 0.0;
   r := 0.0;
   for i := 0 to 10 do
      begin
      if r <> 0.0 then
         s := 1.0 / r;     { divide by zero? }
      s := s + 1.0;
      end;
end.
```

See the *Pascal/iX Programmer's Guide* or the *Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information on +FP. See the *Trap Handling Programmer's Guide* for more information on HPENBLTRAP. See the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* for more information on IEEE floating point instructions and traps.

BUILDINT

BUILDINT is an **HP Pascal** Option.

The BUILDINT compiler option causes the compiler to build an intrinsic file.

Syntax

```
$BUILDINT [string ]$
```

Parameter

string Specifies the name of the intrinsic file that the compiler builds. If the specified file exists and is an intrinsic file, entries are added to it. If it exists, but is not an intrinsic file, it is an error. If the file does not exist, it is created (see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*).

Default System intrinsic file.

Location At front.

The compiler adds an entry to the intrinsic file for each routine declaration in the compilation unit. If the compilation unit declares a routine with the same name as a routine that is already in the intrinsic file, the new routine declaration replaces the old one.

The compilation unit can contain constant, type, and variable declarations and procedure and function headings, but not routine bodies or a nonempty outer block. Each routine must be designated external (with the EXTERNAL directive). The compiler does not generate code for the compilation unit.

NOTE The *pc* option +C on HP-UX affects the BUILDINT compiler option (see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*).

Example

```
$BUILDINT 'MYINTR'$
PROGRAM Show_Buildint;

TYPE
   Smallint = -32768..32767;
   ByteArray = PACKED ARRAY [1..80] OF CHAR;
   RecType = RECORD
                F1 : Integer;
                F2 : ByteArray;
             END;

PROCEDURE Proc1 (    P1 : Smallint;
                     P2 : Integer;
                 VAR P3 : ByteArray;
                 VAR P4 : RecType;
                     P5 : Real
                 );
EXTERNAL;

FUNCTION Func1 (P1 : Real) : Integer;
EXTERNAL;

BEGIN
{Empty outer block}
END.
```

The BUILDINT compiler option is used with the LISTINTR and SYSINTR compiler options. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for details.

CALL_PRIVILEGE and EXEC_PRIVILEGE

CALL_PRIVILEGE and EXEC_PRIVILEGE are **System-Dependent MPE/iX** Options.

The CALL_PRIVILEGE and EXEC_PRIVILEGE compiler options allow routines to call and execute privileged mode routines. To use these compiler options, the option STANDARD_LEVEL 'EXT_MODCAL' is required.

The CALL_PRIVILEGE option specifies, for a given routine, the minimum privilege level that other routines must have to call the specified routine. The EXEC_PRIVILEGE option specifies the privilege level at which a routine will execute.

CAUTION Routines not specified by the CALL_PRIVILEGE or EXEC_PRIVILEGE compiler options are given the lowest privilege level by default. If you specify a routine to have a higher calling or executing privilege level, the routine is allowed to override safety features in the MPE/iX operating system. Therefore, exercise caution when using CALL_PRIVILEGE and EXEC_PRIVILEGE because misuse can destroy your operating system.

Syntax

$CALL_PRIVILEGE *integer* $

$EXEC_PRIVILEGE *integer* $

Parameter

integer An integer in the range 0 . . 3, with 0 being the most privileged level and 3 the least.

Default Privilege level 3.

Location Before the body of the routine, but after the reserved words PROCEDURE or FUNCTION.

Example

```
$STANDARD_LEVEL 'EXT_MODCAL'$
PROGRAM p;

PROCEDURE proc1 $CALL_PRIVILEGE 1$ (
   VAR i : integer);
BEGIN
END;

PROCEDURE proc2 $EXEC_PRIVILEGE 2$ (
   VAR i : integer);
BEGIN
END;

PROCEDURE proc3 $CALL_PRIVILEGE 1$
                $EXEC_PRIVILEGE 0$ (
   VAR i: integer);
BEGIN
END;

BEGIN
END.
```

Any routine calling procedure proc1 must execute at privilege level 1 or level 0. By default, proc1 executes at privilege level 3. Procedure proc2 executes at level 2; a routine calling proc2 may be executing at

any level. Procedure proc3 executes at privilege level 0; any routine calling proc3 must be executing at level 1 or higher.

CHECK_ACTUAL_PARM

CHECK_ACTUAL_PARM is an **HP Pascal** Option.

The CHECK_ACTUAL_PARM compiler option determines how closely the actual parameters of routines must match their formal parameters in separately compiled sources. If the actual and formal parameters are incompatible, the linker does not link the program.

Syntax

$CHECK_ACTUAL_PARM *integer* $

Parameter

integer In the range 0..3. Determines how the linker checks actual parameters against formal parameters, as follows:

Value	**The linker checks:**
0	Nothing.
1	Function result type.
2	Function result type, number of routine parameters.
3	Function result type, number of routine parameters, type of each parameter.

Default 3

Location Anywhere.

CHECK_ACTUAL_PARM affects every routine call that follows it (until superceded by another CHECK_ACTUAL_PARM). However, its practical use is to lower the type checking for a *particular* routine call. (Compare CHECK_FORMAL_PARM, which is intended to lower the type checking for *every* call to a specific routine.) If both CHECK_ACTUAL_PARM and CHECK_FORMAL_PARM apply to a routine, the linker uses the lower type-checking value.

The type-checking for an external routine is compatible with that of the language in which it is written. (An external routine is identified as such with the EXTERNAL directive.)

Example

```
PAGE   1 HEWLETT-PACKARD  ...  (C) HEWLETT-PACKARD CO. 1986 ...

    0    1.000   0
    0    2.000   0
    0    3.000   0   PROGRAM t;
    0    4.000   0
    0    5.000   0   TYPE
    0    6.000   0      int_ptr_type = ^integer;
    1    7.000   0      char_ptr_type = ^char;
    2    8.000   0
    2    9.000   0   VAR
    2   10.000   0      int_ptr  : int_ptr_type;
    3   11.000   0      char_ptr : char_ptr_type;
    4   12.000   0
    0   13.000   0   PROCEDURE proc (ip : int_ptr_type);
    0   14.000   0   EXTERNAL;
```

```
0   15.000   0
0   16.000   0   PROCEDURE $ALIAS 'proc'$ proc_c (cp : char_ptr_type);
2   17.000   0   EXTERNAL;
0   18.000   0
0   19.000   0
0   20.000   0   {Renaming the procedure gets around HP Pascal's}
0   21.000   0   {parameter type checking}
0   22.000   0
4   23.000   1   BEGIN
4   24.000   1
4   24.000   1      proc(int_ptr);
5   25.000   1
5   26.000   1      $CHECK_ACTUAL_PARM 2$
5   27.000   1      proc_c(char_ptr);
6   28.000   1
6   29.000   1      {Using CHECK_ACTUAL_PARM gets around the linker's}
6   30.000   1      {parameter type checking}
6   31.000   1
6   32.000   1   END.
```

CHECK_FORMAL_PARM

CHECK_FORMAL_PARM is an **HP Pascal** Option.

The CHECK_FORMAL_PARM compiler option determines how closely the formal parameters of a routine must match its actual parameters. If the formal and actual parameters are incompatible, the linker does not link the program.

Syntax

$CHECK_FORMAL_PARM *integer* $

Parameter

integer In the range 0..3. Determines how the linker checks actual parameters against formal parameters, as follows:

Value	**The linker checks:**
0	Nothing.
1	Function result type.
2	Function result type, number of routine parameters.
3	Function result type, number of routine parameters, type of each parameter.

Default 3

Location Anywhere.

CHECK_FORMAL_PARM affects every routine call that follows it (until superceded by another CHECK_FORMAL_PARM). It lowers the type checking for *every* call to these routines. (Compare CHECK_ACTUAL_PARM, which is intended to lower the type checking for a *particular* routine call.) If both CHECK_FORMAL_PARM and CHECK_ACTUAL_PARM apply to a routine, the linker uses the lower type-checking value.

The type-checking for an external routine is compatible with that of the language in which it is written. (An external routine is identified as such with the EXTERNAL directive.)

When you call an HP Pascal routine from a non-Pascal program, you must compile the HP Pascal routine with $CHECK_FORMAL_PARM 0$ to turn parameter checking off. Otherwise, HP Pascal generates type-checking information that the non-Pascal program cannot match.

The compiler does not generate type-checking code for intrinsic routines. (An intrinsic routine is identified as such with the INTRINSIC directive. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*.)

Example

```
     PAGE   1 HEWLETT-PACKARD  ...  (C) HEWLETT-PACKARD CO. 1986 ...

         0    1.000   0
         0    2.000   0
         0    3.000   0   $SUBPROGRAM$
         0    4.000   0   PROGRAM t;
         0    5.000   0
         0    6.000   0   $CHECK_FORMAL_PARM 0$
         0    7.000   0
         0    8.000   0   {CHECK_FORMAL_PARM prevents the linker from
         0    9.000   0    complaining if this procedure is called with
         0   10.000   0    fewer than seven actual parameters}
         0   11.000   0
         0   12.000   0   PROCEDURE proc (parm_count : integer;
         2   13.000   0                   parm1,
         3   14.000   0                   parm2,
         4   15.000   0                   parm3,
         5   16.000   0                   parm4,
         6   17.000   0                   parm5,
         7   18.000   0                   parm6      : integer);
         8   19.000   1   BEGIN
         8   20.000   1   END;
         8   21.000   0
         0   22.000   1   BEGIN
         0   23.000   1   END.
```

CODE

CODE is an **HP Pascal** Option.

When the CODE compiler option is ON, the compiler generates object code after parsing a compilation block.

The command line option -C also specifies this option.

Syntax

```
$CODE {ON }$
      {OFF}
```

Default ON

Location Anywhere, but it affects only the procedure, function, or outer block that contains it.

The CODE option affects an entire procedure, function, or outer block. To suppress the object code for smaller portions of source code, use the SKIP_TEXT option, or enclose that portion of source code in comment symbols.

Example

The compiler generates no object code for proc2. Although $CODE OFF$ is in the middle of proc2, it affects the entire procedure.

```
PROGRAM show_code;
PROCEDURE proc1;
BEGIN
   :
END;
PROCEDURE proc2;
```

```
BEGIN
   :
   $CODE OFF$
   :
END;
$CODE ON$
BEGIN
   :
END.
```

CODE_OFFSETS

CODE_OFFSETS is an **HP Pascal** Option.

When the CODE_OFFSETS compiler option is ON (and the LIST compiler option is ON), the compiler prints a table that contains the statement number and offset of each executable statement that it lists.

Syntax

```
$CODE_OFFSETS {ON }$
              {OFF}
```

Default OFF

Location Anywhere.

The offset is the address of the first machine instruction generated for the statement, relative to the start of the routine or outer block. It is in hexadecimal.

The table appears at the end of the compiler listing.

Example

```
 0    1.000   0   $LIST ON, CODE_OFFSETS ON$
 0    2.000   0   $STANDARD_LEVEL 'HP_MODCAL'$
 0    3.000   0   PROGRAM x (output);
 2    4.000   0   import arg;
 0    5.000   0   VAR
 0    6.000   0      x : integer;
 1    7.000   0      y : argarrayptr;
 2    8.000   0      s : string[40];
 5    9.000   1   BEGIN
 5   10.000   1      x := argc;
 6   11.000   1      writeln('There were ',x:1,' argv elements');
 7   12.000   1      writeln('Argv test');
 8   13.000   1      y := argv;
 9   14.000   1      FOR x := 1 TO argc-1 DO
10   15.000   2      BEGIN
10   16.000   2         setstrlen(s,0);
11   17.000   2         strmove(strmax(s), y^[x]^, 1, s, 1);
12   18.000   2         setstrlen(s, strpos(s,#0)-1);
13   19.000   2         writeln('Arg ',x:1,' = >',s,'<');
14   20.000   2      END;
14   21.000   1      writeln('Argn test');
15   22.000   1      FOR x := 0 TO argc-1 DO
16   23.000   1         writeln('Arg ',x:1,' = >',argn(x), '<');
17   24.000   1   END.

C O D E  O F F S E T S

PROGRAM

STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   STMT   OFFSET
   5       70      6       80      7      128      8      174      9      184
  10      1B0     11      1B8     12      21C     13      274     14      390
  15      3DC     16      404
```

Example

```
PROCEDURE outer;

   PROCEDURE inner;
   BEGIN
      .
      .
      .
   END;

BEGIN
   .
   .
   .
END;
```

```
                    C O D E  O F F S E T S

                         outer$4$inner

                 STMT   OFFSET   STMT   OFFSET
                    1       20      2     30

                              outer

                 STMT   OFFSET   STMT   OFFSET
                    1       10      2       2C
```

Outer4inner is the procedure label for the level two procedure, inner, contained in the level one procedure outer. Statement one of inner is offset 20 (hexadecimal) bytes from the address of inner.

NOTE This feature is intended for use with an assembly-level debugger. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide* for information on the debuggers.

If you use optimization with this option, the offsets will not be correct.

CONVERT_MPE_NAMES

CONVERT_MPE_NAMES is a **System-Dependent HP-UX** Option.

The CONVERT_MPE_NAMES compiler option converts file names in the BUILDINT, INCLUDE, LISTINTR, and SYSINTR compiler options from MPE format to HP-UX format.

The command line option +C also specifies this option.

Syntax

```
$CONVERT_MPE_NAMES$
```

Default None.

Location Anywhere.

Fully qualified HP-UX-format file names (those that begin with slash, like '/mnt/srf/file') are not converted.

This option assumes an HP-UX directory structure that is modeled after the MPE/iX accounting structure, in which all files reside in group-level directories and groups are subdirectories of accounts. This option

converts MPE/iX-format file names to lowercase letters.

For example, assume the HP-UX directory structure account/group, where group is a directory containing the file f. If a Pascal source program contains the statement

```
     $INCLUDE 'F.Group.Account'$
```
[REV BEG]

then the compiler prefixes[REV END] the appropriate path information to f, and searches for the resulting name (for example, if the compilation is performed in the group-level directory, then the compiler includes the file ../../account/group/f).

COPYRIGHT

COPYRIGHT is an **HP Pascal** Option.

The COPYRIGHT compiler option puts a copyright notice in the relocatable object file and the program file.

Syntax

```
     $COPYRIGHT string_literal $
```

Parameter

string_literal The name of the copyright owner, to appear in the copyright notice. The compiler distinguishes between uppercase and lowercase letters.

Default None.

Location At front.

The copyright notice is:

```
     (C) Copyright date_string by string_literal.
     All rights reserved. No part of this program may be
     photocopied, reproduced, or transmitted without prior
     written consent of string_literal.
```

The default *date_string* is the current year (see the COPYRIGHT_DATE compiler option).

Example

```
     $COPYRIGHT 'Blaise Pascal'$
     PROGRAM show_copyright;
     BEGIN
        :
     END.
```

The preceding program produces the following copyright notice:

```
     (C) Copyright 1986 by Blaise Pascal. All rights reserved.
     No part of this program may be photocopied, reproduced, or
     transmitted without prior written consent of Blaise Pascal.
```

COPYRIGHT_DATE

COPYRIGHT_DATE is an **HP Pascal** Option.

The COPYRIGHT_DATE compiler option specifies the date that appears in the copyright notice.

Syntax

```
$COPYRIGHT_DATE date_string $
```

Parameter

date_string Specifies the date string to appear in the copyright notice, as follows:

> (C) Copyright *date_string* by *string_literal*.
> All rights reserved. No part of this program may be photocopied, reproduced, or transmitted without prior written consent of *string_literal*.

(The COPYRIGHT option sets *string_literal*.)

Default Current year.

Location At front.

The COPYRIGHT_DATE compiler option has no effect if the program does not contain the COPYRIGHT compiler option, which puts the copyright notice into the relocatable object and program files.

Example

```
$COPYRIGHT 'Blaise Pascal'$
$COPYRIGHT_DATE '1682,1683,1684,1685,1686'$
PROGRAM show_copyright;
BEGIN
END.
```

The copyright notice for the preceding program is:

```
(C) Copyright 1682,1683,1684,1685,1686 by Blaise Pascal.
All rights reserved.  No part of this program may be
photocopied, reproduced, or transmitted without prior
written consent of Blaise Pascal.
```

ELSE

ELSE is an **HP Pascal** Option.

The ELSE compiler option specifies the code to be compiled when the Boolean expression in the IF compiler option has the value FALSE. See the IF option for more information.

Syntax

```
$ELSE$
```

Default Not applicable.

Location Anywhere.

Example 1

```
$SET 'group1=FALSE'$
   .
   .
   .
$IF 'group1'$

 [source_line ]
 [ .         ]
 [ .         ]
 [ .         ]
$ELSE$

 [source_line ]
```

```
 [ .         ]
 [ .         ]
 [ .         ]

     $ENDIF$
```

In this example, the code following $ELSE is compiled because group1 is set to FALSE.

Example 2

```
     $SET 'group3=true,group2=false;group1=false'$
        .
        .
        .
     $IF 'group1'$

 [source_line ]
 [ .         ]{group1}
 [ .         ]
 [ .         ]
     $ELSE$
        $IF 'group2'$

 [source_line ]
 [ .         ]{group2}
 [ .         ]
 [ .         ]

     $ELSE$
        $IF 'group3'$

 [source_line ]
 [ .         ] {group3}
 [ .         ]
 [ .         ]
           $ENDIF$
        $ENDIF$
     $ENDIF$
```

In this example, only group3 is compiled because it is set to true and group1 and group2 are set to false.

ENDIF

ENDIF is an **HP Pascal** Option.

The ENDIF compiler option ends the code to be conditionally compiled. See the IF compiler option for more information.

Syntax

```
     $ENDIF$
```

Default Not applicable.

Location Anywhere.

Example

```
     $SET 'group1=true, group2=false'$
        .
        .
        .
     $IF 'group1 AND (NOT group2)'$

[source_line ]
[ .         ]
```

```
[ .         ]
[ .         ]
     $ENDIF$
```

EXTERNAL

EXTERNAL is an **HP Pascal** Option.

The EXTERNAL compiler option causes the compiler to generate code for routines, but not for statements in the outer block. It also generates symbolic information about global variables, allowing them to be matched (by external name) to their counterparts in the compilation unit compiled with the GLOBAL compiler option. (The EXTERNAL compiler option is used in compilation units compiled with the SUBPROGRAM compiler option.)

Syntax

```
$EXTERNAL ['{PASCAL}']$
          [ {NONE  } ]
```

Parameters

PASCAL	Causes the compiler to include type-checking information in the object file so that the global variables can be compared to those in a compilation unit that was compiled with $GLOBAL 'PASCAL'$.
NONE	Prevents the compiler from including type-checking information for global variables in the object file.
No parameter	Same as PASCAL.

Default PASCAL.

Location At front.

The EXTERNAL option, in conjunction with the GLOBAL option, enables you to compile one program as two or more compilation units. Specify the GLOBAL option in the compilation unit that declares the global variables and contains the main program. Specify the EXTERNAL option in each of the other compilation units that declare routines, and in each of the global variables that those routines use. A compilation unit cannot contain both the EXTERNAL option and the GLOBAL option.

A compilation unit with the EXTERNAL option does not need to declare all of the global variables. It only needs to declare the ones that it uses, and they can be in any order. See the example for the GLOBAL compiler option.

NOTE Do not confuse the EXTERNAL compiler option with the EXTERNAL directive. Refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for information on the EXTERNAL directive.

EXTNADDR

EXTNADDR is a **System Programming** Option.

The EXTNADDR compiler option specifies that a pointer type or pointer variable is a long pointer, and a reference parameter is a long address.

Syntax

```
     $EXTNADDR$
```

Default Not applicable.

Location In a pointer type or variable declaration, between ^ or @ and the type name, or in a parameter list, between VAR, ANYVAR, or READONLY and the following parameter name.

Example

```
TYPE
   RecType = RECORD
               F1 : INTEGER;
               F2 : CHAR;
             END;
   ExtRecType = ^$EXTNADDR$ RecType;
   IntPtrType = ^INTEGER;
   ExtPtrType = ^$EXTNADDR$ integer;
VAR
   ExtVar : ^$EXTNADDR$ integer;  {cannot be a parameter to new}
   ExtP1,
   ExtP2  : ExtPtrType;
   IntP   : IntPtrType;
   RecP   : ExtRecType;

PROCEDURE ExtProc (VAR $EXTNADDR$ Parm1,Parm2 : IntPtrType);

PROCEDURE ExtProc2 (VAR $EXTNADDR$ Parm3 : INTEGER;
                    VAR Parm4 : INTEGER;
                        Parm5 : INTEGER);
```

FONT

FONT is a **System-Dependent MPE/iX** Option.

The FONT compiler option specifies primary and secondary character sets to be used in the title and comments in the listing (provided that the printer supports multiple fonts, as the HP268x laser printers do.)

Syntax

$FONT *string* $

Parameter

string Is of the form:

'*primary_font*,*secondary_font* '

Where:

primary_font Is an unsigned integer that sets the number for the primary font.

secondary_font Is an unsigned integer that sets the number for the secondary font.

Default Not applicable.

Location Anywhere.

To change fonts within the *string_literal* parameter of the TITLE option, or within a comment, shift to the secondary character set with CONTROL N. Shift back to the primary character set with CONTROL O.

Example

Assume that font 5 is *this font* in the environment file.

```
$
```

```
     FONT '0,5'$
     $TITLE 'Dptcore. CONTROLNPort Data DefinitionsCONTROLO'$
        .
        .
        .
     PROCEDURE Proc1;  {This is the CONTROLNfirstCONTROLO procedure}
        .
        .
        .
```

The listing prints the title and comment shown above this way:

```
     Dptcore. Port Data Definitions
     {This is the first  procedure}
```

GLOBAL

GLOBAL is an **HP Pascal** Option.

The GLOBAL compiler option causes the compiler to generate code for the entire compilation unit (including the outer block) and symbolic information about global variables that allows them to be matched with their counterparts in compilation units compiled with the EXTERNAL compiler option. See the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information.

Syntax

```
$GLOBAL ['{PASCAL}']$
        [ {NONE  } ]
```

Parameters

PASCAL — Causes the compiler to include type-checking information in the object file so that its global variables can be compared to those in a compilation unit that was compiled with $EXTERNAL 'PASCAL'$ (or its equivalent, $EXTERNAL$).

NONE — Prevents the compiler from emitting type-checking information for global variables.

Default PASCAL.

Location At front.

The GLOBAL option, in conjunction with the EXTERNAL option, enables you to compile one program as two or more compilation units. Specify the GLOBAL option in the compilation unit that declares all of the global variables and contains the main program. Specify the EXTERNAL option in each of the other compilation units (which declare routines and the global variables that those routines use). A compilation unit cannot contain both the GLOBAL option and the EXTERNAL option.

Example

One compilation unit:

```
     $GLOBAL$
     PROGRAM show_global (input,output);

     VAR
        a,b,c,d : integer;
        state : Boolean;

     PROCEDURE proc1; EXTERNAL;

     BEGIN                {Main program}
        .
```

```
        .
        .
     END.
```

Another compilation unit:

```
     $EXTERNAL$
     PROGRAM show_external (input,output);

     VAR
        state : Boolean;  {Matches variable in show_global's outer block}
                          {a,b,c,d need not be declared here because this
                           compilation unit does not use them.}
     PROCEDURE proc1;
     BEGIN
        .
        .
        .
     END;

     BEGIN
        {Empty outer block}
     END.
```

GPROF

GPROF is a **System-Dependent HP-UX** Option.

The compiler option GPROF causes the compiler to produce code that profiles itself as it runs. You can analyze the profiles with the HP-UX utility gprof.

Syntax

```
$GPROF {ON }$
       {OFF}
```

Default OFF.

Location Anywhere before the keyword PROGRAM (illegal in modules).

Example

```
     $GPROF ON$
     PROGRAM a;

     PROCEDURE b;
     BEGIN
     END;

     BEGIN
        b;
     END;
```

NOTE A program containing the GPROF compiler option must be linked with the *pc* option -G.

HEAP_COMPACT

HEAP_COMPACT is an **HP Pascal** Option.

When the HEAP_COMPACT compiler option is ON (and the HEAP_DISPOSE option is also ON), free space in the heap is concatenated when the predefined

procedure *dispose* is called.

Syntax

```
$HEAP_COMPACT {ON }$
              {OFF}
```

Default OFF.

Location At front.

The HEAP_COMPACT option is recommended for programs that manipulate many dynamic record variables of different sizes via calls to the predefined procedures *new* and *dispose*. It allows free space to be merged and reused.

Example

```
$HEAP_COMPACT ON; HEAP_DISPOSE ON$
PROGRAM show_compact;
TYPE
   big_rec = RECORD
                f1 : ARRAY [1..4] OF integer;
             END;
   small_rec = PACKED RECORD

                  f1 : integer;
                  f2 : integer;
               END;
VAR
  p1,p2 : ^mall_rec;
     p3 : ^big_rec;

BEGIN
   new(p1);
   new(p2);
   dispose(p1);
   dispose(p2);
   new(p3);     {p3 is allocated in the space previously
                 occupied by p1 and p2}
END.
```

HEAP_DISPOSE

HEAP_DISPOSE is an **HP Pascal** Option.

When the HEAP_DISPOSE compiler option is ON, the predefined procedure *dispose* frees space in the heap so that the predefined procedure *new* can reallocate it. By default, such disposed space cannot be reused.

Syntax

```
$HEAP_DISPOSE {ON }$
              {OFF}
```

Default OFF

Location At front.

Example

```
$HEAP DISPOSE ON$
PROGRAM show_heap;
TYPE
   big_array = ARRAY [1..1000] OF longreal;
VAR
   ptr : ^big_array;
   i : integer;
```

```
BEGIN
   FOR i := 1 TO maxint DO  {If HEAP_DISPOSE were OFF, the heap}
      BEGIN                 {would overflow and an error would occur}
         new(ptr);
         .
         .
         .
         dispose(ptr);
      END;
END.
```

HP_DESTINATION
[REV BEG]

HP_DESTINATION is a **System-Dependent HP-UX** Option.

Syntax

```
$HP_DESTINATION '{ARCHITECTURE PAmodel }'$
                {SCHEDULER PAmodel    }
```

Where:

ARCHITECTURE — Specifies the desired destination architecture.

PA*model* — Can be a model number, such as 750 or 870, or one of the following architecture specifications:

- 1.0 Generates object code suitable for all implementations of PA-RISC 1.0 or higher. This is the default for the Series 800 models.
- 1.1 Generates object code suitable for all implementations of PA-RISC 1.1. This is the default for all Series 700 models.

SCHEDULER — Specifies the desired instruction scheduling algorithm.

PA*model* — Can be a model number, such as 750 or 870, or one of the following architecture specifications:

- 1.0 Performs generic scheduling tuned to a model representative of PA-RISC 1.0 implementations.
- 1.1 Performs generic scheduling tuned to a model representative of PA-RISC 1.1 implementations.

Default — The native architecture of the machine the program is being compiled on.

Location — The beginning of the source file.

HP_DESTINATION 'ARCHITECTURE' Option

The HP_DESTINATION 'ARCHITECTURE' option specifies your intended destination architecture so you can cross-compile a program to run on a different PA-RISC architecture without having to purchase that machine. Specifying a destination architecture ensures that the compiler generates appropriate object code for that destination architecture.

The first occurrence of the HP_DESTINATION 'ARCHITECTURE' option takes precedence over later occurrences of the same option. For more information on HP_DESTINATION 'ARCHITECTURE', refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending

on your implementation.[REV END]
[REV BEG]

NOTE If you specify an architecture other than the native architecture of your machine, your compiled program may not run on your machine. Specifically, code compiled with HP_DESTINATION 'ARCHITECTURE PA1.1' may not run on a PA-RISC 1.0 machine.

NOTE A program containing the HP_DESTINATION 'ARCHITECTURE' compiler option must be linked with the pc command line option +DA*model*. This is because +DA*model* does more than specify the destination architecture. It also specifies which math libraries the program is to be linked with: PA-RISC 1.0 or PA-RISC 1.1. See the *HP-UX Floating Point Guide* for more information about using math libraries.

HP_DESTINATION 'SCHEDULER' Option

The HP_DESTINATION 'SCHEDULER' option specifies an instruction scheduling algorithm that is not native to your architecture; it optimizes your program for a PA-RISC architecture other than the one you are compiling on.

The first occurrence of the HP_DESTINATION 'SCHEDULER' option takes precedence over later occurrences of the same option. For more information on HP_DESTINATION 'SCHEDULER', refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation.

Note that the command line option +DS*model* also specifies this option.

This option can be used with the HP_DESTINATION 'ARCHITECTURE' option. For example, if you want your program to run on both a PA-RISC 1.0 architecture machine and a PA-RISC 1.1 architecture machine, you can use HP_DESTINATION 'ARCHITECTURE PA1.0' to specify PA-RISC 1.0 architecture. Because the PA-RISC 1.1 instruction set is a superset of the PA-RISC 1.0 instruction set, the code will run on both machines. If you use HP_DESTINATION 'SCHEDULER PA1.1', your program will run on both architectures, but will run as fast as possible on PA-RISC 1.1 architecture machines.[REV END]

HP3000_16

HP3000_16 is a **System-Dependent MPE/iX** Option.

The HP3000_16 compiler option specifies the Pascal/V packing algorithm for the allocation and alignment of all data structures.

Syntax

```
$HP3000_16$
```

Default HP Pascal optimized data structures (see the HP3000_32 compiler option).

Location At front.

The HP3000_16 compiler option causes all data types (except files and pointers) to be allocated and aligned according to the Pascal/V packing algorithm. A structure compiled by the HP Pascal compiler with HP3000_16

looks exactly like the same structure compiled by the Pascal/V compiler. This is useful for reading data files generated by Pascal/V.

HP3000_16 does not affect file and pointer types. The allocation and alignment of file variables is system-dependent, and HP Pascal does not allow the creation of files that contain files.

The allocation and alignment of pointers is also system dependent, so pointers are not portable. A pointer declared in an HP Pascal program can be used only with HP Pascal (not Pascal/V).

Real numbers declared in an HP3000_16 program are represented in MPE V floating-point representation. Operations performed with these numbers emulate MPE V floating-point operations.

All constants declared in an HP3000_16 program are Pascal/V constants.

Example

See the example for the HP3000_32 compiler option.

NOTE A program that contains the HP3000_16 compiler option cannot call the PAUSE intrinsic directly. The work-around is to declare PAUSE this way, instead of declaring it as an intrinsic:

```
PROCEDURE pause $ALIAS 'em_pause'$ (VAR r : real);
                                              EXTERNAL;
```

HP3000_32

HP3000_32 is a **System-Dependent MPE/iX** Option.

The HP3000_32 compiler option specifies that a given type in an HP3000_16 program is to be allocated and aligned according to the HP Pascal packing algorithm.

Syntax

```
$HP3000_32$
```

Default HP3000_32 is the default when HP3000_16 is not used.

Location After the symbol = in a type definition.

If a program does not specify HP3000_16, then HP3000_32 has no effect, and the compiler issues a warning.

A user-defined type that is within a structure declared with HP3000_32 must also be declared with HP3000_32.

A user-defined type that is within a structure declared without HP3000_32 must also be declared without HP3000_32.

HP3000_32 is illegal with these types:

* Boolean
* Char
* Integer
* Text

String, set, and real operations are illegal on HP3000_32 strings, sets,

and real numbers. HP3000_32 strings, sets, and real numbers are not assignment compatible with HP3000_16 strings, sets, and real numbers. Use the predefined procedures strconvert and *setconvert* and the intrinsic *HPFPconvert* to convert HP3000_32 strings, sets, and real numbers to HP3000_16 strings, sets, and real numbers.

Example

```
$HP3000_16$
PROGRAM show_packing_algorithms;

TYPE
   t_pac = PACKED ARRAY [1..10] OF char;
   s_pac = $HP3000_32$
           PACKED ARRAY [1..10] OF char;

   t_starray = ARRAY [1..5] OF string[10];
   s_starray = $HP3000_32$
               ARRAY [1..5] OF string[10];

   t_rec = RECORD
             f1 : -32768..32767;   {16 bits allocated}
             f2 : real;            {HP 3000 real number}
             f3 : string[10];      {16 bits allocated}
             f4 : t_pac;
             f5 : s_pac;           {error}
             f6 : t_starray;
             f7 : s_starray;       {error}
             f8 : s_starray;       {error}
           END;

   s_rec = $HP3000_32$ RECORD
              f1 : -32768..32767;  {32 bits allocated}
              f2 : real;           {IEEE real number}
              f3 : string[10];     {32 bits allocated}
              f4 : t_pac;          {error}
              f5 : s_pac;
              f6 : t_starray;      {error}
              f7 : s_starray;      {error}
           END;

   t_array = ARRAY [1..5] OF t_rec;
   t_array1 = ARRAY [1..5] OF s_rec;  {error}

   s_array = $HP3000_32$
             ARRAY [1..5] OF s_rec;

   s_array1 = $HP3000_32$
              ARRAY [1..5] OF t_rec;  {error}

   t_file = FILE OF t_rec;
   t_file1 = FILE OF s_rec;          {error}

   s_file = $HP3000_32$
            FILE OF s_rec;
   s_file1 = $HP3000_32$
             FILE OF t_rec;          {error}

   t_array2 = ARRAY [1..5] OF RECORD
                                f1 : -32768..32767;  {16 bits allocated}
                                f2 : real;           {HP 3000 real number}
                                f3 : string[10];     {16 bits allocated}
                              END;
   s_array = $HP3000_32$
             ARRAY [1..5] OF RECORD
                               f1 : -32768..32767;   {32 bits allocated}
                               f2 : real;            {IEEE real number}
                               f3 : string[10];      {32 bits allocated}
```

```
                                   END;

     VAR
        v_file1 : t_file;
        v_file2 : s_file;
        v_file3 : FILE OF t_rec;
        v_file4 : FILE OF s_rec;  {error}

     BEGIN
     END.
```

IF

IF is an **HP Pascal** Option.

The IF compiler option specifies code to be compiled conditionally, depending on the value of a Boolean expression.

Syntax

```
     $IF 'Boolean_expression '$
```

Parameter

Boolean_expression Any constant Boolean expression containing the operators AND, OR, and NOT and parentheses. The SET compiler option must assign the value TRUE or FALSE to each identifier before it appears in *Boolean_expression*. The identifier operands cannot have the spellings of Boolean operators (NOT, AND, OR). The operators are evaluated in the order dictated by HP Pascal operator precedence.

Default Not applicable.

Location Anywhere.

The IF option *must* be used with the SET and ENDIF options, and *can* be used with the ELSE option, as follows:

```
$SET 'identifier  = Boolean  [{,}identifier  = Boolean ]'$
                           [{;}                      ]
     $IF Boolean_expression$

[source code           ]
[to be compiled        ]
[if Boolean_expression ]
[is TRUE               ]
     $ELSE$

[source code           ]
[to be compiled        ]
[if Boolean_expression ]
[is FALSE              ]

     $ENDIF$
```

IF options can be nested; that is, the source code to be compiled conditionally can contain IF options. The maximum nesting level is 16.

Because the IF, ENDIF, ELSE, and SET options (together) allow conditional compilation, several programmers with different needs can use them to customize a single compilation unit.

Example 1

The following two program fragments are equivalent:

```
     {Fragment 1}
     $SET 'group1=true, group2=false'$
        .
        .
        .
     $IF 'group1 AND (NOT group2)'$

[source_line ]
[ .         ]
[ .         ]
[ .         ]
     $ENDIF$

     {Fragment 2}
     $SET 'group1 = true'$
     $SET 'group2 = false'$
        .
        .
        .
     $IF 'group1'$
        $IF 'NOT group2'$

 [source_line ]
 [ .         ]
 [ .         ]
 [ .         ]
        $ENDIF$
     $ENDIF$
```

Example 2

```
     $SET 'group1=FALSE'$
        .
        .
        .
     $IF 'group1'$

 [source_line ]
 [ .         ]
 [ .         ]
 [ .         ]

     $ELSE$

 [source_line ]
 [ .         ]
 [ .         ]
 [ .         ]
     $ENDIF$
```

Example 3

```
     $SET 'group3=true,group2=false;group1=false'$
        .
        .
        .
     $IF 'group1'$

 [source_line ]
 [ .         ]{group1}
 [ .         ]
 [ .         ]

     $ELSE$
        $IF 'group2'$

 [source_line ]
 [ .         ]{group2}
```

```
 [ .         ]
 [ .         ]
     $ELSE$
        $IF 'group3'$

 [source_line ]
 [ .         ] {group3}
 [ .         ]
 [ .         ]

           $ENDIF$
        $ENDIF$
     $ENDIF$
```

INCLUDE

INCLUDE is a **System-Dependent MPE/iX** and **HP-UX** Option.

The INCLUDE compiler option includes text from a specified file in the source code being compiled.

Syntax

```
     $INCLUDE string_literal $
```

Parameter

string_literal Specifies the name of the file to be included at the current position in the program. The file specification depends upon the operating system.

Default None.

Location Anywhere.

The file that contains the INCLUDE option is the *including* file, and the file specified by *string_literal* is the *included* file.

When the compiler encounters the INCLUDE option, it processes text from the included file, as if the text were part of the including file. When the included file ends, the compiler continues processing the including file, resuming with the line that follows the INCLUDE option; therefore, ignoring options and source code that follow INCLUDE on the same line).

An included file can contain an INCLUDE option; that is, included files can be nested. The maximum nesting level is the maximum number of files that the operating system allows to be open simultaneously.

On the HP-UX operating system, if the file to be included cannot be found or opened, and its name is not an absolute path name (that is, it does not start with the character "/"), then the compiler looks for the file in the following places (this is called the *search path*).

* The directory that contains the .p file being compiled (the main source file).
* The current working directory.
* The directory /usr/include.

If the file still cannot be found or opened, the compiler issues an error message and the compile aborts.

NOTE The *pc* option *+C* on HP-UX affects the INCLUDE compiler option (see the *HP Pascal/HP-UX Programmer's Guide*).

Example 1

This example applies only to HP-UX.

```
PROGRAM show_include;
VAR
   $INCLUDE '/users/pascal/prog1/global'$
BEGIN
   i := 3;
   j := 1.55;
END.
```

If the file /users/pascal/prog1/global is:

```
i : INTEGER;
j : REAL;
```

Then the preceding program is equivalent to:

```
PROGRAM show_include;
VAR
   i : INTEGER;
   j : REAL;
BEGIN
   i := 3;
   j := 1.55;
END.
```

Example 2

This example applies only to MPE/iX.

```
PROGRAM show_include;
VAR
   $INCLUDE 'global.prog1.pascal'$
BEGIN
   i := 3;
   j := 1.55;
END.
```

If the file global.prog1.pascal is:

```
i : INTEGER;
j : REAL;
```

Then the preceding program is equivalent to:

```
PROGRAM show_include;
VAR
   i : INTEGER;
   j : REAL;
BEGIN
   i := 3;
   j := 1.55;
END.
```

INCLUDE_SEARCH

INCLUDE_SEARCH is a **System-Dependent MPE/iX** and **HP-UX** Option.

You can use the INCLUDE_SEARCH compiler option to set or modify the search path used by the compiler. This search path specifies the order of directories a compiler searches to find files specified in the INCLUDE directive. The search stops on t he first successful attempt to open a file. Files specified in the INCLUDE directive are called *included*

files.

The command line option -I include-search path also specifies this option. You can specify multiple paths by repeating the command line option -I for each path.

syntax

$INCLUDE_SEARCH '[+] [&] *string* [, *string*]...'$

Parameter

Default None.

Location Anywhere.

The *string* parameter specifies a path for the compiler to search for an included file. This path is called the include-search path. The + parameter, if specified, appends this new include-search path to the end of the existing include-search path. If the + parameter is omitted, the new include-search path replaces the existing one.

Although there is no default search path for INCLUDE on the MPE/iX operating system, you can define one with the INCLUDE_SEARCH option. You can modify the name of the included file by appending each component of the include path to the search path. Using the & symbol, you can indicate that the compiler should search for the unmodified file name. Note that the unmodified file name is not searched first by default. In fact, it will not be searched at all if the & is omitted from the INCLUDE_SEARCH list. If an INCLUDE_SEARCH list is specified, the compiler will search *only* the locations specified in the include-search path.

The search order is:

1. The directory of the immediate including file.
2. The include-search path.
3. The user's current working directory.
4. The system standard location /usr/include.

MPE/iX Example

```
PROGRAM show_include;
   $INCLUDE_SEARCH '&, .exp, .official, .official:indy'$
   $INCLUDE 'globals.foo'$
BEGIN
END.
```

The compiler will attempt to find the included file globals.foo by looking successively for it under each filename modification specified by the include path. In this example, the compiler will search successively for the following files:

* globals.foo
* globals.foo.exp
* globals.foo.official
* globals.foo.official:indy

Note that the compiler attempts to open the unmodified filename globals.foo only because the first element of the include path is "&". The search will stop on the first successful attempt to open one of these files.

HP-UX Example

The following program is in a file called /tmp/test.p, and the current working directory is /users/myself/work.

```
PROGRAM show_include;
   $INCLUDE_SEARCH '../experimental, ../official, /c/official'$
   $INCLUDE 'globals'$
BEGIN
END.
```

The compiler will attempt to find the included file globals by searching successively in each location specified by the search path. In this example, the compiler will look for the files listed below in the following order.

1. In the directory of the including file: /tmp/globals
2. In each element of the include-search path:

   /users/myself/experimental/globals

   /users/myself/official/globals

   /c/official/globals
3. In the current working directory: /users/myself/work/globals
4. In the system standard directory: /usr/include/globals

The search will stop at the first successful attempt to open one of these files.

If set with INCLUDE_SEARCH, the include-search path becomes part of the search path used by INCLUDE. Each path specified in the INCLUDE_SEARCH option denotes a directory in which the compiler will look, in turn, for an included file. The search stops after the first successful attempt to open the file.

INLINE

INLINE is an **HP Pascal** Option.

The INLINE compiler option causes the code for a certain routine to be duplicated in-line wherever it is called.

Syntax

```
$INLINE$
```

Default None.

Location Heading.

The advantage of duplicating routine code in-line is that it eliminates the overhead of routine calls. Unlike macro expansion, it preserves call-by-reference parameters as such and allows local parameters. The disadvantages are that it increases the amount of object code and prevents recursion: a routine whose code is duplicated in-line cannot call itself or any other routine that calls it.

Example

```
PROCEDURE Proc1 (X,Y: Integer) $INLINE$;
   .
   .
   .
PROCEDURE Proc2 $INLINE$
```

```
        (X,Y: Integer);
        .
        .
        .
```

In each compilation unit where you want to duplicate the code of a specific routine in-line, you must specify the entire routine definition. If you use the same routine in-line in more than one compilation unit, put them in a separate file and use the INCLUDE compiler option to include that file in each compilation unit.

Example

The file procfile contains this procedure, which other compilation units use in-line:

```
PROCEDURE x (a,b : integer; VAR c : char) $INLINE$;
BEGIN
   c := chr(a+b);
END;
```

The following compilation unit uses the procedure x in-line:

```
PROGRAM prog;
BEGIN
      .
      .
      .
   $INCLUDE 'procfile'$
      .
      .
      .
END.
```

The INLINE compiler option is equivalent to the INLINE procedure option. The procedure option requires STANDARD_LEVEL 'EXT_MODCAL'; the compiler option does not.

You cannot debug inline routines with a symbolic debugger. You can debug routines that call inline routines, but the inlined code is treated as a single statement and skipped. Breakpoints can only be set before and after the inlined code.

INTR_NAME

INTR_NAME is an **HP Pascal** Option.

The INTR_NAME compiler option specifies the name to be returned for an intrinsic. It is only valid when used in conjunction with the BUILDINT compiler option.

Syntax

$INTR_NAME *string_literal* $

Parameter

string_literal Specifies the return name of the intrinsic to be entered in the intrinsic file. Lowercase and uppercase are significant.

Default None.

Location Heading.

If an entry in the intrinsic file specifies INTR_NAME when it is added to the intrinsic file, the name *string_literal* is returned by the intrinsic file search mechanism, and is used in calls to the intrinsic routine.

Actually, the intrinsic search mechanism searches for the intrinsic name (other than specified by INTR_NAME) or the alias (if specified by the ALIAS compiler option), but it returns the name specified by INTR_NAME.

Example

```
$BUILDINT 'MYINTR'$
PROGRAM Show_Buildint;

   PROCEDURE Proc2 (p1 : Boolean;
                    p2 : integer;
                    p3 : real
                   );

   $ALIAS 'proc2alias'$
   $INTR_NAME 'proc2returnname'$
   EXTERNAL;

BEGIN
END.
```

The intrinsic file search mechanism searches for proc2alias, but returns proc2returnname.

KEEPASMB

KEEPASMB is an **HP Pascal** Option.

The KEEPASMB compiler option causes the compiler to leave behind an assembler source file containing the code for the entire compilation unit. This file can usually be run through the assembler to produce the same object file that the compiler produces directly.

On MPE/iX, the KEEPASMB option produces a file with the formal designator PASASSM, which is a temporary file by default. You are recommended to file-equate this name. You must file-equate it if the current group contains more than one compilation unit, or if the resultant assembler source is too big. For information on file equations, refer to the *MPE/iX Commands Reference Manual*.

On HP-UX, the KEEPASMB option produces a file with the same name as the source file, except that its suffix is .s instead of .p.

The command line option -S also specifies this option.

Syntax

```
$KEEPASMB {ON }$
          {OFF}
```

Default OFF

Location At front.

When you use the LIST_CODE option with KEEPASMB, LIST_CODE turns KEEPASMB on.

Example

```
$KEEPASMB ON$
PROGRAM x;
BEGIN
END.
```

The program above produces the following assembly file:

```
          .SPACE $TEXT$
          .SUBSPA $LIT$,QUAD=0,ALIGN=8,ACCESS=44
```

```
     .SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=44,CODE_ONLY
PROGRAM
_start
     .PROC
     .CALLINFO CALLER,FRAME=0,SAVE_SP,SAVE_RP
     .ENTRY
     STW    2,-20(0,30)  ;offset 0x0
     LDO    48(30),30    ;offset 0x4
     STW    0,-4(0,30)   ;offset 0x8
     .CALL   ;

     BL   P_INIT_ARGS,2  ;offset 0xc
     NOP     ;offset 0x10
     .CALL   ;
     BL   U_INIT_TRAPS,2 ;offset 0x14
     NOP     ;offset 0x18
$00002711
     .CALL
     BL   P_TERMINATE,2  ;offset 0x1c
     NOP     ;offset 0x20
     NOP     ;offset 0x24
     .CALL
     BL   U_EXIT,2       ;offset 0x28
     NOP         ;offset 0x2c
     LDW   -68(0,30),2   ;offset 0x30
     BV   0(2)  ;offset 0x34
     .EXIT
     LDO   -48(30),30    ;offset 0x38
     .PROCEND ;ln=24,25,26;
     .SUBSPA $UNWIND$,QUAD=0,ALIGN=8,ACCESS=44
     .WORD PROGRAM
     .WORD PROGRAM+56 ; = 0x38
     .WORD 24 ; = 0x18
     .WORD 6 ; = 0x6
     .SUBSPA $RECOVER$,QUAD=0,ALIGN=4,ACCESS=44
     .SPACE $PRIVATE$
     .SUBSPA $DATA$,QUAD=1,ALIGN=8,ACCESS=31
     .SUBSPA $GLOBAL$,QUAD=1,ALIGN=8,ACCESS=31
M$1
     .ALIGN 8
     .BLOCKZ 8
     .SPACE $TEXT$
     .SUBSPA $CODE$
     .EXPORT PROGRAM,PRIV_LEV=3
     .EXPORT _start,PRIV_LEV=3
     .IMPORT P_INIT_ARGS,CODE
     .IMPORT U_INIT_TRAPS,CODE
     .IMPORT P_TERMINATE,CODE
     .IMPORT U_EXIT,CODE
     .END
```

LINES

LINES is an **HP Standard** Option.

The LINES compiler option specifies the number of lines per page of the listing.

The command line option -P also specifies this option.

Syntax

$LINES *integer* $

Parameters

integer Positive integer not less than 20.

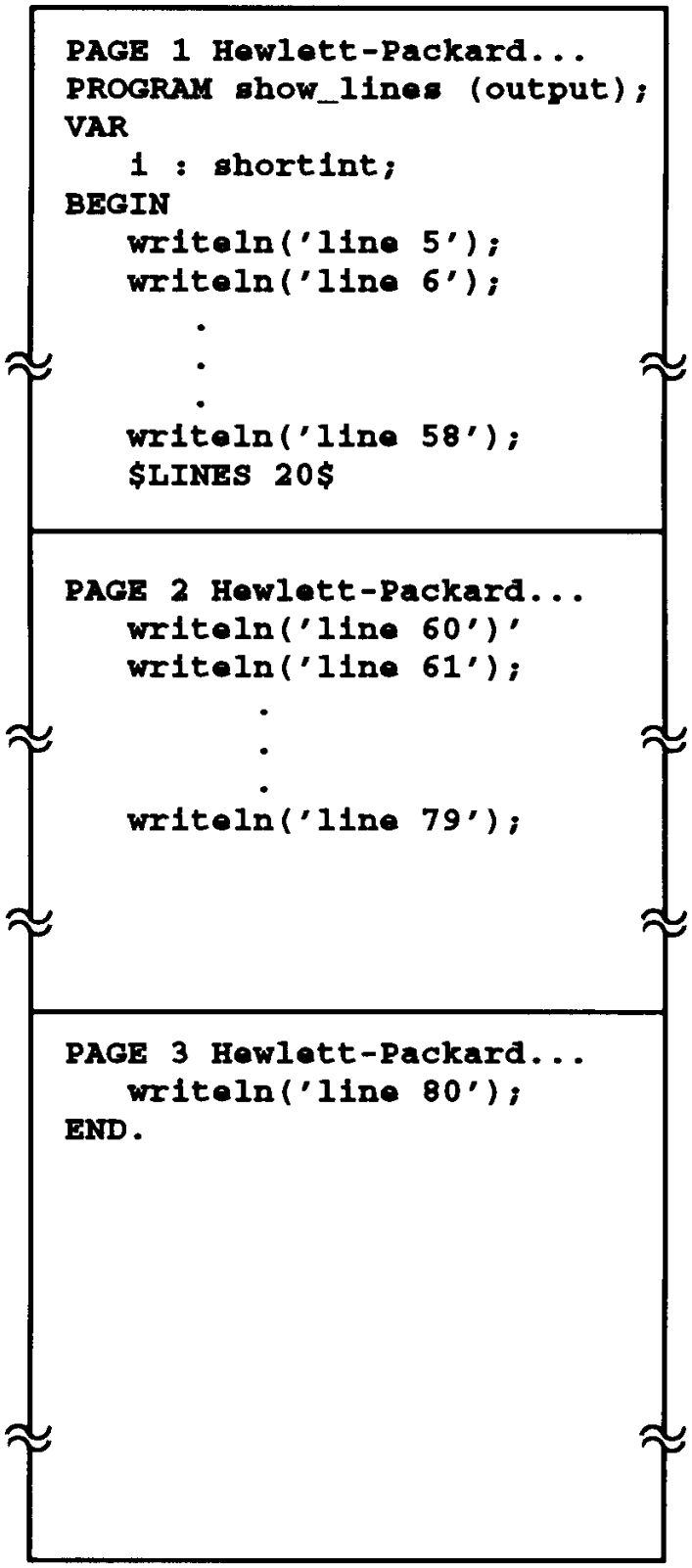
Default 59

Location Anywhere.

Example

```
PROGRAM show_lines (output);
VAR
   i : shortint;
BEGIN
   writeln('line 5');
   writeln('line 6');
      .
      .
      .
   writeln('line 58');
   $LINES 20$
   writeln('line 60');
   writeln('line 61');
      .
      .
      .
   writeln('line 79');

  writeln('line 80');
END.
```

The listing (simplified) looks like this:


PAGE 1 Hewlett-Packard...
PROGRAM show_lines (output);
VAR
i : shortint;
BEGIN
writeln('line 5');
writeln('line 6');
writeln('line 58');
$LINES 20$
PAGE 2 Hewlett-Packard...
writeln('line 60')'
writeln('line 61');
writeln('line 79');
PAGE 3 Hewlett-Packard...
writeln('line 80');
END.

LIST

LIST is an **HP Standard** Option.

When the LIST compiler option is ON, the compiler produces a listing of the source code.

The command line option -L also specifies this option.

Syntax

```
$LIST {ON }$
      {OFF}
```

Default ON.

Location Anywhere.

The first column of the listing shows the source statement number. This number appears in the code offset table, is used by the symbolic debugger, and is returned by the predefined function *statement_number*.

The second column of the listing shows a line number. The line number is provided by the editor if the source file is numbered; by the compiler if the source file is unnumbered. If the compiler numbers the lines, the lines are numbered consecutively, starting with 1. Included files are numbered separately (see the second example below, and the paragraph above it).

The third column of the listing shows the source statement nesting level (if the line is part of a structured statement). If the line was not compiled (because it is a comment or is affected by the SKIP_TEXT option), then ** replaces the number.

The end of the listing shows the processor time, elapsed time, the number of lines compiled, the number of lines compiled per minute, and the number of notes, warnings, and errors issued during the compilation. Sample listings in this manual omit this information (except where the example requires it). Times and rates vary, depending on the operating system, the memory configuration, system load, and the number of source lines.

If the compiler issues a message for a source line, it appears beneath that line in the listing in this form:

```
     ****  {NOTE}
           {WARNING}
           {ERROR} #ord_num  [message ] (message_num )
```

If the compiler can pinpoint the item in the source line that caused the note, warning, or error, the listing indicates that item with a caret (^).

The *ord_num* is the ordinal number of the note, warning, or error (it is the *ord_num* th note, warning, or error in the compilation). The *message_num* is the number that identifies the message, and *message* is the text that explains it.

Error and warning messages on multipage listings are chained; that is, the first such message on a page gives the page number of the previous such message. If the listing has no error or warning messages, its last page states this.

If LIST is OFF, and the compiler issues a message, it prints both the name of the include file that contains the line, and a copy of the line before issuing the message.

The LIST option must be ON for other options that affect the listing to

have any effect.

Example

```
PAGE   1 HEWLETT-PACKARD  ...  (C) HEWLETT-PACKARD CO. 1986 ...

    0    1.000   0
    0    2.000   0
    0    3.000   0   PROGRAM sort (infile,outfile,output);
    0    4.000   0
    0    5.000   0   VAR
    0    6.000   0      infile : text;
    1    7.000   0      outfile : text;
    2    8.000   0
   **    8.100   0   (* This line and the next three are not compiled:
   **    8.200   0   CONST
   **    8.300   0      max_array_size = 20000;
   **    8.400   0   *)
    2    9.000   0   CONST
    2   10.000   0      max_array_size := 4000;
                                      ^
**** ERROR # 1           FOUND UNEXPECTED ":=" (025)
    3   11.000   0   TYPE
    3   12.000   0      data_type = integer;
    4   13.000   0
    4   14.000   0   VAR
    4   15.000   0      data_array = array [1..max_array_size] OF data_type;
                                    ^
**** ERROR #2             FOUND UNEXPECTED "=" (025)
    5   16.000   0
    5   17.000   0      array_size : 0..max_array_size;
    6   18.000   0
    6   19.000   0   $PAGE$

PAGE   2 HEWLETT-PACKARD  ...  (C) HEWLETT-PACKARD CO. 1986 ...
    0   20.000   0   PROCEDURE read_data;
    1   21.000   1   BEGIN
    1   22.000   1      reset(infile);
    2   23.000   1      array_size := 0;
    3   24.000   1
    3   25.000   1      WHILE ((NOT eof(infile))
    4   26.000   1            AND
    4   27.000   2            (array_size < max_array_size)) DO BEGIN
    4   28.000   2
    4   29.000   2            array_size := array_size + 1;
    5   30.000   2            readln(infile,data_array[array_size]);
    6   31.000   2      END;
    6   32.000   1
    6   33.000   2      IF (NOT eof(infile)) THEN BEGIN
    7   34.000   2         writeln('Too many data points for sort program.');
    8   35.000   2         writeln('Sorting partial list only.');
    9   36.000   2      END;
    9   37.000   1
    9   38.000   1      close(infile);
   10   39.000   1   END;
   10   40.000   0
    0   41.000   0   $PAGE$

PAGE   3 HEWLETT-PACKARD  ...  (C) HEWLETT-PACKARD CO. 1986 ...

    0   42.000   0   PROCEDURE write_data;
    1   43.000   0   VAR
    1   44.000   0      index : 0..max_array_size;
    2   45.000   1   BEGIN
    2   46.000   1      rewrite(outfile);
    3   47.000   1
    3   48.000   1      FOR index := 1 TO array_size DO
    4   49.000   1         writeln(outfile,'data_array[index]);
```

```
    5   50.000   1
    5   51.000   1      close(outfile);
    6   52.000   1   END;
    6   53.000   0
    0   54.000   0   $PAGE$

PAGE   4 HEWLETT-PACKARD  ...  (C) HEWLETT-PACKARD CO. 1986 ...

    6   55.000   1   BEGIN
    6   56.000   1      writeln('starting sort');
    7   57.000   1
    7   58.000   1      read_data;
    8   59.000   1      sort_data;
                           ^
PREVIOUS ERROR ON PAGE  1
**** ERROR #  3   IDENTIFIER NOT DEFINED (014)
    9   60.000   1      write_data;
   10   61.000   1
   10   62.000   1      writeln('sort done');
   11   63.000   1   END.

          NUMBER OF ERRORS = 3      NUMBER OF WARNINGS =  0
          PROCESSOR TIME 0: 0: 0    ELAPSED TIME 0: 0: 0
          NUMBER OF LINES =   63    LINES/MINUTE =  0.0
          NUMBER OF NOTES =  0
```

Line numbers for statements in included files are independent of line numbers for the files that include them.

Example

```
0    1.000    0    $LIST ON$                                  }
0    2.000    0    PROGRAM show_list (input,output);          } Lines 1-3 of show_list
0    3.000    0    $INCLUDE 'decls1'$                         }
0    1.000    0    CONST                                      }
0    2.000    0       k = 100;                                }
1    3.000    0    VAR                                        } Lines 1-5 of decls
1    4.000    0       n : integer;                            }
2    5.000    0       t : Boolean;                            }
3    4.000    0    $INCLUDE 'checkp'$                           Line 4 of show_list
3    1.000    0                                               }
0    2.000    0    PROCEDURE check (VAR b : Boolean);         }
2    3.000    1    BEGIN                                      }
2    4.000    1       IF n > k THEN b := true;                } Lines 1-6 of chekp
4    5.000    1       ELSE b := true;                         }
5    6.000    1    END;                                       }
3    5.000    1    BEGIN                                      }
3    6.000    1       readln(n);                              }
4    7.000    1       check(t);                               } Lines 5-10 of show_list
5    8.000    1       IF t THEN writeln ('Too big')           }
7    9.000    1       ELSE writeln ('No Problem');            }
8   10.000    1    END.                                       }
```

LIST_CODE

LIST_CODE is an **HP Pascal** Option.

When the LIST_CODE compiler option is ON (and the LIST option is also ON), the compiler produces a mnemonic listing of the object code of each procedure in the program. The mnemonic listing appears at the end of the source listing of the compilation unit.

Syntax

```
$LIST_CODE {ON }$
           {OFF}
```

Default OFF.

Location Anywhere.

Scope Applies to the entire compilation unit that contains it. The effective value is the last value before the compilation unit's END statement.

Example

```
     PAGE 1 HEWLETT-PACKARD ...

        0    1.000  0   $LIST_CODE ON$
        0    2.000  0   $STANDARD_LEVEL 'HP_MODCAL'$
        0    3.000  0   $OS 'HPUX'$
        0    4.000  0   PROGRAM x;
        0    5.000  0   VAR
        0    6.000  0      lp : globalanyptr;
        1    7.000  0      bigarr : PACKED ARRAY [1.10] OF char;
        2    8.000  0      i,j : integer;
        4    9.000  1   BEGIN
        4   10.000  1      i := 5;  j := 10;
        6   11.000  1      lp := addr(bigarr, i+j);
        7   12.000  1   END.

                                  PROGRAM

         0    STW    2,-20(0,30)          38    LDW    28(0,27),22
         4    LDO    48(30),30            3C    ADD0   21,22,1
         8    STW    0,-4(0,30)           40    ADD    19,1,31
         C    BL     P_INIT_ARGS,2        44    STW    20,8(0,27)
        10    NOP                         48    STW    31,12(0,27)
        14    BL     U_INIT_TRAPS,2         00002711
        18    NOP                         4C    BL     P_TERMINATE,2
        1C    LDI    5,1                  50    NOP
        20    STW    1,32(0,27)           54    NOP
        24    LDI    10,31                58    BL     U_EXIT,2
        28    STW    31,28(0,27)          5C    NOP
        2C    LDO    16(27),19            60    LDW    -68(0,30),2
        30    LDSID  (0,19),20            64    BV     0(2)
        34    LDW    32(0,27),21          68    LDO    -48(30),30
```

LISTINTR

LISTINTR is an **HP Pascal** Option.

The LISTINTR compiler option lists to a specified file the contents of an intrinsic file. The intrinsic file is that specified by the BUILDINT or SYSINTR compiler option. If neither BUILDINT nor SYSINTR is specified, the system intrinsic file is accessed.

Syntax

```
$LISTINTR [string ]$
```

Parameter

string Specifies the name of the file into which the compiler lists the contents of the intrinsic file that BUILDINT specifies.

Default 'PASLIST'.

Location Anywhere.

On MPE the default size is 1023 records. If this record limit is too small, the LISTINTR operation will not complete. You can use the :BUILD command or a :FILE equation to specify a larger file. For more information on :BUILD and :FILE, see the *MPE/iX Commands Reference Manual*.

NOTE The *pc* option +C on HP-UX affects the LISTINTR compiler option (see the *HP Pascal/HP-UX Programmer's Guide*).

Example

```
                              Intrinsic File Listing

        Display of SYSINTR.PUB.SYS
        ( TUE, OCT 7, 1986, 4:33 PM )

        fopen (FOPEN) :
        LANGUAGE is HP PASCAL
        FUNCTION [SHORTINT(16) at OFFSET 0] with 13 PARAMETERS
        PARM # 1: STRUCTURE(65536) at OFFSET 0 by UNCHECKABLE ANYVAR;
                   SHORT ADDR, 8-BIT ALIGNED
                  DefaultValue = NIL
        PARM # 2: SHORTNNINT(16) at OFFSET 32 by VALUE
                  DefaultValue = NIL
        PARM # 3: SHORTNNINT(16) at OFFSET 48 by VALUE
                  DefaultValue = NIL
        PARM # 4: SHORTINT(16) at OFFSET 64 by VALUE
                  DefaultValue = 0
        PARM # 5: STRUCTURE(65536) at OFFSET 80 by UNCHECKABLE ANYVAR;
                   SHORT ADDR, 8-BIT ALIGNED
                  DefaultValue = NIL
        PARM # 6: STRUCTURE(65536) at OFFSET 112 by UNCHECKABLE ANYVAR;
                   SHORT ADDR, 8-BIT ALIGNED
                  DefaultValue = NIL
        PARM # 7: SHORTINT(16) at OFFSET 144 by VALUE
                  DefaultValue = 0
        PARM # 8: SHORTINT(16) at OFFSET 160 by VALUE
                  DefaultValue = 0
        PARM # 9: SHORTINT(16) at OFFSET 176 by VALUE
                  DefaultValue = 0
        PARM # 10: INTEGER(32) at OFFSET 192 by VALUE
                   DefaultValue = 0
        PARM # 11: INTEGER(16) at OFFSET 224 by VALUE
                   DefaultValue = 0
        PARM # 12: INTEGER(16) at OFFSET 240 by VALUE
                   DefaultValue = 0
        PARM # 13: INTEGER(16) at OFFSET 256 by VALUE
                   DefaultValue = 0

        fread (FREAD) :
        LANGUAGE is HP PASCAL
        FUNCTION [SHORTINT(16) at OFFSET 0] with 3 PARAMETERS
        PARM # 1: INTEGER(16) at OFFSET 0 by VALUE
        PARM # 2: INTEGER(65536) at OFFSET 16 by UNCHECKABLE ANYVAR;
```

```
                LONG ADDR, 8-BIT ALIGNED
        PARM # 3: SHORTINT(16) at OFFSET 80 by VALUE
```

LITERAL_ALIAS

LITERAL_ALIAS is an **HP Pascal** Option.

When the LITERAL_ALIAS compiler option is ON, the compiler takes aliases literally (exactly as they are spelled, differentiating between uppercase and lowercase letters). When LITERAL_ALIAS is OFF, the compiler downshifts aliases (or upshifts them if the compiler option UPPERCASE is ON).

Syntax

```
$LITERAL_ALIAS {ON }$
               {OFF}
```

Default OFF

Location Anywhere.

The LITERAL_ALIAS compiler option overrides the UPPERCASE compiler option.

Example

```
$LITERAL_ALIAS ON$
PROCEDURE $ALIAS 'PRoc1Name'$ PROC1;  {External name is PRoc1Name}

$LITERAL_ALIAS OFF$
PROCEDURE $ALIAS 'PRoc2Name'$ PROC2;  {External name is proc2name}
```

LOCALITY

LOCALITY is an **HP Pascal** Option.

The LOCALITY compiler option specifies a locality name to be associated with the code for all subsequent routines until the next LOCALITY option. The compiler puts the locality name in the object file.

Syntax

```
$LOCALITY string $
```

Parameter

string Specifies a locality name for the object code. The compiler does not distinguish between uppercase and lowercase letters in *string*.

Default The nameless locality.

Location Anywhere.

Using locality names can improve the performance of a program in cases where calling a routine in the same locality can require fewer instructions and fewer page faults than calling a routine in a different locality. If you use $LOCALITY and want to go back to using default locality, use $LOCALITY ‘CODE'$. Refer to LINKEDITOR manuals for details.

Example

```
$LOCALITY 'Sample'$
PROGRAM show_locality;
```

```
PROCEDURE proc1;
BEGIN
  .
  .
END;
BEGIN
  .
  .
  proc1;
  .
  .
END.
```

LONG_CALLS

LONG_CALLS is an **HP Pascal** option.

The LONG_CALLS option can be used to change the type of branches that are generated for calls or millicode calls.

Syntax

```
            {integer }
$LONG_CALLS {ON     }$
            {OFF    }
```

Parameters

0 or OFF — Regular short calls are generated.

1 or ON — Long calls are generated for regular calls and millicode calls.

2 — Millicode calls are long and regular calls are short.

3 — Millicode calls are short and regular calls are long.

Default — MPE/iX: 0

On HP-UX: 0 if any one of the following options are used:

-O +O +z +Z

3 if none of the above options are used.

Location — Anywhere, but in order to be effective, it must appear before a place in the code where label declarations or directives can appear.

Normally, for small programs, the branches generated reach their targets. If the branch does not reach, the linker generates Long Branch stubs. These stubs take longer to execute and may change the program's locality. These stubs are shared within a subspace, with one or more procedures.

By using the LONG_CALLS options, the compiler can generate a different and longer code sequence that always reach the branch. The disadvantage of using the longer call sequence is that the longer call sequence is done on calls that do reach the branch. This causes a code expansion for every call.

For HP-UX, millicode calls usually reach in program files, so options 1 and 2 are not needed. Also, they are not needed when compiling with +z or +Z (or SHLIB_CODE).

See the *Procedure Calling Conventions Reference Manual* and the *Precision Architecture Instruction Set Reference Manual* for more information on stubs and branches.

Example

```
$LONG_CALLS 1$
program call;
procedure p_r(x:real); external;
begin
  p_r(1.5);
end.
```

MAPINFO

MAPINFO is an **HP Pascal** Option.

The compiler option MAPINFO prints information for array and record types.

Syntax

```
$MAPINFO {ON }$
         {OFF}
```

Default OFF

Location Anywhere.

The information printed with the MAPINFO option is the same as that printed with the TABLES option set to ON (see "TABLES" in this chapter.) However, MAPINFO prints this information at the same time the type is declared instead of at the end of the scope in which the type is declared. In addition, MAPINFO prints the minimum alignment of the structured type.

Example

The example below shows a listing of PROGRAM p created with the MAPINFO option.

```
$MAPINFO ON$
PROGRAM p;
TYPE
   rec = RECORD
      f1 : integer;
      f2 : integer;
      END;

   REC                                MAX RECORD SIZE = x8 bytes
     F1                                 x0.0 @ 4.0
     F2                                 x4.0 @ 4.0
   MIN ALIGNMENT = x4 byte

BEGIN
END;
```

In the example above, the x indicates hexadecimal notation is being used. The table below further explains how to interpret the information generated by MAPINFO.

Relative Starting Position		**Storage Size**
x *bytes.bits*	@	*bytes.bits*

MLIBRARY

MLIBRARY is an **HP Pascal** Option.

The MLIBRARY compiler option specifies the file into which the compiler puts a compiled module definition, instead of putting it in the object file. The file specified here can then be used in a SEARCH option (see "SEARCH"). Program comments must not be written on the same line as $MLIBRARY.

Syntax

```
$MLIBRARY string $
```

Parameter

string Specifies the name of the file into which the compiler writes the module definition.

If the file exists, it must be an external library (otherwise, it is an error). If the file is an external library, the compiler updates the module definition in the file.

If the file does not exist, the compiler creates a new file with the specified name.

Default Compiled module definition goes into the object file.

Location Anywhere.

Example

```
$MLIBRARY 'xmodule'$
MODULE x;
   EXPORT
      .
      .
      .
   IMPLEMENT
      .
      .
      .
END.
```

NLS_SOURCE

NLS_SOURCE is a **System-Dependent MPE/iX** and **HP-UX** Option.

When the NLS_SOURCE compiler option is ON, the compiler supports the parsing of two-byte characters within string literals and comments.

The command line option -Y also specifies this option.

Syntax

```
$NLS_SOURCE {ON }$
            {OFF}
```

Default OFF.

Location Anywhere.

NLS_SOURCE ON enables the compiler to parse 16-bit characters within literal strings and comments. (Note that eight-bit characters are always parsed correctly.)

NLS_SOURCE OFF specifies that 16-bit characters are not supported.

Example

```
$NLS_SOURCE ON$
{Native Mode language source code can appear here.}
   .
   .
   .
CONST
   s = "some string literal";
$NLS_SOURCE OFF$
{Native Mode language source code cannot appear here.}
```

NOTE On MPE/iX, a warning occurs if the NLUSERLANG JCW is not set before compiling a program that turns the NLS_SOURCE compiler option ON.

On HP-UX, a warning occurs if the LANG environment variable is not set before compiling a program that turns the NLS_SOURCE compiler option ON.

NOTES

NOTES is an **HP Pascal** Option.

When the NOTES compiler option is ON, the compiler prints notes, which give you information that can help you correct possible run-time errors or make your program more efficient.

Syntax

```
$NOTES {ON }$
       {OFF}
```

Default ON.

Location Anywhere.

Example

```
PAGE 1 HEWLETT-PACKARD ...

   0    1.000  0  PROGRAM Note_Example;
   0    2.000  0
   0    3.000  0  VAR
   0    4.000  0     Ptr1 : LocalAnyPtr;
   1    5.000  0     Ptr2 : ^Integer;
   2    6.000  0
   2    7.000  1  BEGIN
   2    8.000  1     Ptr1 := NIL;
   3    9.000  1     Ptr2 := Ptr1;

**** NOTE #  1   CODE GENERATED TO VERIFY CORRECT POINTER ALIGNMENT (377)

   4   10.000  1     $NOTES OFF$
   4   11.000  1     Ptr2 := Ptr1;
   5   12.000  1  END.
```

OPTIMIZE

OPTIMIZE is an **HP Pascal** Option.

The OPTIMIZE compiler option specifies level one, level two, or no optimization for the program being compiled Refer to the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information on the optimizer.

The command line options +O1, +O2, and -O also specify this option.

Syntax

```
          {'LEVEL1'                }
          {'LEVEL2'                }
$OPTIMIZE {'BASIC_BLOCKS num '      }$
          {'BASIC_BLOCKS_FENCE num '}
          {ON                      }
          {OFF                     }
```

Parameters

LEVEL1	The compiler compiles the program with level one optimization.
LEVEL2	The compiler compiles the program with level two optimization. [REV BEG]
BASIC_BLOCKS *num*	The compiler compiles the program with level two optimization, but drops down to level one for those procedures with more than *num* basic blocks.
BASIC_BLOCKS_FENCE *num*	No optimization is requested but when it is, the number of basic blocks at which the compiler drops to level one is *num*. [REV END]
ON	The compiler compiles the program with level two optimization.
OFF	The compiler compiles the program without optimization.

Default OFF.

Location Anywhere, but in order to be effective, it must be before the place in the code where label declarations or directives can appear.

Scope All following source code, until overridden by another OPTIMIZE option.

Basic Blocks
[REV BEG]

A basic block is a sequence of code with a single entry point and a single exit point. A basic block has no internal branches. Optimizing procedures with a large number of basic blocks can take a long time and use a large amount of virtual memory. Therefore, the compiler behaves differently on large procedures, when optimizing at Level 2. Any procedure containing more than 500 basic blocks causes the optimizer to drop down to Level 1 optimization for that procedure. A warning is emitted that states the name of the procedure and the number of basic blocks it contains:[REV END]

```
Optdriver: <num> basic blocks; dropping to level 1 optimization for <proc>. (6059)
```

OPTIMIZE 'BASIC_BLOCKS num' Compiler Option

This option allows you to request Level 2 optimization and change the number of basic blocks at which the optimizer drops down to Level 1 optimization.

Syntax

```
$OPTIMIZE 'BASIC_BLOCKS num '$
```

where *num* is the number of basic blocks a procedure can have before the optimizer drops down to Level 1 optimization.

NOTE To get the "old" behavior of -O, (for example, to disable completely the basic blocks feature), you can use the following form of the directive:

```
$OPTIMIZE 'BASIC_BLOCKS 0'$
```

Notice that 0 has a special meaning here; it does not mean zero basic blocks.

On HP-UX, the +Obb*num* command-line option can be specified instead of the $OPTIMIZE 'BASIC_BLOCKS *num* '$ compiler option.

OPTIMIZE 'BASIC_BLOCK_FENCE num' Compiler Option

This option allows you to change the default level of basic blocks (500) at which the optimizer drops down to Level 1 optimization.

Syntax

```
$OPTIMIZE 'BASIC_BLOCK_FENCE num '$
```

where *num* is the number of basic blocks at which the optimizer drops down to Level 1 optimization.

This option does *not* request optimization; it only says that when Level 2 optimization is requested, to change the default level at which the optimizer drops down to Level 1.

NOTE To disable completely the basic blocks feature (for example, to disable the dropping from Level 2 to Level 1), you can use the following form of the directive:

```
$OPTIMIZE 'BASIC_BLOCK_FENCE 0'$
```

When this form of the option is specified Level 2 is requested, the "old" level 2 will be available; that is, no dropping from Level 2 to Level 1.

Notice that 0 has a special meaning here; it does not mean zero basic blocks.

Example

```
$OPTIMIZE 'LEVEL1'$
PROGRAM x;
   PROCEDURE y $OPTIMIZE 'LEVEL2'$; {Compiled with level two optimization}
   BEGIN {y}
      .
      .
      .
   END; {y}

   PROCEDURE z;  {Compiled with level two optimization}
   BEGIN {z}
      .
      .
      .
```

```
        END; {z}

        PROCEDURE a $OPTIMIZE OFF$;  {Compiled with no optimization}
           PROCEDURE b;  {Compiled with no optimization}
           BEGIN {b};
              .
              .
              .
           END; {b};
        BEGIN {a}
           .
           .
           .
        END; {a}

     BEGIN {x}  {Compiled with no optimization}
       .
       .
       .
     END. {x}
```

OS

OS is an **HP Pascal** Option.

The OS compiler option specifies the operating system on which the program is intended to run (not to be confused with the operating system on which it is compiled). Then, the compiler identifies language features that are not available on that operating system.

Syntax

```
      {NONE  }
      {HPUX  }
$OS ' {MPE/XL}'$
      {MPEXL }
      {MPE   }
```

Parameters

NONE — The compiler identifies language features that are unavailable on the HP-UX operating system or the MPE/iX and MPE V operating systems.

Available features are:
- All ANSI Pascal features.
- All HP Standard Pascal features.
- All HP Pascal predefined routines.

HPUX — The compiler recognizes language features that are available on the HP-UX operating system.

Available features are:
- All ANSI Pascal features.
- All HP Standard Pascal features.
- All HP Pascal predefined routines.
- Predefined procedure *assert*.
- Predefined function *baddress*.
- Predefined function *bitsizeof*.
- Predefined function *fnum*.
- Predefined function *sizeof*.
- Predefined function *waddress*.
- Standard program parameter *stderr*.

MPE/XL or MPEXL — The compiler recognizes language features that are available on the MPE/iX operating system.

Available features are:
- All ANSI Pascal features.

All HP Standard Pascal features.
All HP Pascal predefined routines.
Predefined procedure *assert*.
Predefined function *baddress*.
Predefined function *bitsizeof*.
Predefined function *fnum*.
Predefined function *sizeof*.
Predefined function *waddress*.
Predefined function *ccode*.
RUN command parameter INFO.
RUN command parameter PARM.

MPE — The compiler recognizes language features that are available on the MPE V operating system.

Available features are the same as for MPE/iX.

Default — Operating system on which the compiler is running.

Location — Anywhere.

If the compiler encounters a language feature that is unavailable on the intended operating system, it issues an error.

If you are writing a program on one operating system and intend to run it on another operating system, use the OS option to recognize language features that are available on the intended system.

Example

```
PROGRAM prog;

VAR
   condcode : 0..2;
     .
     .
     .
BEGIN
   $OS 'MPE'$
   condcode := ccode;  {this is legal}
     .
     .
     .
   $OS 'NONE'$
   condcode := ccode;  {this is a compile-time error}
END.
```

OVFLCHECK

OVFLCHECK is an **HP Pascal** Option.

When the OVFLCHECK compiler option is ON, the compiler generates overflow checking code for all integer arithmetic operations. Overflow-checking code stops the program and issues an error message if an arithmetic operation results in an integer overflow.

Syntax

```
$OVFLCHECK {ON }$
           {OFF}
```

Default — ON.

Location — Anywhere, but it affects an entire statement at a time. If OVFLCHECK is ON when the compiler processes a statement terminator, then all arithmetic operations in the statement are checked for overflow at run time. The OVFLCHECK option stays ON or OFF until another OVFLCHECK option overrides

it.

When OVFLCHECK is OFF, integer overflows are not detected. One use for this is in a random number generator, when overflows are expected and are to be ignored.

NOTE This option can be used to turn off overflow for bit32 multiplication; this option has no effect on bit52 or longint multiplication.

Example

```
PROGRAM t (output);

MODULE rand;
EXPORT
   FUNCTION random : integer;
   PROCEDURE init_random (seed,
                          range : integer);
IMPLEMENT
   CONST
      multiplier = 31415821;
   VAR
      rand_seed,
      rand_range : integer;

   PROCEDURE init_random (seed,
                          range : integer);
   BEGIN
      rand_seed := seed;
      rand_range := range;
   END;

   FUNCTION random : integer;
   BEGIN
      $PUSH, OVFLCHECK OFF$
      rand_seed := (rand_seed * multiplier +1) MOD rand_range;
      $POP$
      random := rand_seed;
   END;
END;

IMPORT rand;

BEGIN
   init_random(1234567,1000);
   writeln(random);
   writeln(random);
   writeln(random);
END.
```

PAGE

PAGE is an **HP Standard** Option.

The PAGE compiler option starts a new page of the listing if the LIST option is ON.

Syntax

```
$PAGE$
```

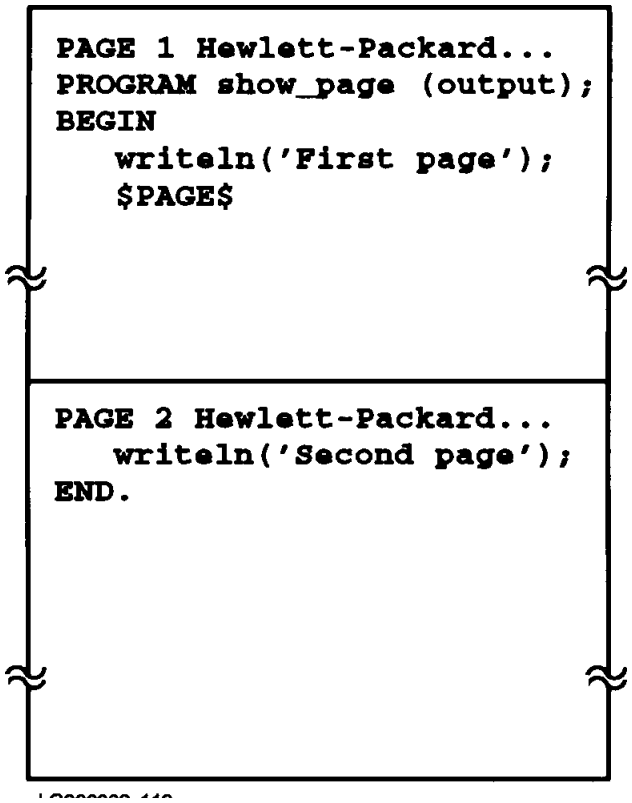
Default Not applicable.

Location Anywhere.

Example

```
PROGRAM show_page (output);
BEGIN
   writeln('First page');
   $PAGE$
   writeln('Second page');
END.
```

The listing (simplified) looks like this:



PAGEWIDTH

PAGEWIDTH is an **HP Pascal** Option.

The PAGEWIDTH compiler option specifies the width of the compiler listing.

Syntax

```
$PAGEWIDTH integer $
```

Parameter

integer An integer in the range 80..132, the number of characters per line in the compiler listing.

Default 120.

Location Anywhere.

Example

```
$PAGEWIDTH 80$
```

PARTIAL_EVAL

PARTIAL_EVAL is an **HP Standard** Option.

When the PARTIAL_EVAL compiler option is ON, the compiler produces code that determines the value of each Boolean expression by evaluating the minimum number of operands, from left to right. When the PARTIAL_EVAL option is OFF, the compiler produces code that evaluates every operand of each Boolean expression in an implementation dependent order.

Syntax

```
$PARTIAL_EVAL {ON }$
              {OFF}
```

Default ON.

Location Statement.

The advantages of partial evaluation are more readable source code and more efficient object code.

Examples

```
$PARTIAL_EVAL OFF$
IF (index IN [lower..upper]) AND
   (ptr_array[index] <> NIL) AND
   (ptr_array[index]^ = 5) THEN ...
```

In this first example, if *index* is out of range, then ptr_array[index] causes a run-time error. If index is valid but ptr_array[index] is nil, then ptr_array[index]^ causes a run-time error.

```
$PARTIAL_EVAL ON$
IF (index IN [lower..upper]) AND
   (ptr_array[index] <> NIL) AND
   (ptr_array^ = 5) THEN ...
```

In this second example, if index is out of range, then (ptr_array[index] <> nil) is not evaluated. If ptr_array[index] is nil then (ptr_array^ = 5) is not evaluated.

```
$PARTIAL_EVAL OFF$
IF (index IN [lower..upper] THEN
   IF (ptr_array[index] <> NIL) THEN
      IF (ptr_array[index]^ = s) THEN ...
```

This third example is equivalent to the second example.

POP

POP is an **HP Pascal** Option.

The POP compiler option restores the compiler option settings that the last PUSH option saved (with the exceptions listed below.)

Syntax

```
$POP$
```

Default Not applicable.

Location Anywhere.

Compiler options with the location "At front" are not affected by POP.
The following compiler options are not affected by POP either:

ALIAS	GLOBAL	PUSH
COPYRIGHT	IF	SKIP_TEXT
ELSE	INCLUDE	SUBPROGRAM
ENDIF	LOCALITY	SYSINTR
EXTERNAL	PAGE	TITLE
FONT	POP	

Example

```
{Include file for supporting types.}

$PUSH, LIST OFF$

{Do not list the supporting types.
 To preserve the LIST state (ON or OFF) that this program set,
 save it first}

TYPE
   bit1 = 0..1;
   bit2 = 0..2;
   bit3 = 0..7;
     .
     .
     .
   bit16 = 0..32767;

$POP$

TYPE
   posshortint = bit16;
      .
      .
      .
```

PUSH

PUSH is an **HP Pascal** Option.

The PUSH compiler option saves the current compiler option settings.

Syntax

```
$PUSH$
```

Default Not applicable.

Location Anywhere.

The PUSH option can execute 15 times before the POP option must execute.

Example

```
{Include file for supporting types.}

$PUSH, LIST OFF$

{Do not list the supporting types.
 To preserve the LIST state (ON or OFF) that this program set,
 save it first}

TYPE
   bit1 = 0..1;
   bit2 = 0..2;
   bit3 = 0..7;
     .
```

```
          .
          .
       bit16 = 0..32767;

    $POP$

    TYPE
       posshortint = bit16;
          .
          .
          .
```

RANGE

RANGE is an **HP Standard** Option.

When the RANGE compiler option is ON, the compiler generates range-checking code for assignments, array indices, parameter passing, extensible parameters, pointers, CASE statements, and set operations. If a range check fails, an error message is issued and the program aborts (or causes an *escape* to be executed if a TRY-RECOVER construct is active).

The command line option +R also specifies this option.

Syntax

```
$RANGE {ON }$
       {OFF}
```

Default ON.

Location Statement.

NOTE Even when RANGE is ON, the compiler generates as little range-checking code as possible. If it can determine that a value can never be out of range at run time, it does not generate range-checking code for that variable.

RLFILE

RLFILE is a **System-Dependent MPE/iX** Option.

When the RLFILE compiler option is ON, every level-one routine goes into its own object module in the RL file. (Routines nested within level-one routines go into the same object module as the level-one routine in which they are nested.)

Syntax

```
$RLFILE {ON }$
        {OFF}
```

Default OFF.

Location At front.

When RLFILE is OFF (the default), the entire compilation unit goes into one object file. If the object file is an existing RL file, the entire compilation is placed into it. If the object file is an existing NMOBJ file, the object file is rewritten. If the object file is neither an RL nor an NMOBJ file, an error occurs. If the object file does not exist, the system creates an NMOBJ file with the specified name.

When RLFILE is ON, the compilation unit goes into an RL file procedure by procedure. This allows procedural-level manipulation similar to that on MPE V. An error occurs if the object file exists, but is not an RL file (that is, if it is an NMOBJ file). If the object file is an existing RL file, object modules replace existing modules in the RL file. If the object file does not exist, an RL file is created with the specified name.

When RLFILE is ON, the RL file can be significantly larger than if the program were compiled into an NMOBJ file, due to the duplicate information in each level-one object module. If SYMDEBUG is also ON, the RL file is even larger, because debug information is duplicated in each level-one object module if a local variable is declared using a global type.

Example

```
$RLFILE ON$
PROGRAM prog;
   .
   .
   .
```

NOTE If you use Pascal modules, all procedures and data in a particular module are put into one object module.

RLINIT

RLINIT is a **System-Dependent MPE/iX** Option.

The RLINIT compiler option initializes an RL file to empty.

Syntax

```
$RLINIT$
```

Default None.

Location At front.

The RLINIT compiler option initializes an RL file to empty before placing any object code in it. If RLINIT is not used, the compiler appends the new object code to any code that is already in the RL file. If $OLDPASS is used, or no file with the specified name exists, the system creates an RL file. If the specified object file is not an RL file (that is, if it is an NMOBJ file), an error occurs.

S300_EXTNAMES

S300_EXTNAMES is an **HP Pascal** Option.

The S300_EXTNAMES compiler option specifies that the external names of procedures in modules are of the form *modulename_procedurename*.

Syntax

```
$S300_EXTNAMES {ON }$
               {OFF}
```

Default OFF.

Location Before the EXPORT part of a module.

The S300_EXTNAMES option tells the linker to use the external name

modulename_procedurename instead of *procedurename* when linking a program. The name *modulename_procedurename* is in lowercase letters (as far as the linker is concerned) unless the procedure was compiled with the compiler option UPPERCASE ON.

The S300_EXTNAMES option applies to the entire module, but not to other modules in the same compilation unit. If a compilation unit contains several such modules, each one must contain the S300_EXTNAMES option.

The purpose of this option is to allow non-Pascal source code that calls external procedures that are in a Pascal module to be ported from, or be common with, HP9000 Series 300 source code, without changing the source code. The HP9000 Series 300 prefixes *modulename_* to *procedurename* in forming the link name; HP Pascal does not.

Example

```
MODULE M1;
$S300_EXTNAMES ON$
EXPORT
   VAR
      V1 : INTEGER;
      PROCEDURE P1 (P : CHAR);
IMPLEMENT
   .
   .
   .
END;

MODULE M2;
EXPORT
   VAR
      V2 : INTEGER;
      PROCEDURE P2 (P : INTEGER);
IMPLEMENT
   .
   .
   .
END;
```

The external names for V1, P1, V2, and P2 are M1_V1, M1_P1, V2, and P2, respectively.

SEARCH

SEARCH is an **HP Pascal** Option.

The SEARCH compiler option specifies one or more files for the compiler to search for module definitions. The files can be:

* Created with the MLIBRARY compiler option.
* Object files into which the modules were compiled (without the MLIBRARY compiler option).
* Archives (.a files) of such object files. On MPE/iX, these are RL files created by the Link Editor using such object files.

You must use the SEARCH option when a module being imported is not defined within the same compilation unit as the IMPORT statement.

Syntax

```
$SEARCH string [, string ]...$
```

Parameter

string Has value of the form:

'[+]*file_name* [, *file_name*]...'

The compiler searches the *file_name* s (in the order specified) for module definitions. If + is specified, the compiler concatenates this list of file names to the existing list (which was created by previous SEARCH options). If + is not specified, this list of file names replaces the existing list. (Note that + can only appear before the first string.)

An empty string resets the search list to the default.

Default On MPE/iX: PASLIB.PUB.SYS
On HP-UX: /usr/lib/paslib
Module definitions for predefined modules are kept in the system default module library (paslib), so you do not need to specify the search options for these modules.

Location Anywhere before the *import* statement.

Pascal requires that lower level modules be included in the $SEARCH path, even if the higher level modules do not use them. For example:

```
module a          $search 'a.o'$        $search 'a.o, b.o'$
export            module b              module c
  .               import a              import b
  .               export                export
  .                 .                     .
end.                .                     .
                    .                     .
                  end.                  end.
```

Example

```
$SEARCH 'file1,file2','file3'$    {The search list contains file1, file2, file3.}
$SEARCH '+file4'$                 {Adds file4 to the search list.}
IMPORT                            {The search list now contains file4.}
   MOD1,MOD2,MOD3;
      .
      .
      .
$SEARCH 'file5,file6',            {Replaces old search list.}
        'file7,file8'$            {Can span more than one line.}
<rev begin>
IMPORT MOD4;                      {The search list now contains only file5,}
   .                              {file6, file7, file8.}
<rev end>
   .
   .
```

The SEARCH compiler option tells the compiler which files to search for module definitions. It does not indicate to the linker which files should be linked with the main program. All object files, or archives and object files that appear in any search options in the main program must be explicitly specified to the linker at link time.

Example

```
Program main(input,output);
$SEARCH ‘a.o,b.o,c.o'$       {All object files for lower level}
                             {modules must be included.       }
import c;
  .
  .
  .
end.
```

The object files a.o, b.o, and c.o must be specified to the linker for the example program to be successfully linked.

SET

SET is an **HP Pascal** Option.

The SET compiler option assigns a Boolean value (TRUE or FALSE) to each of one or more identifiers that appear in subsequent IF options.

Syntax

```
$SET identifier =Boolean  [{,} identifier =Boolean ]'$
                      [{;}                    ]
```

Parameters

identifier Appears in an IF option later in the program. The *identifier* cannot be AND, OR, or NOT.

Boolean The value TRUE or FALSE (the compiler is not case-sensitive).

Default Not applicable.

Location Anywhere.

Example

The following two program fragments are equivalent:

```
     {Fragment 1}
     $SET 'group1=true, group2=false'$
        .
        .
        .
     $IF 'group1 AND (NOT group2)'$

[source_line ]
[ .         ]
[ .         ]
[ .         ]
     $ENDIF$

     {Fragment 2}
     $SET 'group1 = true'$
     $SET 'group2 = false'$
        .
        .
        .
     $IF 'group1'$
        $IF 'NOT group2'$

 [source_line ]
 [ .         ]
 [ .         ]
 [ .         ]

        $ENDIF$
     $ENDIF$
```

SHLIB_CODE

SHLIB_CODE is a **System_Dependent HP-UX** Option.

The compiler option SHLIB_CODE causes the compiler to generate position independent code (PIC) for use in shared libraries.

The command line options +Z and +z also specify this option.

Syntax

```
            {integer }
$SHLIB_CODE {ON     }$
            {OFF    }
```

Parameters

integer Must be in the range 0..2.

Value	Compiler generates:
0	Position dependent code.
1	Short load sequence PIC.
2	Long load sequence PIC.

ON Compiler generates short load sequence PIC.

OFF Compiler generates position dependent code.

Default OFF (Position dependent code).

Location At front.

Command Line Option +z or +Z

Programs that have been linked to shared libraries (.sl) use less disk space than those linked to archive libraries (.a). Also, programs linked to shared libraries get automatic updates when a new version of the shared library is installed.

When compiling for a shared library use $SHLIB_CODE ON$ or $SHLIB_CODE 1$. However, if the number of external references in the resulting shared library exceeds a system-dependent limit, use $SHLIB_CODE 2$. The linker will indicate when the limit has been exceeded.

SHLIB_CODE is valid only when the target operating system is HP-UX (see "OS"). The resulting object file will link only HP-UX.

For more information about shared libraries and PIC, refer to *Programming on HP-UX*.

SHLIB_VERSION

SHLIB_Version is a **System-Dependent HP-UX** Option.

The compiler option SHLIB_VERSION causes the compiler to place a shared library version string into the resulting object file.

$SHLIB_VERSION 'string'$

Parameters

String Specifies the date stamp to be used by the linker for shared library version control. Must be in the form: mm/yy or mm/yyyy.

Default '01/1990'.

Location At front.

Example

```
$OS ‘HPUX'$
$SHLIB_CODE ON$
$SHLIB_VERSION ‘04/1990'$
```

SHLIB_VERSION is designed to be used with the SHLIB_CODE compiler option. For more information about shared library version control, refer to *Programming on HP-UX*.

SKIP_TEXT

SKIP_TEXT is an **HP Pascal** Option.

The compiler ignores everything between $SKIP_TEXT ON$ and $SKIP_TEXT OFF$.

Syntax

```
$SKIP_TEXT {ON }$
           {OFF}
```

Default OFF

Location Anywhere.

Example

```
PROGRAM show_skiptext (output);
BEGIN
   writeln('This will print.');
   $SKIP_TEXT ON$
   writeln('This won''t print.');
   $SKIP_TEXT OFF$
   writeln('This will print.');
END.
```

The preceding program prints:

```
This will print.
This will print.
```

There is one exception to how SKIP-TEXT works. Symbols that begin a comment ({ or (*) are recognized and cause text to be commented out until a closing comment symbol (} or *)) is encountered.

Example

```
        0     1.000   0   PROGRAM show_skiptext_exception (output);
        0     2.000   1   BEGIN
        0     3.000   1      writeln('This will print.');
        1     4.000   1   $SKIP_TEXT ON$
       **     5.000   1      (* This unclosed comment causes the following option
       **     6.000   1         to be considered part of the comment:
       **     7.000   1   $SKIP_TEXT OFF$
       **     8.000   1      writeln('This will not print because the ');
       **     9.000   1      writeln('"skip_text off" option was commented out.');
       **    10.000   1       Comment is closed on the next line.
       **    11.000   1      *)
        1    12.000   1   $SKIP_TEXT OFF$
        1    13.000   1      writeln('This also will print.');
        2    14.000   1   END.
```

Output:

```
This will print.
This also will print.
```

SPLINTR

SPLINTR is an **HP Pascal** Option.

The SPLINTR compiler option specifies the intrinsic file that the compiler searches for information on intrinsic routines. It is the same

as the SYSINTR compiler option and is provided only for backward compatibility with Pascal/V.

Syntax

```
$SPLINTR [string ]$
```

Parameter

string Specifies the name of the intrinsic file that the compiler must search for information about intrinsic routines. This intrinsic file must be in SYSINTR format, not SPLINTR format (see Table 12-2 in "SYSINTR").

Default System intrinsic file (see the *HP Pascal/iX Programmer's Guide* or the the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation).

Location Anywhere.

Example

See the example for the SYSINTR compiler option.

NOTE The *pc* option +C on HP-UX affects the SPLINTR compiler option (see the *HP Pascal/HP-UX Programmer's Guide*).

STANDARD_LEVEL

STANDARD_LEVEL is an **HP Standard** Option.

The STANDARD_LEVEL compiler option specifies the level of syntax that the compiler routinely processes. The compiler issues a warning if it encounters a language feature that is illegal at that level. The compiler compiles the illegal feature if possible; otherwise, it is a syntax error.

Syntax

```
                 {ANSI      }
                 {ISO       }
$STANDARD_LEVEL '{HP_PASCAL }'$
                 {HP_MODCAL }
                 {EXT_MODCAL}
```

Parameters

ANSI Allows only ANSI Pascal.

ISO Allows only ISO Pascal (and ANSI Pascal).

HP_PASCAL Allows only HP Pascal (and ISO Pascal).

HP_MODCAL Allows HP Pascal and some system programming extensions.

EXT_MODCAL Allows HP Pascal and all system programming extensions.

Default HP_PASCAL.

Location Anywhere.

The HP Standard specifies the STANDARD_LEVEL compiler option only with the standard levels ANSI, ISO, and HP_PASCAL. HP Pascal accepts the additional standard levels HP_MODCAL and EXT_MODCAL.

A standard level violation (use of a language feature that is not available at the current standard level) causes the compiler to issue a warning, except if the violation involves a reserved word, in which case it is an error.

NOTE The STANDARD_LEVEL compiler option also accepts the Pascal/V standard levels 'HP' and 'HP3000', which it treats like 'HP_PASCAL'.

Example

```
$STANDARD_LEVEL 'ANSI'$  {equivalent to $ANSI ON$}
$OS 'MPE'$
PROGRAM show_level (output);
PROCEDURE proc1;
VAR
   i : integer;
   b : Boolean;
BEGIN
   assert(b,i);
     ^
**** WARNING #1  THIS FEATURE REQUIRES $STANDARD_LEVEL 'HP_PASCAL'$ (539)
  i := 0;
  b := true;
END;

BEGIN
END.
```

Figure 12-4 illustrates the relationship between the STANDARD_LEVEL parameters.

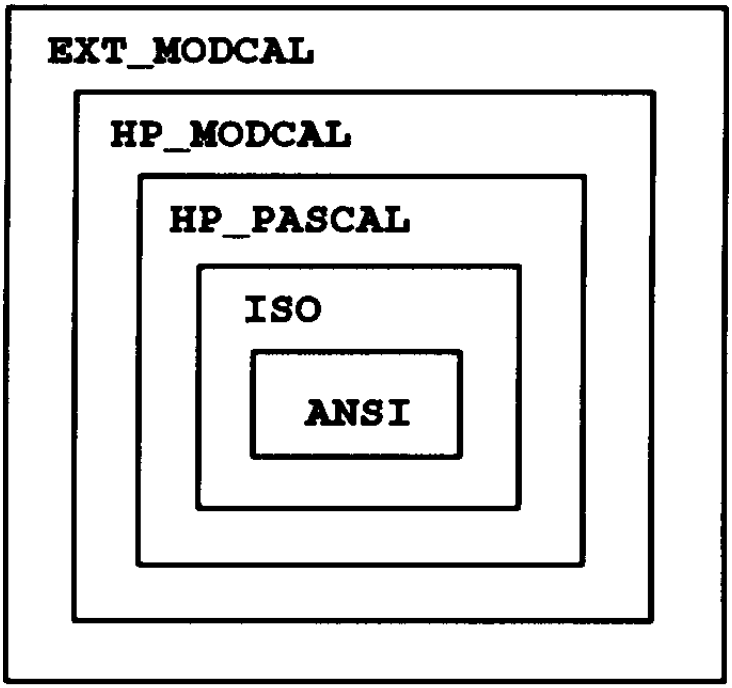


Figure 12-4. Relationships Between STANDARD_LEVEL Compiler Option Parameters

STATEMENT_NUMBER

STATEMENT_NUMBER is an **HP Pascal** Option.

When the STATEMENT_NUMBER compiler option is ON, the compiler generates a special instruction to identify a code sequence with its corresponding Pascal statement.

Syntax

```
$STATEMENT_NUMBER {ON }$
                  {OFF}
```

Default OFF

Location Anywhere.

The special instruction that the compiler generates is a LoaD Immediate Left (LDIL) instruction with destination register R0. It is equivalent to a No OPeration (NOP) instruction. The immediate field contains the statement number. When the debugger displays the mnemonic for the instruction, it shows the statement number instead of the LDIL instruction.

NOTE The STATEMENT_NUMBER compiler option is ignored when optimization is in effect.

Example

```
$STATEMENT_NUMBER ON$
$LIST_CODE ON$
PROGRAM a (output);
BEGIN
   writeln('Hi, mom!');
END.
```

The listing for the preceding program is:

```
 0       1.000    0    $statement_number on$
 0       2.000    0    $list_code on$
 0       3.000    0    program x;
 0       4.000    0    var
 0       5.000    0       i,j,k : integer;
 3       6.000    1    begin
 3       7.000    1       i := 0;
 4       8.000    1       j := i;
 5       9.000    1       k := j + i;
 6      10.000    1    end.

                                    PROGRAM

 0        STW     2,_20(0,30)                34       LDW     12(0,27),31
 4        LDO     48(30),30                  38       LDW     16(0,27),19
 8        STW     0,-4(0,30)                 3C       ADDO    31,19,20
 C        BL      P_INIT_ARGS,2              40       STW     20,8(0,27)
10        NOP                                      00002711
14        BL      U_INIT_TRAPS,2             44       BL      P_TERMINATE,2
18        NOP                                48       NOP
1C        *** Stmt    3                      4C       NOP
20        STW     0,16(0,27)                 50       BL      U_EXIT,2
24        *** Stmt    4                      54       NOP
28        LDW     16(0,27),1                 58       LDW     -68(O,30),2
2C        STW     1,12 (0,27)                5C       BV      0(2)
30        *** Stmt    5                      60       LDO     -48(30),30
```

STDPASCAL_WARN

STDPASCAL_WARN is an **HP Pascal** Option.

The STDPASCAL_WARN compiler option allows you to compile and execute syntax, which otherwise would have been issued an error message due to non-conformity with the ANSI/ISO standard.

A new error message can now be issued for syntax in HP Pascal that does not conform to the ANSI/ISO standard. This message may be issued only when one of the following compiler options is specified: ANSI ON, STANDARD_LEVEL 'ANSI', or STANDARD_LEVEL 'ISO'. In order to have a warning issued rather than this error message, specify the compiler option STDPASCAL_WARN.

Syntax

```
$STDPASCAL_WARN {ON }$
                {OFF}
```

Location Anywhere.

Default OFF.

Examples

```
$STANDARD_LEVEL 'ANSI'$
PROGRAM p;
VAR
   lr : longreal;
         ^
**** ERROR #1 THIS FEATURE DOES NOT CONFORM WITH THE ANSI/ISO STANDARD (044)
BEGIN
END.
```

In this example, an error message is issued because PROGRAM p does not conform to ANSI/ISO standard Pascal.

```
$STDPASCAL_WARN ON$
$STANDARD_LEVEL 'ANSI'$
PROGRAM p;
VAR
   lr : longreal;
         ^
**** WARNING # 1 THIS FEATURE REQUIRES $STANDARD_LEVEL 'HP_PASCAL' (539)
BEGIN
END.
```

In this example, STDPASCAL_WARN is specified so a warning is issued instead of an error message.

STRINGTEMPLIMIT

STRINGTEMPLIMIT is an **HP Pascal** Option.

The STRINGTEMPLIMIT option causes all temporary strings of unknown size to be allocated a fixed maximum size. Instead of being allocated in the heap, the temporary string is allocated in the stack.

Syntax

```
$STRINGTEMPLIMIT integer $
```

Parameters

integer Maximum size in bytes of any string temporary that the compiler can not calculate at compile time.

NOTE This value must include the length word of the string and any padding.

Default 0 (each request is allocated from the heap with the exact size required).

Location Heading.

Example

The following example shows two cases where the compiler can not determine the size of a string at compile time. For performance reasons, the temporary string is allocated in the stack at a fixed maximum of 400 bytes.

```
$STRINGTEMPLIMIT 400$
program strings;
type
   str80 = string[80];
var
   s1 : str80;
   n  : integer;

   function f(var x : string) : str80;
      begin
         f := x + ':';         { size of x is unknown }
      end;

begin
   n  := 40;
   s1 := strrpt('*',n);  { value of n is unknown }
   s1 := f(s1);
end.
```

SUBPROGRAM

SUBPROGRAM is an **HP Pascal** Option.

The SUBPROGRAM compiler option causes the compiler to emit code for specified level-one routines only. This includes routines nested within those routines, but not the outer block.

Syntax

$SUBPROGRAM ['*pfname* [*] [,*pfname* [*]]...']$

Parameters

pfname Name of a level-one routine. The compiler emits code for *pfname* and the routines nested within it, but not for the outer block. If no *pfname* s are specified, or they are entirely blank, the compiler compiles every routine, but not the outer block.

* Causes the compiler to compile the immediately preceding *pfname* with the LIST, CODE, and TABLES options ON. (Subsequent LIST, CODE, and TABLES options override *.)

Default All level-one routines.

Location At front.

A compilation unit can contain more than one SUBPROGRAM option. The following are equivalent:

```
     $SUBPROGRAM 'Proc1,Proc2'$
```

and

```
     $SUBPROGRAM 'Proc1'$
     $SUBPROGRAM 'Proc2'$
```

The SUBPROGRAM option enables you to compile selected routines of a program. The compiler checks the syntax and semantics of the entire program, but generates object code for the selected routines only.

Example

```
          0    1.000   0
          0    2.000   0
          0    3.000   0   $SUBPROGRAM 'proc2#, proc3#'$
          0    4.000   0   PROGRAM show_subprogram (output);
          0    5.000   0
          0    6.000   0   PROCEDURE proc1 (p : integer);
          2    7.000   1   BEGIN
          2    8.000   1      writeln(p);
          3    9.000   1   END;
          3   10.000   0
          0   11.000   0   PROCEDURE proc2 (p : integer);
          2   12.000   1   BEGIN
          2   13.000   1      writeln(p);
          3   14.000   1   END;
          3   15.000   0

                             I D E N T I F I E R   M A P

              IDENTIFIER      CLASS        TYPE       ADDRESS/VALUE

              P               PARAMETER    INTEGER    PSP-24.0 (4.0)

              LOCAL STORAGE USED  = 0.0      TEMPORARY STORAGE USED = 0.0
              PARAMETER STORAGE USED  = 4.0  CONSTANT STORAGE USED = 0.0

          0   17.000   0   PROCEDURE proc3 (p : integer);
          2   18.000   1   BEGIN
          2   19.000   1      writeln(p);
          3   20.000   1   END;
          3   21.000   0

                             I D E N T I F I E R   M A P

              IDENTIFIER      CLASS        TYPE       ADDRESS/VALUE

              P               PARAMETER    INTEGER    PSP-24.0 (4.0)

              LOCAL STORAGE USED  = 0.0      TEMPORARY STORAGE USED = 0.0
              PARAMETER STORAGE USED  = 4.0  CONSTANT STORAGE USED = 0.0

          0   22.000   1   BEGIN
          0   23.000   1   END.
```

SYMDEBUG

SYMDEBUG is a **System-Dependent MPE/iX** and **HP-UX** Option.

The SYMDEBUG compiler option emits symbolic debugging information for use with the HP TOOLSET/XL debugger or the HP Symbolic Debugger (see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*, depending on your implementation, for more information). You cannot use the optimizer if you use the SYMDEBUG option.

The command line option g also specifies this option.

Syntax

```
$SYMDEBUG ['XDB     ]$
          ['TOOLSET']
```

Parameters

None	The HP Symbolic Debugger for the HP-UX operating system; HP TOOLSET/XL for the MPE/iX operating system.
XDB	Emits information for the HP Symbolic Debugger, for either the HP-UX or MPE/iX operating system.
TOOLSET	Emits information for the HP TOOLSET/XL debugger. (Only the MPE/iX operating system allows HP TOOLSET/XL.)

Default None.

Location At front.

Example

```
$SYMDEBUG 'XDB'$
PROGRAM any_program;
BEGIN
  .
  .
  .
END.
```

NOTE A program containing the SYMDEBUG compiler option must be linked with the *pc* option -g.

SYSINTR

SYSINTR is an **HP Pascal** Option.

The SYSINTR compiler option specifies the intrinsic file that the compiler searches for information on intrinsic routines.

Syntax

```
$SYSINTR [string ]$
```

Parameter

string	Specifies the name of the intrinsic file that the compiler must search for information about intrinsic routines.
Default	System intrinsic file (see the *HP Pascal/iX Programmer's Guide* or the *HP Pascal/HP-UX Programmer's Guide*.)
Location	Anywhere.

NOTE The *pc* option +C on HP-UX affects the SYSINTR compiler option (see the *HP Pascal/HP-UX Programmer's Guide*.)

Example

```
PROGRAM Show_Intrinsic (Input,Output);
```

```
TYPE
   LogArray = ARRAY [1..80] OF shortint;

PROCEDURE FCheck;
INTRINSIC;          {FCheck comes from the system intrinsic file}

$SYSINTR 'MYINTR'$

PROCEDURE FWrite;
INTRINSIC;          {FWrite comes from MYINTR}

$SYSINTR$
FUNCTION FRead (    FileNum : shortint;
                VAR Target  : LogArray;
                    TCount  : shortint) : shortint;

INTRINSIC;    {This FRead description is compared to the one
               in the system intrinsic file.}
```

Table 12-2 compares SPLINTR (SPL) and SYSINTR (HP Pascal) formats. Neither format can be converted to the other automatically. For instructions on conversion by hand, see the *HP Pascal/iX Migration Guide* or the *HP Pascal/HPUX Migration Guide*.

Table 12-2. SPLINTR Format vs SYSINTR Format

	Pascal/V	**HP Pascal in Native Mode**
Creation	BUILDINT utility (independent of Pascal/V)	BUILDINT compiler option (in HP Pascal)
Result	SPLINTR (SPL) format file	SYSINTR (HP Pascal) format file
Access	$SPLINTR '*file* '$, where *file* is in SPLINTR (SPL) format	$SYSINTR '*file* '$, where *file* is in SYSINTR (HP Pascal) format

SYSPROG

SYSPROG is an **HP Pascal** Option.

The SYSPROG compiler option is equivalent to $STANDARD_LEVEL 'EXT_MODCAL'$ (see "STANDARD_LEVEL" in this chapter). It provides compatibility with Pascal on the HP 9000 Series 300 and 400 machines.

Syntax

```
$SYSPROG {ON }$
         {OFF}
```

Default OFF.

Location Heading.

Example

```
$SYSPROG ON$
PROGRAM machine_dependent;
   .
   .
   .
```

TABLES

TABLES is an **HP Pascal** Option.

When the TABLES compiler option is ON (and the LIST option is also ON), the listing includes an identifier map for each compilation block.

Syntax

```
$TABLES {ON }$
        {OFF}
```

Default OFF.

Location Anywhere.

In order for the listing to contain a table of a specific compilation block, the TABLES and LIST options must be ON when the compiler finishes parsing that block.

The table for a compilation block shows each identifier that the block declares and its class, type, and address or constant value. This information helps you debug your program.

The information in a table is arranged in four columns, as follows:

Col.	**Content**
1	Alphabetical list of the identifiers accessible to the current compilation block. If an identifier is the name of a record type, its field names appear beneath it, indented.
2	The class of the identifier in column one. The classes of identifiers are: USER DEFINED, CONSTANT, VARIABLE, FIELD, FUNCTION, TAG FIELD, PARAMETER, and PROCEDURE. For nonlocal references, the classes NON LOC VAR, NON LOC PARM, and NON LOC FUNC are used for nonlocal variables, nonlocal parameters, and nonlocal function returns, respectively.
3	The type of the identifier in column one. The types of identifiers are: INTEGER, SHORT INTEGER, REAL, BOOLEAN, SUBRANGE, ENUMERATED, BIT16, LONGREAL, CHAR (character) VALUE, CHAR ARRAY, STRING LITERAL, ARRAY, RECORD, SET, FILE, and POINTER.
4	The address or constant value of the identifier in column one. Addresses of variables and parameters are of the form REG+*offset*, where *offset* has the format *byte_offset.bit_offset* (both *byte_offset* and *bit_offset* are hexadecimal). REG is one of these four values:

Value	**Meaning**
DP+	for global variables
SP-	for local variables
PSP-	for parameters
name	for global variables whose locations cannot be determined at compile time (for example, module globals and globals in GLOBAL/EXTERNAL compilation units). No offset is printed in this case.

The meanings of the four REG values are as follows.

Value **Meaning**

DP+

The offset is relative to the contents of the DP register (the "Data Pointer," register 27). This register points to the base of the global variables. Its value can be displayed in an assembly-level debugger.

SP-

The offset is a negative offset from the contents of the SP register (the "Stack Pointer," register 30). This register points to the top of the activation record of the currently executing routine. Its value can be displayed in an assembly-level debugger.

PSP-

The offset is a negative offset from the contents of the Stack Pointer (SP register) for the caller's frame (the "Previous Stack Pointer"). Its value can be displayed by stopping the program at the first instruction of the current routine and examining the contents of the SP register before it is incremented to accommodate the frame of the current routine.

name The compiler cannot determine the location of the variable at compile time. Instead, it generates a symbol in the object file for the variable, and the link editor resolves the references at link time.

On HP-UX, you can display the actual location of such a variable with the assembly-level debugger *adb*, which allows you to specify the variable by name (rather than by address.)

On MPE/iX, request that the link editor produce a symbol map of the program file with the command

listprog *programfile; data*

Function return values are indicated by the class FUNCTION and the "address" RETURN.

Nonlocal (neither local nor global) variables, parameters (of enclosing routines), and function returns (of enclosing functions) are indicated by the address LEVEL *n*, where *n* is the level of the routine that contains the declaration of the variable or parameter in question.

The address of a FIELD or TAG FIELD is in the format *offset @ length*, where *offset* is in the format *byte_offset.bit_offset*, and *length* is in the format *byte_length.bit_length*. The values *byte_offset, bit_offset, byte_length,* and *bit_length* are hexadecimal.

The ADDRESS/VALUE column that TABLES ON produces provides packing information.

Example

```
0    1.000   0   $TABLES ON$
0    2.000   0   PROGRAM show_map (input,output);
0    3.000   0   CONST
0    4.000   0      realnum = 19.9;
1    5.000   0      maxsize = 100;
2    6.000   0      title = 'Customer list';
```

```
 3    7.000   0   TYPE
 3    8.000   0      answer = (yes,no);
 4    9.000   0      rec = RECORD
 5   10.000   0         ch : char;
 6   11.000   0         CASE tag : answer OF
 7   12.000   0            yes : (message : PACKED ARRAY [1..20] OF char);
 8   13.000   0            no  : (i : integer);
 9   14.000   0      END;
 9   15.000   0   VAR
 9   16.000   0      customer : rec;
10   17.000   0
 0   18.000   0   PROCEDURE proc1 (VAR num : real);
 2   19.000   0   VAR
 2   20.000   0      debt : Boolean;
 3   21.000   0
 3   22.000   0   PROCEDURE subproc1;
 4   23.000   1   BEGIN
 4   24.000   1      IF debt THEN writeln;
 6   25.000   1   END;
                        I D E N T I F I E R   M A P

IDENTIFIER          CLASS         TYPE           ADDRESS/VALUE

DEBT                NON LOC VAR   BOOLEAN        LEVEL 1

LOCAL STORAGE USED  = 0         TEMPORARY STORAGE USED  = 0
PARAMETER STORAGE USED  = 0     CONSTANT STORAGE USED  = 0

 6   26.000   1   BEGIN
 6   27.000   1   END;

**** WARNING #  1 "DEBT" ACCESSED, BUT NOT INITIALIZED (535)

                        I D E N T I F I E R   M A P

IDENTIFIER          CLASS         TYPE           ADDRESS/VALUE
DEBT                VARIABLE      BOOLEAN        SP-  28.0 (1.0)
NUM                 PARAMETER     REAL           PSP- 24.0 (4.0)
SUBPROC1            PROCEDURE

LOCAL STORAGE USED  = 1         TEMPORARY STORAGE USED  = 0
PARAMETER STORAGE USED  = 4     CONSTANT STORAGE USED  = 0

 0   28.000   0   FUNCTION func1 : integer; EXTERNAL;
 0   29.000   0
10   30.000   1   BEGIN
10   31.000   1   END.

                        I D E N T I F I E R   M A P

IDENTIFIER          CLASS         TYPE           ADDRESS/VALUE

ANSWER              USER DEFINED  ENUMERATED
CUSTOMER            VARIABLE      RECORD         DP+   8.0 (18.0)
FUNC1               FUNCTION
INPUT               PARAMETER     FILE           input (248.0)
MAXSIZE             CONSTANT      INTEGER                 100
NO                  CONSTANT      ENUMERATED                1
OUTPUT              PARAMETER     FILE           output (248.0)
PROC1               PROCEDURE
REALNUM             CONSTANT      REAL           1.99000E+01
REC                 USER DEFINED  RECORD         MAX RECORD SIZE = C0 BITS
  CH                FIELD         CHAR VALUE     0.0 @ 1.0
  TAG               TAG FIELD     ENUMERATED     1.0 @ 1.0
    MESSAGE         FIELD         ARRAY          4.0 @ 14.0
    I               FIELD         INTEGER        4.0 @ 4.0
TITLE               CONSTANT      STRING LITERAL 'Customer list'
YES                 CONSTANT      ENUMERATED                0
```

```
GLOBAL STORAGE USED  = 18       TEMPORARY STORAGE USED  = 0
PARAMETER STORAGE USED  = 0     CONSTANT STORAGE USED  = 0
```

TITLE

TITLE is an **HP Pascal** Option.

The TITLE compiler option specifies the title to appear on subsequent pages of the listing. (The title appears next to the page number in the top left-hand corner of the page.)

Syntax

```
$TITLE string_literal $
```

Parameter

string_literal Exact title (the compiler distinguishes between uppercase and lowercase letters.) The empty string ('') restores the default title. The string literal ' ' specifies a blank title.

Default

```
                        {iX}
HP PASCAL/{UX} HP product_number.v.uu.ff  COPYRIGHT HEWLETT-PACKARD
CO. year date time
```

where *product_number* is 31502 for MPE/iX and 92431 for HP-UX.

Location Anywhere.

Example

```
PAGE   1 HEWLETT-PACKARD  ...  (C) HEWLETT-PACKARD CO. ...

    0    1.000   0   $TITLE 'Payroll Program'$
    0    2.000   0   $PAGE$

PAGE   2 Payroll Program

    0    3.000   0   PROGRAM show_title (output);
    0    4.000   0
    0    5.000   1   BEGIN
    0    6.000   1   END.
```

TYPE_COERCION

TYPE_COERCION is a **System Programming** Option.

The TYPE_COERCION compiler option determines the level of value type coercion that the compiler allows.

Syntax

```
                {NONE          }
                {CONVERSION    }
$TYPE_COERCION ' {STRUCTURAL    }'$
                {REPRESENTATION}
                {STORAGE       }
                {NONCOMPATIBLE }
```

Parameters

NONE Prevents type coercion.

CONVERSION — Permits value type coercion of ordinal and pointer types. This is the most useful and transportable form of type coercion.

STRUCTURAL — Permits coercion of any data type to any structurally compatible data type. (This is equivalent to renaming components. It is fully transportable.)

REPRESENTATION — Permits coercion of any data type to any representation-size compatible data type. Representation-size compatible types have identical *BitSizeof* values.

STORAGE — Permits any type coercion that does not extend the amount of storage accessed. The data type being coerced must have a *Sizeof* value less than or equal to the *Sizeof* value of the data type to which it is being coerced.

NONCOMPATIBLE — Permits coercion of any data item to any data type. **This coercion can be dangerous, and errors cannot be detected.**

Default NONE.

Location Anywhere.

See Chapter 11 for more information on type coercion.

Example

```
         0    1.000   0   $STANDARD_LEVEL 'HP_MODCAL'$
         0    2.000   0   PROGRAM show_type_coercion;
         0    3.000   0
         0    4.000   0   TYPE
         0    5.000   0      Rec1 = RECORD
         1    6.000   0             F1 : integer;
         2    7.000   0             F2 : integer;
         3    8.000   0      END;
         3    9.000   0      Arr1 = PACKED ARRAY [1..8] OF char;
         4   10.000   0
         4   11.000   0   VAR
         4   12.000   0      R : Rec1;
         5   13.000   0      A : Arr1;
         6   14.000   0
         6   15.000   1   BEGIN
         6   16.000   1
         6   17.000   1      R.F1 := 101;  R.F2 := 280;
         8   18.000   1
         8   19.000   1      $TYPE_COERCION 'Structural'$
         8   20.000   1
         8   21.000   1      A := Arr1(R); {illegal, not structurally compatible}
                                 ^
     **** ERROR #  1  COERCION REQUIRES $TYPE_COERCION 'REPRESENTATION'$ (809)
         9   22.000   1
         9   23.000   1      $TYPE_COERCION 'Representation'$
         9   24.000   1
         9   25.000   1      A := Arr1(R);
        10   26.000   1
        10   27.000   1   END.
```

UPPERCASE

UPPERCASE is an **HP Pascal** Option.

When the UPPERCASE compiler option is ON, the compiler upshifts all external names (names of routines and global variables), including aliases. When UPPERCASE is OFF, the compiler downshifts these names.

The LITERAL_ALIAS compiler option overrides the UPPERCASE compiler option in aliases.

Syntax

```
$UPPERCASE {ON }$
           {OFF}
```

Default OFF.

Location Anywhere, but if you want the compiler to upshift the program parameter names, then UPPERCASE must precede the program header.

Scope All subsequent external names. If program parameter names are to be upshifted, then UPPERCASE must precede the program header.

Example

```
$UPPERCASE ON$
PROCEDURE proc1;  {External name is "PROC1".}

PROCEDURE $ALIAS 'Proc2Name'$ proc2;  {External name is "PROC2NAME".}

$UPPERCASE OFF$
PROCEDURE proc3;  {External name is "proc3".}

PROCEDURE $ALIAS 'Proc4Name'$ proc4;  {External name is "proc4name".}
```

VERSION

VERSION is an **HP Pascal** Option.

The VERSION compiler option specifies a string for the compiler to put in the version identification area of the current object module. The purpose of VERSION is to allow you to include the version number of your code in this area.

Syntax

```
$VERSION string_literal $
```

Parameter

string_literal Any string of characters (including unprintable characters).

Default Not applicable.

Location Anywhere.

Example

```
PROGRAM prog;
$VERSION 'A.00.00'$
BEGIN
   .
   .
   .
END.
```

The compiler puts the string *A.00.00* in the version identification area of the object module that contains the program prog.

A compilation unit can have multiple VERSION compiler options.

VOLATILE

VOLATILE is an **HP Pascal** Option.

You can apply the VOLATILE compiler option to a variable to specify that the memory location associated with the variable may be modified by other processes. Using VOLATILE signals the optimizer that a specified variable must not reside in a register, but must always be updated.

Syntax

$VOLATILE$

Location The VOLATILE compiler option is allowed after the ":" in a VAR declaration. It is also allowed after a "^" in a pointer type or variable declaration.

Example

```
TYPE
   ptrtype = ^$VOLATILE$ rectype;

VAR
   intptrr : ^$VOLATILE$ integer;
   recvar  : $VOLATILE$ rectype;
```

WARN

WARN is an **HP Pascal** Option.

The WARN compiler option suppresses warning messages and notes.

The command line option -w also specifies this option.

Syntax

```
$WARN [ON ]$
      [OFF]
```

Default ON.

Location Anywhere.

If neither ON nor OFF is specified, ON is assumed, and warning messages and notes are issued.

Warning messages may indicate program bugs or faulty processing. Turning them off may cause these potential problems to go unreported.

Example

```
$WARN OFF$
```

WIDTH

WIDTH is an **HP Pascal** Option.

The WIDTH compiler option sets the number of columns of each source line that the compiler will read.

Syntax

$WIDTH *integer* $

Parameter

integer In the range 10..132.

Default 132.

Location Anywhere.

The WIDTH option allows the compiler to ignore text beyond a specified column.

The WIDTH option applies only to the file that contains it, and not to files that it includes (see the INCLUDE option). If File1 with width *n* includes File2, the width while File2 is being included is specified by File2 (if File2 does not contain a WIDTH option, the width defaults to 132.) At the end of File2, the width returns to *n*.

Example

```
          1         2         3         4
1234567890123456789012345678901234567890 <--- Column number guide

$WIDTH 30$
PROGRAM show_width (output);      The compiler ignores this text
BEGIN                             since it is beyond column 30.
   writeln('The width is 30');
   $INCLUDE 'File2'$
   writeln('The width is 30');
END.

File2:

$WIDTH 40$
writeln('The width is 40');       This text (31-40) is not ignored.
$WIDTH 20$
writeln('Hi');                    This text (beyond 20) is ignored.
```

XREF

XREF is an **HP Pascal** Option.

When the XREF compiler option is ON (and the LIST option is also ON), the listing includes a cross reference for each function, procedure, and outer block.

Syntax

```
$XREF {ON }$
      {OFF}
```

Default OFF.

Location Anywhere.

A cross reference lists each identifier that is accessible to the block. For each file that references the identifier, the cross reference shows the file name and gives the numbers of the lines on which the identifier is referenced. The symbol @ after a line number means that the identifier is declared on that line. The symbol * after a line number means that the value of the identifier is (or could be) changed on that line.

The line numbers are assigned by the editor (if the source file is numbered) or by the compiler (if the source file is unnumbered.) Lines from included files (see the INCLUDE option) are numbered independently (see the second example for the LIST option).

Although the XREF option is legal anywhere in the source code, it affects only the code that follows it. Therefore, its most practical location is the beginning of the source code.

Example

```
0    1.000   0   $XREF ON$
0    2.000   0   $TITLE 'Show_xref'$
0    3.000   0   PROGRAM show_xref (input,output);
0    4.000   0   $INCLUDE 'const'$
0    1.000   0   CONST
0    2.000   0      k = 100;
1    5.000   0   VAR
1    6.000   0      n : integer;
2    7.000   0      t : Boolean;
0    8.000   0   PROCEDURE check (VAR b : Boolean);
2    9.000   1   BEGIN
2   10.000   1      IF n > k THEN b := true
4   11.000   1      ELSE  b := false;
5   12.000   1   END;

                        C R O S S   R E F E R E N C E
                        ---------   -----------------

      Page   Line #   Page   Line #   Page   Line #   Page   Line #
B
    PXA32.EXAMPLES.ATFTEST

         1 00008.000@    1 00010.000*    1 00011.000*
BOOLEAN              global scope
    PXA32.EXAMPLES.ATFTEST

         1 00008.000
CHECK                global scope
    PXA32.EXAMPLES.ATFTEST

         1 00008.000
FALSE                global scope
    PXA32.EXAMPLES.ATFTEST

         1 00011.000
K                    global scope
    PXA32.EXAMPLE.ATFTEST

         1 00010.000
N                    global scope
    PXA32.EXAMPLES.ATFTEST

         1 00010.000
TRUE                 global scope
    PXA32.EXAMPLES.ATFTEST

         1 00010.000

3   13.000   1   BEGIN
3   14.000   1      readln(n);
4   15.000   1      check(t);
5   16.000   1      IF t THEN writeln ('Too big!')
7   17.000   1      ELSE writeln ('No Problem');
8   18.000   1   END.

PAGE   2  Show_xref

                        C R O S S   R E F E R E N C E
                        ---------   -----------------

      Page   Line #   Page   Line #   Page   Line #   Page   Line #
BOOLEAN
   PXA32.EXAMPLES.ATFTEST

         1 00007.000     1 0008.000
CHECK
```

```
   PXA32.EXAMPLES.ATFTEST

         1 00008.000     1 0015.000
FALSE
   PXA32.EXAMPLES.ATFTEST

         1 00011.000
INPUT
   PXA32.EXAMPLES.ATFTEST

         1 00003.000
INTEGER
   PXA32.EXAMPLES.ATFTEST

         1 00006.000
K
   PXA32.EXAMPLES.ATFTEST

         1 00010.000
   const

         1 00002.000
N
   PXA32.EXAMPLES.ATFTEST

         1 00006.000@    1 00010.000    1 00014.000*

OUTPUT
   PXA32.EXAMPLES.ATFTEST

         1 00003.000
READLN
   PXA32.EXAMPLES.ATFTEST

         1 00014.000
SHOW_XREF
   PXA32.EXAMPLES.ATFTEST

         1 00003.000
T
   PXA32.EXAMPLES.ATFTEST

         1 00007.000     1 00015.000*    1 00016.000

PAGE   3  Show_xref

                        C R O S S   R E F E R E N C E
                        ---------   -----------------

      Page   Line #   Page   Line #   Page   Line #   Page   Line #
TRUE
   PXA32.EXAMPLES.ATFTEST

         1 00010.000
WRITELN
   PXA32.EXAMPLES.ATFTEST

         1 00016.000     1 0017.000
```

Appendix A Error Messages

On HP-UX, error messages and their explanatory text are in the file named */usr/lib/paserrs*. To list this file, use the command:

```
cat usr/lib/paserrs
```

On MPE/iX, error messages and their explanatory text are in the file named *PASXLCAT.PUB.SYS*. To list this file, use the command:

```
:PRINT PASXLCAT.PUB.SYS
```

In reading the error messages, note that:

* A dollar sign ($) in the left margin indicates a comment line containing explanatory text.
* An exclamation mark (!) indicates that an item is variable. The compiler substitutes a specific item for the exclamation mark when it issues the message.

Example

```
043 THIS FEATURE REQUIRES $OS ! (043)
$      1. This feature is not available under the current OS level
060 OPERAND NOT OF TYPE BOOLEAN (060)
$      1. A non-Boolean operand appears with the operator NOT, OR,
$         or AND.
```

When the compiler issues error message 43, it will substitute an OS level for the exclamation mark; for example:

```
THIS FEATURE REQUIRES $OS 'HPUX'$
```

Finding Undetected Errors

The following errors are currently undetected by the compiler at compile time or by the system at run time. In any future release, an undetected error may become a detected error.

Errors that are only detected when the ANSI option is ON, or when STANDARD_LEVEL is set to ANSI, do not appear on this list.

There is no significance to the order in which errors are listed here.

1. Each component of a structured function result must be assigned a value in the body of the function.
2. If assignment to a function result is conditional, it must occur at run time.
3. A control variable in a FOR statement cannot be changed in the statement after DO by calling a procedure or function with a nonlocal reference to the variable.
4. A parameter of *dispose* cannot be an actual variable parameter, an element of a record variable list of a WITH statement, or both. Similarly, a dynamic variable in a region of the heap deallocated by *release* cannot fall in one of these categories.
5. When the tag field of a record with variants is changed, all previous variants become undefined.
6. For records with tagless variants, reference to a field for a particular variant means that other previous variants become undefined.
7. All possible record variants must be specified in a record declaration.

8. When a value is established for the tag field of a record with variants, it is illegal to use a field in another variant.

9. The compiler does not always detect uninitialized variables, especially in these cases:

   a. The path to use a variable cannot include the initializing statement. Suppose:

```
PROCEDURE proc_a;
VAR
   x,y : integer;
BEGIN
   IF condition  THEN x := 10 ELSE y := x;
      .
      .
      .
END;
```

      The assignment after ELSE does not cause a compile-time error, even if *x* has not been initialized outside the IF statement. (The compiler counts the assignment after THEN as initialization.)

   b. Not all the components of a record or array have been assigned values. (The compiler counts the assignment to a single component as initialization of the entire variable.)

   c. An uninitialized global variable appears in a program compiled with GLOBAL or EXTERNAL options, or in a program that contains procedures or functions declared with the EXTERNAL directive. (The compiler cannot check outside the current source code.)

   d. An uninitialized dynamic variable on the heap. (The compiler cannot detect this at run time.)

   e. *Strwrite* into an uninitialized string variable.

   However, some of the above errors are detected when the compiler option OPTIMIZE is ON.

10. An actual reference parameter cannot be an expression consisting of a single variable in parentheses.

11. Case constant labels cannot be constant expressions.

12. Range checking code is suppressed when the type of logical file is identical to the type of a variable to which a file component is assigned. However, a physical file associated with the logical file can have values out of range and the consequent errors are undetected.

13. Applying *put* to an undefined file buffer variable.

14. The control variable of a FOR statement is undefined after the execution of the FOR statement.

15. Dereferencing an undefined pointer is not always detected, especially for pointers that have never been explicitly disposed.

16. Using a variable created with the long form of *new* as an actual parameter.

17. Using a variable created with the long form of *new* in a assignment statement.

18. Using a variable created with the long form of *new* in a factor (for example, as an operand in an expression).

19. Altering the value of the record variable of a WITH statement within the scope of the WITH statement.

20. Using *put, dispose,* or *release* to make an actual variable parameter to a procedure undefined within the body of the procedure.

Using This Appendix

This appendix describes the errors, notes, and warnings that can be detected during the compilation or execution of an HP Pascal program. These errors are listed in numeric order.

The text of each message is followed by a brief explanation of the situation, the CAUSE. When it is necessary for the user to do something, there is an ACTION following the particular CAUSE. In some cases there may only be one action for several causes. Messages in the warnings and notes categories usually do not require actions.

Each message contains a code under its message number in the left column. This code indicates whether the message is a note (**N**), a warning (**W**), a compile-time error (**CT**), run-time error (**RT**), or an internal error (**I**). An exclamation point , "!", in the messages reproduced here is replaced in the actual message with appropriate text.

The error messages are grouped by number as follows:

Number Range	**Category**
Pascal Messages	
000 - 299	**CT** - Compile-time errors
300 - 399	**N** - Notes
400 - 499	**CT** - Compile-time errors
500 - 599	**W** - Warnings
600 - 799	**RT** - Run-time errors
800 - 899	**CT** - Compile-time errors
900 - 999	**RT** - Run-time errors
Code Generation Messages	
5000 - 5099	**W** - Warnings
5100 - 5199	**I** - Internal errors
5200 - 5399	**CT** - Compile-time errors
5400 - 5999	**I** - Internal errors
Optimizer Messages	
6000 - 6099	**W** - Warnings
6100 - 6199	**I** - Internal errors
6200 - 6399	**CT** - Compile-time errors
6400 - 6999	**I** - Internal errors
Code Generation Messages	
7000 - 7099	**W** - Warnings
7100 - 7199	**I** - Internal errors
7200 - 7399	**CT** - Compile-time errors
7400 - 7999	**I** - Internal errors

If there are previous syntax errors, the compiler will sometimes produce internal errors. Should this occur, correct the syntax errors and recompile. If you still receive internal errors, submit a service request.

NOTE When an error message says "contact Hewlett-Packard," please submit a service request (SR) and the appropriate source and object files. This allows Hewlett-Packard to duplicate the problem you are reporting.

Messages 001-200

---

001 MESSAGE FLOATING POINT OVERFLOW (001)

CT CAUSE The absolute value of a real number is too large.

ACTION Check the permitted range of real/longreal values.

---

002 MESSAGE FLOATING POINT UNDERFLOW (002)

CT CAUSE The absolute value of a real number is non-zero and too small.

ACTION Check the permitted range of real/longreal values.

---

003 MESSAGE ERROR IN FLOATING POINT NUMBER REPRESENTATION (003)

CT CAUSE The real or longreal number must have a digit after the decimal point.

ACTION Correct the constant to specify a fractional part.

---

004 MESSAGE AN EXPONENT IS REQUIRED HERE (004)

CT CAUSE The exponent for a real or longreal number is missing. A number is required after the 'E' or 'L'.

ACTION Correct the constant to specify an exponent.

---

005 MESSAGE ILLEGAL CONTROL CHARACTER CONSTANT (005)

CT CAUSE The value of the constant following the sharp (#) is greater than 255.

ACTION Check nonprinting character formation rules.

CAUSE The only nonnumeric characters that can follow a sharp (#) are a letter, @, [,], ||, ^, or _.

ACTION Check the permitted range of character values.

---

006 MESSAGE A QUOTE IS EXPECTED HERE (006)

CT CAUSE The end of line was found before the terminating quote. String literals cannot span source lines.

ACTION Check string constant for missing closing quote or shorten constant.

---

007 MESSAGE INTEGER OVERFLOW (007)

CT CAUSE The absolute value of the integer is greater than maxint.

ACTION Check the permitted range of integer values.

---

008 MESSAGE END OF FILE FOUND BEFORE EXPECTED (008)

CT CAUSE The compiler expects more source code. There may be an unmatched BEGIN-END or an unclosed comment.

ACTION Check for missing END, semicolon, period, or incomplete statement. Also check for an unclosed comment or $SKIP_TEXT ON$.

---

009 MESSAGE UNRECOGNIZED CHARACTER (009)

CT CAUSE An illegal character was found in the source.

ACTION Check for unprintable characters and character validity in context.

---

010 MESSAGE 100 ERRORS--PROGRAM TERMINATED (010)

CT CAUSE Only 100 errors are allowed before the compiler stops.

ACTION Correct earlier errors so that compilation can continue.

---

011 MESSAGE A COMMA IS REQUIRED HERE (011)

CT CAUSE A comma is needed to separate procedure/function names in the SUBPROGRAM compiler option.

ACTION Check syntax and insert a comma where necessary.

---

012 MESSAGE VARIABLE SPECIFICATION NOT ALLOWED HERE (012)

CT CAUSE Only SPL procedures are allowed to have a variable number of parameters.

ACTION Remove the keyword VARIABLE or declare the routine SPL VARIABLE.

---

013 MESSAGE IDENTIFIER DOUBLY DEFINED (013)

CT CAUSE An identifier in a parameter list is a duplicate of another identifier.

The procedure/function name is defined earlier and is not a FORWARD procedure/function.

The field name of a record is already declared.

The identifier is already declared in the current scope.

ACTION Delete duplicate declaration.

---

014 MESSAGE IDENTIFIER NOT DEFINED (014)

CT CAUSE The identifier is an undeclared variable, constant, procedure,

or function.

The type identifier is undeclared.

ACTION Add identifier to the declaration section.

---

015 MESSAGE INVALID VARIABLE USE (015)

CT CAUSE The control variable of a FOR loop is being modified in the statement component of the FOR loop. For example:

* It is the control variable of a nested FOR loop.
* It appears on the left side of an assignment statement.
* It is being passed by reference to a user-defined or standard procedure.

ACTION Remove assignment to loop control or conformant bound variable. Do not pass this variable as a VAR, ANYVAR, or READONLY parameter.

CAUSE The variable appears in the variable list of a WITH statement but is not a record type.

ACTION Remove the variable from the WITH list.

CAUSE The identifier appears with subscripts, but it is not an array or string.

ACTION Correct the array expression or remove the subscript.

---

016 MESSAGE TYPE IDENTIFIER REQUIRED HERE (016)

CT CAUSE A constant or variable identifier has been used where a type identifier is required.

ACTION Replace the constant or variable identifier with a type identifier.

---

017 MESSAGE INVALID TYPE IDENTIFIER USE (017)

CT CAUSE A type identifier has been used where a constant or variable identifier is required.

The construct in which the identifier occurs is not legal in this context. This is often an array or record in executable code.

ACTION Replace the type identifier with a constant or variable identifier.

---

018 MESSAGE A CONSTANT EXPRESSION IS REQUIRED HERE (018)

CT CAUSE A variable occurs where a constant is required.

An expression with variables occurs where a constant expression is required.

The expression contains an operator or a standard procedure or

		function that is not legal in a constant expression. The expression contains constant operands that are not legal; for example, set or Boolean values.
	ACTION	Check the constant expression for a variable, or illegal type of operand.
019	MESSAGE	INVALID FORWARD TYPE IDENTIFIER DEFINITION (019)
CT	CAUSE	The identifier appeared in a forward pointer type definition and is now being declared as something other than a type.
	ACTION	Check the FORWARD definition.
020	MESSAGE	BOOLEAN EXPRESSION IS REQUIRED HERE (020)
CT	CAUSE	An expression with a Boolean result is required here.
	ACTION	Check the source and correct the expression.
021	MESSAGE	AN ORDINAL EXPRESSION IS REQUIRED HERE (021)
CT	CAUSE	An expression with an ordinal result is required here.
	ACTION	Check the source and correct the expression.
022	MESSAGE	INCOMPATIBLE SUBRANGE BOUNDS (022)
CT	CAUSE	The type of the lower bound is not compatible with the type of the upper bound in a subrange.
	ACTION	Check the type of the lower and upper bounds and make them the same.
023	MESSAGE	AN INTEGER EXPRESSION IS REQUIRED HERE (023)
CT	CAUSE	An expression with an integer result is required for the repeat factor in the 'OF' construct in an array constructor.
	ACTION	Check the source code and correct the expression.
024	MESSAGE	LOWER BOUND OF SUBRANGE IS GREATER THAN UPPER BOUND (024)
CT	CAUSE	The lower bound is greater than the upper bound in a subrange type declaration.
	ACTION	Increase the upper bound, or decrease the lower bound.
025	MESSAGE	FOUND UNEXPECTED "! " (025)
CT	CAUSE	The compiler was not expecting this token and it has been discarded. The token is illegal here or a previous undetectable error has caused the compiler to issue this message; for example, a semicolon (;) before ELSE.

	ACTION	Remove "! " or correct earlier error.

026	MESSAGE	MISSING "! " (026)
CT	CAUSE	The compiler expected this token, but it was omitted or misspelled. The correct token was inserted.
	ACTION	Insert "! "

027	MESSAGE	"! " FOUND BEFORE EXPECTED. SOURCE MISSING. (027)
CT	CAUSE	The compiler found this token before it was expected. The compiler was able to accept it by inserting dummy tokens.
	ACTION	Correct the syntax error and recompile.

028	MESSAGE	MISUNDERSTOOD SOURCE BEFORE "! " (028)
CT	CAUSE	The compiler has discarded some previously accepted source code preceding this token. Either the token is inappropriate, but the compiler has been able to accept it by ignoring previous code, or the token is correct and code must now be discarded.
	ACTION	Check the source code and fix the syntax.

029	MESSAGE	" NOT ALLOWED AS A STRING LITERAL DELIMITER (029)
CT	CAUSE	A double quote cannot delimit a string literal.
	ACTION	Replace " with expected '.

030	MESSAGE	OPEN FAILED ON FILE "! " (030)
CT	CAUSE	The compiler could not open the source file. The compiler could not open the INCLUDE file. The compiler could not open the SYSINTR or SPLINTR file.
	ACTION	Check for the correct file name spelling, file existence, and any file equations.

031	MESSAGE	READ FAILED ON SOURCE FILE (031)
CT	CAUSE	The compiler could not read the source file. The compiler could not read the INCLUDE file.
	ACTION	Correct the condition causing the read to fail, such as a corrupted file or any internal compiler errors.

032	MESSAGE	EMPTY SOURCE FILE (032)
CT	CAUSE	The source file is empty.

	ACTION	Check the file name.

033	MESSAGE	MISSPELLED RESERVED WORD: "! " (033)
CT	CAUSE	The reserved word is misspelled.
	ACTION	Correct the spelling of the reserved word.

034	MESSAGE	FORWARD TYPE "! " NOT FOUND (034)
CT	CAUSE	The identifier occurs in a pointer type definition but is not subsequently defined.
	ACTION	Define the identifier.

035	MESSAGE	FORWARD PROCEDURE "! " NOT DECLARED (035)
CT	CAUSE	A procedure declared with the FORWARD directive is not subsequently defined. The definition may be missing, or the name appearing in the definition may be misspelled.
	ACTION	Declare the procedure.

036	MESSAGE	VIOLATION OF PASCAL SCOPING RULES (036)
CT	CAUSE	The scope of an HP Pascal identifier is the entire block in which it is declared. It is not possible to use an identifier from an enclosing level and then to redefine it at the new level.
	ACTION	Use a separate identifier in this text.

037	MESSAGE	INVALID USE OF "! " IN POINTER DEFINITION (037)
CT	CAUSE	A non-type identifier defined on a previous level was used in a pointer type definition.
	ACTION	Replace the non-type identifier with a type identifier.

038	MESSAGE	ILLEGAL PASCAL CONSTRUCT (038)
CT	CAUSE	The use of the FOR construct within strings is illegal.
	ACTION	Use another looping construct with strings.

039	MESSAGE	"! " ACCESSED, BUT NOT INITIALIZED (039)
CT	CAUSE	A simple variable appears in an expression, as a value parameter, or in some other accessing reference and it has never appeared in an assigning reference, such as a reference parameter, or on the left side of an assignment statement. Some component of a structured variable appears in an accessing reference but no component of that variable has yet appeared in

		an assigning reference.
	ACTION	Initialize the variable before it is used.

040	MESSAGE	INVALID STRING TYPE USE (040)
CT	CAUSE	The standard type identifier string is not used to define a string type.
	ACTION	Use the standard identifier string to define this type.

041	MESSAGE	MISSING SEPARATOR BETWEEN NUMBER AND IDENTIFIER (041)
CT	CAUSE	A character was detected immediately following a number. HP Pascal requires a separator, such as a space, comment, or end-of-line between a number and an identifier or reserved word.
	ACTION	Insert a separator between the number and the identifier.

042	MESSAGE	^STRING IS NOT ALLOWED IN TYPE DECLARATIONS (042)
CT	CAUSE	^STRING was used in a pointer type declaration. A user definition for STRING did not follow so an error was produced when the compiler checked for unresolved forward pointer declarations. The generic type STRING is only allowed for VAR parameters.
	ACTION	Remove use of string in type declaration.

043	MESSAGE	THIS FEATURE REQUIRES $OS ! (043)
CT	CAUSE	This feature is not available under the current OS level.

044	MESSAGE	THIS FEATURE DOES NOT CONFORM WITH THE ANSI/ISO STANDARD (044)
CT	CAUSE	This feature is not available under the current STANDARD_LEVEL.
	ACTION	Remove this feature if ANSI/ISO conformance is desired. Remove STANDARD_LEVEL compiler option if this feature is desired. Use the compiler option STDPASCAL_WARN if a warning message rather than an error message is desired with the current STANDARD_LEVEL that is set.

045	MESSAGE	ONLY COMMENTS AND COMPILER OPTIONS ARE ALLOWED IN '!' (045)
CT	CAUSE	Text which is neither a comment nor a compiler option was detected in the system-wide option file.
	ACTION	Remove any text which is neither a comment nor a compiler option from the system-wide option file. Because this file is write-protected, your system administrator should be notified.

---

060 MESSAGE OPERAND NOT OF TYPE BOOLEAN (060)

CT CAUSE A non-Boolean operand appears with the operator NOT, OR, or AND.

ACTION Change the operator to a Boolean type.

---

061 MESSAGE WRONG TYPE OF OPERAND FOR ARITHMETIC OPERATOR (061)

CT CAUSE A nonnumeric operand appears with an arithmetic operator.

ACTION Check and correct the operand or operator.

---

062 MESSAGE TYPE OF OPERAND NOT ALLOWED WITH OPERATOR (062)

CT CAUSE An operand of this type cannot be used with this operator.

ACTION Check and correct the operand or operator.

---

063 MESSAGE BASE TYPE OF OPERAND AND SET DO NOT AGREE (063)

CT CAUSE The operand on the left of an IN operator is not type compatible with the set on the right.

ACTION Check the operands to ensure compatible types.

---

064 MESSAGE TYPES OF OPERANDS DO NOT AGREE (064)

CT CAUSE The operands can be used separately but not with the operator. For example, <Boolean> = <integer>.

ACTION Check and correct one of the two operands.

---

065 MESSAGE ASSIGNMENTS CANNOT BE MADE TO FILES (065)

CT CAUSE An assignment cannot be made to a file or a structured variable with a file type component.

Structured constants cannot contain files. Building a structured constant with a type that contains a file is illegal.

Variables which contain files cannot be passed as value parameters.

ACTION Remove the file assignment.

---

066 MESSAGE ASSIGNMENT TYPE CONFLICT (066)

CT CAUSE The expression on the right side of an assignment statement is not assignment compatible with the receiving entity on the left.

A constant in a constructor is not assignment compatible with

		the component to which it is being assigned. The subrange type of the expression being assigned does not intersect the type of the receiving entity.
	ACTION	Check the assignment compatibility rules.

067	MESSAGE	TYPE OF EXPRESSION NOT ALLOWED IN SUBRANGE (067)
CT	CAUSE	The expression defining a subrange bound is not an ordinal expression.
	ACTION	Replace the expression with an ordinal expression.

068	MESSAGE	ILLEGAL ASSIGNMENT TARGET (068)
CT	CAUSE	An assignment was made to an identifier that is not a non-file variable or a function result; for example, a declared constant, a set, or string type identifier.
	ACTION	Correct the left-hand side of the assignment.

069	MESSAGE	INVALID CONSTANT EXPRESSION (069)
CT	CAUSE	This expression is not legal in a CONST declaration. It is not a constant expression, or it is a constant expression and the results of the arithmetic would be out of range of minint..maxint.
	ACTION	Correct the expression.

070	MESSAGE	ILLEGAL TO ASSIGN TO (070)
CT	CAUSE	The identifier denotes an entity that cannot appear on the right side of an assignment statement; for example, a set or string type identifier.
	ACTION	Correct the right-hand side of the assignment.

072	MESSAGE	REAL CONSTANT FOLDING NOT AVAILABLE IN $HP3000_16$ (072)
CT	CAUSE	Temporary restriction on real constant folding in $HP3000_16$. This is transparent, except when an integer value is specified for a real field in a structured constant declaration. This also occurs if a real constant is specified for a longreal constant .
	ACTION	Change the integer constant to a real one by appending ".0", or add "L0" to the real number.

080	MESSAGE	ARRAY INDEX TYPES NOT COMPATIBLE (080)
CT	CAUSE	The subscript in an array reference is not compatible with the type of the index in the array declaration.
	ACTION	Change the array subscript to be compatible with the type of the index.

---

081 MESSAGE ARRAY ELEMENT TYPES NOT EQUIVALENT (081)

CT CAUSE PACK and UNPACK array parameters must have identical component types.

ACTION Use identical component types.

---

082 MESSAGE INVALID ARRAY SIZE (082)

CT CAUSE The size of the array is too big for the compiler.

In PACK or UNPACK the destination array is not large enough.

ACTION Use a smaller array size.

---

083 MESSAGE WRONG NUMBER OF ELEMENTS FOR ARRAY OR STRING CONSTANT (083)

CT CAUSE While building an array or string constant, more components were specified than declared.

Not all the components were specified while building an array constant.

ACTION Use the correct number of components that need to be specified.

---

084 MESSAGE INVALID INDEX TYPE (084)

CT CAUSE Index type is not an ordinal type.

ACTION Use an ordinal type.

---

085 MESSAGE REFERENCE TYPE MUST BE STRING OR ARRAY (085)

CT CAUSE Tried to index a structure that is not an array or string.

ACTION Use an array or string in this context.

---

086 MESSAGE MAXIMUM STRING LENGTH MUST BE BETWEEN 1 AND ! (086)

CT CAUSE Tried to declare string with a maximum length < 1 or > the limit mentioned in the message.

ACTION Correct the string maximum length specification so it is in the permitted range.

---

087 MESSAGE EXPRESSION FOR MAXIMUM LENGTH MUST BE TYPE INTEGER (087)

CT CAUSE Tried to declare a string with a noninteger constant expression for the maximum length.

ACTION Use an integer constant in this context.

---

088 MESSAGE INCORRECT NUMBER OF INDICES FOR STRING DECLARATION (088)

CT CAUSE A string can only have one index in a declaration.

No index was supplied in a string declaration.

ACTION Use only one index in a string declaration.

---

089 MESSAGE TOO MANY SUBSCRIPTS IN STRING OR ARRAY REFERENCE (089)

CT CAUSE The number of subscripts in the reference exceeds the number of subscripts in the declaration of the array or string.

ACTION Correct the number of subscripts.

---

090 MESSAGE ILLEGAL CONSTRUCT FOR AN ARRAY OR STRING INDEX (090)

CT CAUSE A subrange construct was used as an array or string index.

ACTION Correct the subrange construct.

---

100 MESSAGE INVALID RECORD REFERENCE (100)

CT CAUSE Record field referenced without specifying a record variable, constant, or function call that returns a record.

ACTION Qualify the name completely (i.e., specify which record variable this is a field of).

---

101 MESSAGE INVALID FIELD IDENTIFIER (101)

CT CAUSE The identifier is not one of the fields of the record used in the reference.

ACTION Check the field name and the record type definition.

---

102 MESSAGE INVALID TAG TYPE (102)

CT CAUSE The tag in a NEW or DISPOSE procedure call is not a tag value of the specified record.

ACTION Correct or remove the non-tag value.

---

103 MESSAGE POINTER OR FILE REQUIRED FOR DEREFERENCE (103)

CT CAUSE A pointer or file is required in a dereference.

ACTION Remove up-arrow or change preceding expression to be of type pointer or file.

---

104 MESSAGE POINTER VARIABLE IS REQUIRED HERE (104)

CT CAUSE NEW, DISPOSE, MARK, and RELEASE all require a pointer variable as the first parameter.

	ACTION	Declare and supply a pointer variable.
106	MESSAGE	MISSING TAG VALUES FOR TAG TYPE (106)
CT	CAUSE	Not all tag values for a tag type in the record are specified.
	ACTION	Add empty variant declarations for the missing tag values.
120	MESSAGE	INVALID BASE TYPE FOR SET (120)
CT	CAUSE	The base type of a set is not an ordinal type.
	ACTION	Check usage in the source program.
121	MESSAGE	ITEM NOT A LEGAL SET ELEMENT (121)
CT	CAUSE	Element of a set is not an ordinal type.
	ACTION	Replace item with a valid element for this set.
122	MESSAGE	OPERAND NOT A SET (122)
CT	CAUSE	Right operand for an IN operator is not a set.
	ACTION	Change expression to set type.
123	MESSAGE	SET ELEMENTS NOT TYPE COMPATIBLE WITH EACH OTHER (123)
CT	CAUSE	In an untyped set constructor, this element is not compatible with the first element in the set.
	ACTION	Change types so they are compatible.
124	MESSAGE	SET ELEMENT NOT COMPATIBLE WITH SET TYPE (124)
CT	CAUSE	In a typed set constructor, the set element is not assignment compatible with the base type of the set.
	ACTION	Replace element with a valid element for this set.
125	MESSAGE	SET OF THIS SIZE CANNOT BE CONSTRUCTED (125)
CT	CAUSE	To construct this set would require more bytes than can be specified for this implementation.
	ACTION	Define/declare set to have fewer elements.
140	MESSAGE	BUILDING OF STRUCTURED CONSTANTS NOT ALLOWED HERE (140)
CT	CAUSE	A constructor that is not a set constructor occurs outside of a CONST declaration section.
	ACTION	Create a named constant in the CONST section and use its name

here.

CAUSE The constructor occurs as an element of a set or string constructor.

ACTION Remove the constructor from the set or string.

---

141 MESSAGE RECORD CONSTANT HAS MISSING FIELD(S) (141)

CT CAUSE One or more fields missing in a record constructor.

The name of a field is misspelled.

ACTION Correct erroneous field name. Add the missing fields.

---

142 MESSAGE DUPLICATE FIELD NAME (142)

CT CAUSE This field has already been defined in the constructor.

ACTION Delete the duplicate declaration.

---

143 MESSAGE FIELD NAME DESIGNATOR NOT ALLOWED HERE (143)

CT CAUSE The constructor is not a record constructor.

This construction <field name>:<expression> appears outside of a record constructor.

ACTION Remove the field name designator from the code.

---

144 MESSAGE MISSING FIELD NAME DESIGNATOR (144)

CT CAUSE The construction <field name>:<expression> is required in a record constructor.

ACTION Add a field name designator to the code.

---

145 MESSAGE TYPE IDENTIFIER REQUIRED HERE (145)

CT CAUSE The identifier preceding the left square bracket of a constructor is not a type identifier.

ACTION Check the syntax of structured constants.

CAUSE The identifier in the bounds construct of a conformant array parameter is not a type identifier.

ACTION Change either the declaration or the usage of the identifier to make sure they are consistent.

---

146 MESSAGE CONSTRUCT ONLY ALLOWED FOR ARRAYS AND STRINGS (146)

CT CAUSE <Count> OF <expression> occurs when the constructor is not an array or string constructor.

ACTION List each element individually and specify its value.

---

147 MESSAGE CONSTRUCT ONLY ALLOWED IN CONSTRUCTORS (147)

CT CAUSE <Count> OF <expression> is used outside of a constructor.

ACTION Remove the <count> OF <expression> from the code.

---

148 MESSAGE SUBRANGE CONSTRUCT ILLEGAL EXCEPT IN SET CONSTRUCTORS (148)

CT CAUSE A subrange construct was used in a string declaration or a non set structured constant.

ACTION Remove the subrange construct from the code.

---

149 MESSAGE TOO BIG STRUCTURED CONSTANT (149)

CT CAUSE The compiler's structured constant table has overflowed.

ACTION If there are one or more structured constants larger than the table size, break them up into smaller constants, if possible.

If the total size of all the structured constants exceeds the limit, break your compilation unit into smaller pieces and spread the constants over them.

Note: The first action may cause the second condition to arise!

---

150 MESSAGE EXPANDED STRING LITERAL IS TOO BIG (150)

CT CAUSE The compiler's identifier table has overflowed.

ACTION Break your compilation unit into smaller pieces and spread your string literals over them.

Read in the string literals from a message catalog.

If the same quoted string is used over and over in the code, declare it as a constant in one place and use the named constant instead.

---

151 MESSAGE TYPE OF CONFORMANT ARRAY BOUNDS MUST BE SCALAR (151)

CT CAUSE The type identifier in the index specification for a conformant array parameter does not designate a scalar type.

ACTION Change index to scalar or subrange.

---

152 MESSAGE PARAMETER DOUBLY DEFINED (152)

CT CAUSE In the index specification of a conformant array parameter the upper bound identifier has the same spelling as the lower bound identifier.

In an index specification of a conformant array parameter a bounds identifier has the same spelling as another parameter or as another bounds identifier in the parameter list of the

		current procedure header.
	ACTION	Rename one of the duplicate identifiers.
153	MESSAGE	NOT ALLOWED AS AN ANYVAR PARAMETER (153)
CT	CAUSE	A parameter can not be an AnyVar parameter and a conformant array parameter.
	ACTION	Change the formal parameter specifier to VAR or omit it.
154	MESSAGE	NON CONFORMANT BASE TYPE (154)
CT	CAUSE	The base type of an array being passed as an actual conformant array parameter must be identical to the base type of the formal conformant array parameter.
	ACTION	Change either the actual array's index type or the formal conformant array's index type so the two are compatible.
155	MESSAGE	NON CONFORMANT ACTUAL PARAMETER (155)
CT	CAUSE	The parameter being passed as an actual conformant array parameter does not have an array type.
	ACTION	Check the parameter and make sure it has an array type.
	CAUSE	The parameter being passed as an actual conformant array parameter does not have the same packing as the formal parameter.
	ACTION	Check that the packing of both parameters are the same and correct if necessary.
156	MESSAGE	NON CONFORMANT ARRAY INDEX (156)
CT	CAUSE	The index type of the actual conformant array parameter is out of range of the type of the index type of the formal parameter.
	ACTION	Change either the actual array's bounds or the formal conformant array's bounds so the actual bounds lie within the formal bounds.
157	MESSAGE	NON IDENTICAL TYPE FOR PARAMETER IN CONFORMANT PARAMETER LIST (157)
CT	CAUSE	In a parameter declaration of the form: p1, p2...pn: <conformant array declaration>, the actual parameters passed must have identical types.
	ACTION	Check the type declarations of the actual parameters, and ensure that they have the *same* type. Break up the formal parameter specifications i.e., make separate and complete declarations of each of p1, p2...pn.

158	MESSAGE	CRUNCHED CONFORMANT ARRAYS ARE NOT ALLOWED (158)
CT	CAUSE	Conformant array parameters cannot be CRUNCHED.
	ACTION	Remove CRUNCHED, or change to PACKED.

159	MESSAGE	NO PACKED CONFORMANT ARRAYS OF CONFORMANT ARRAYS (159)
CT	CAUSE	Packed conformant arrays cannot have, as their elements, conformant arrays.
	ACTION	Add PACKED to the inner type. Remove PACKED from the outer type.

160	MESSAGE	INVALID BASE TYPE FOR FILE (160)
CT	CAUSE	The component type of a file may not be a file or a structure with a file type component.
	ACTION	Remove/change the file being referenced or the declaration of the file.

161	MESSAGE	TEXTFILE VARIABLE IS REQUIRED HERE (161)
CT	CAUSE	The predefined procedure or function in question may only be used with a file of type text.
	ACTION	Remove/change the file being referenced or the routine being used.

162	MESSAGE	TEXTFILE NOT ALLOWED HERE (162)
CT	CAUSE	The standard procedure or function in question may not be used with a file of type text.
	ACTION	Remove/change the file being referenced or the routine being used.

163	MESSAGE	INVALID TYPE FOR A PROGRAM PARAMETER (163)
CT	CAUSE	An identifier in the program parameter list has not been declared as a file variable, or a variable of type PAC, string, or integer.
	ACTION	Correct the actual declaration to be a file declaration or remove the identifier from the program statement.

164	MESSAGE	VARIABLE IS REQUIRED HERE (164)
CT	CAUSE	A variable is required as the target for reading from a file or a string.
	ACTION	Supply a variable in the code.

165	MESSAGE	DEFAULT FILE INPUT MUST BE IN PROGRAM PARAMETER LIST (165)
CT	CAUSE	The file variable in a standard procedure or function call was defaulted to INPUT, but INPUT was not declared in the program parameter list.
	ACTION	Either add 'INPUT' to the program heading or remove the redefinition of 'INPUT', if one was made.

166	MESSAGE	DEFAULT FILE OUTPUT MUST BE IN PROGRAM PARAMETER LIST (166)
CT	CAUSE	The file variable in a standard procedure or function call was defaulted to OUTPUT, but OUTPUT did not appear in the program parameter list.
	ACTION	Add 'OUTPUT' to the program heading or remove the redefinition of 'OUTPUT', if one was made.

167	MESSAGE	FORMAT EXPRESSION ALLOWED ONLY FOR TEXTFILES (167)
CT	CAUSE	A formatted output expression may only occur when writing to a textfile or a string.
	ACTION	Remove the formatted expression from the code.

168	MESSAGE	INTEGER VALUE IS REQUIRED HERE (168)
CT	CAUSE	The expressions specifying the field width and the number of decimal digits for an output expression are not type integer or a subrange of integer.
	ACTION	Replace with an integer expression.

169	MESSAGE	SECOND FORMAT VALUE ALLOWED ONLY FOR REAL OR LONGREAL (169)
CT	CAUSE	The format value that specifies the number of decimal digits in an output expression is only legal for output values of type real or longreal.
	ACTION	Check type of parameter or remove decimal position specifier.

190	MESSAGE	THIS PROGRAM PARAMETER WAS UNDECLARED: "! " (190)
CT	CAUSE	The identifier appeared in the program parameter list but was never declared.
	ACTION	Add the identifier declaration.

191	MESSAGE	DUPLICATE PROGRAM PARAMETER (191)
CT	CAUSE	There is more than one PARM parameter or more than one INFO parameter in a program parameter list.
	ACTION	Remove duplicate declarations.

---

192 MESSAGE PARAMETER "! " DOES NOT MATCH POSSIBLE SPL TYPES (192)

CT CAUSE The HP Pascal type of the parameter does not correspond to an acceptable SPL type.

ACTION Change the parameter definition to a type that will correspond to the SPL type.

---

193 MESSAGE PARAMETER "! " DOES NOT MATCH INTRINSIC PARM TYPE (193)

CT CAUSE The HP Pascal type of the parameter does not match the parameter type required by the INTRINSIC.

ACTION Change the parameter definition to a type that will correspond to the intrinsic type.

---

194 MESSAGE MISSING FUNCTION RETURN SPECIFICATION (194)

CT CAUSE The return type is not specified in the function heading.

ACTION Insert the result type declaration.

---

195 MESSAGE INVALID PARAMETER TO HALT (195)

CT CAUSE The optional parameter to HALT is not type integer or an integer subrange.

ACTION Change the parameter to type integer or supply no parameter.

---

196 MESSAGE THIS INTRINSIC MAY NOT BE USED AS A FUNCTION (196)

CT CAUSE The specified intrinsic does not return a result and cannot be declared as a function.

ACTION Redeclare the intrinsic as a procedure.

---

197 MESSAGE ELEMENTS OF PACKED OR CRUNCHED STRUCTURES CANNOT BE PASSED BY VAR (197)

CT CAUSE Elements of packed arrays or records may not be passed to a routine expecting a reference parameter.

ACTION Redeclare the intrinsic as a procedure.

---

198 MESSAGE EMPTY PARAMETER MAY NOT BE USED HERE (198)

CT CAUSE Actual parameters may only be omitted for EXTERNAL SPL VARIABLE procedures or for intrinsics that are extensible and/or have default parameters.

ACTION Supply a value for the parameter in question.

---

199 MESSAGE PROCEDURE NOT DECLARED (199)

CT	CAUSE	The identifier used in the procedure call either has not been declared, or it is not a procedure name.
	ACTION	Check the spelling of the procedure and make sure it is declared.

200	MESSAGE	PARAMETER "! " MUST BE VAR PARAMETER. (200)
CT	CAUSE	The parameter in the intrinsic declaration was specified as a value parameter, but the intrinsic requires a reference parameter.
	ACTION	Change the intrinsic declaration so that it specifies the parameter in question as a VAR parameter.

Messages 201-527

201	MESSAGE	PARAMETER "! " MUST BE VALUE PARAMETER (201)
CT	CAUSE	The parameter in the intrinsic declaration was specified as a reference parameter, but the intrinsic requires a value parameter.
	ACTION	Change the intrinsic declaration to specify the parameter in question as a value parameter.

202	MESSAGE	INVALID USE OF PROCEDURE OR FUNCTION IDENTIFIER (202)
CT	CAUSE	A procedure identifier appears as a function call. A function identifier appears as a procedure call. A valid identifier mistakenly appears as a function or procedure identifier.
	ACTION	Change either the declaration or the usage of the identifier to make sure they are consistent.

203	MESSAGE	INCONSISTENT DEFINITION OF FORWARD PROCEDURE OR FUNCTION (203)
CT	CAUSE	The definition of a procedure declared FORWARD is a function. The definition of a function declared FORWARD is a procedure.
	ACTION	Change either the declaration or the usage of the identifier to make sure they are consistent.
	CAUSE	The ALIAS in the definition differs from the ALIAS in the FORWARD declaration of a procedure or function.
	ACTION	Make ALIAS names identical or only use ALIAS in the FORWARD declaration.
	CAUSE	A FORWARD declaration is already provided for a function or procedure now declared FORWARD, EXTERNAL, or INTRINSIC.
	ACTION	Remove all but one of the declarations.
	CAUSE	The definition is missing a routine option or compiler option which was specified in the FORWARD declaration.

	ACTION	Make sure all routine options or compiler options are repeated in the definition of the procedure or function.

204	MESSAGE	INVALID DIRECTIVE (204)
CT	CAUSE	EXTERNAL, EXTERNAL SPL, EXTERNAL SPL VARIABLE, EXTERNAL FORTRAN, EXTERNAL FTN77, EXTERNAL C, EXTERNAL COBOL, FORWARD, and INTRINSIC are the only legal directives.
	ACTION	Remove the directive from the code or correct the spelling.

205	MESSAGE	INVALID LANGUAGE SPECIFICATION (205)
CT	CAUSE	The language specified was not FORTRAN, SPL, COBOL, FTN77, or C. A language cannot be specified with the FORWARD or INTRINSIC directives.
	ACTION	Remove or correct the language specification.

206	MESSAGE	INCORRECT NUMBER OF PARAMETERS (206)
CT	CAUSE	The number of actual parameters given is either too few or too many for the procedure or function.
	ACTION	Check consistency between the procedure call and procedure declaration.

207	MESSAGE	UNMATCHED PARAMETERS IN FORWARD (207)
CT	CAUSE	Parameters in the definition of a procedure or function declared FORWARD do not match the parameters of the original heading.
	ACTION	Check whether the FORWARD routine declaration and the routine declaration are consistent.

208	MESSAGE	ACTUAL PARAMETER NOT COMPATIBLE WITH FORMAL PARAMETER (208)
CT	CAUSE	This actual reference parameter is not type identical with the formal reference parameter in a user-defined function or procedure. This actual value parameter is not assignment compatible with the formal value parameter in a user-defined function or procedure.
	ACTION	Check the types of the actual and formal parameters.
	CAUSE	This actual reference parameter to a standard function or procedure is not type identical with the formal reference parameter. This actual value parameter to a standard function or procedure is not assignment compatible with the required type.
	ACTION	Check the types of the actual parameter and the parameter

		accepted by the predefined routine.
	CAUSE	This actual parameter is not intrinsic compatible with the intrinsic parameter.
	ACTION	Check the types of the actual parameter and the intrinsic parameter.
	CAUSE	The parameter of the standard SQR function is an integer subrange type with a lower bound greater than the square root of maxint, or an upper bound less than the negation of the square root of maxint. In either case, an integer overflow is possible at run time.
	ACTION	Do not call SQR.
209	MESSAGE	NO FURTHER CASE CONSTANT PARAMETERS ALLOWED TO NEW (209)
CT	CAUSE	The pointer parameter to NEW points to a record that has no additional nested variant parts. The pointer parameter to NEW points to a record that does not have a variant part. The pointer parameter to NEW points to a structure that is not a record.
	ACTION	Check the record type definition for the correct variant record or remove the extra variant labels from the call.
210	MESSAGE	NO FURTHER CASE CONSTANT PARAMETERS ALLOWED TO DISPOSE (210)
CT	CAUSE	The pointer parameter to DISPOSE points to a record that has no additional nested variant parts. The pointer parameter to DISPOSE points to a record that does not have a variant part. The pointer parameter to DISPOSE points to a structure that is not a record.
	ACTION	Check the record type definition for the correct variant or remove the extra variant labels from the call.
211	MESSAGE	NO FURTHER PARAMETERS ALLOWED TO MARK (211)
CT	CAUSE	More than one pointer parameter in a call to MARK.
	ACTION	Remove the extra parameter.
212	MESSAGE	NO FURTHER PARAMETERS ALLOWED TO RELEASE (212)
CT	CAUSE	More than one pointer parameter in a call to RELEASE.
	ACTION	Remove the extra parameter.
213	MESSAGE	VALUE PARAMETER MAY NOT CONTAIN FILE COMPONENT (213)

CT	CAUSE	This value formal parameter is a file or a structured type with a file type component. This is equivalent to assigning to a file.
	ACTION	Remove the file component from the source code.

214	MESSAGE	FUNCTION TYPE MAY NOT CONTAIN FILE COMPONENT (214)
CT	CAUSE	This function return type is a file or a structured type that contains a file type component. This is equivalent to assigning to a file.
	ACTION	Remove the file component from the source code.

215	MESSAGE	COMPILER LEVEL WRONG--PROBABLY UNMATCHED "END" (215)
CT	CAUSE	This occurrence of END cannot match a BEGIN because all compound statements have been terminated. The compiler disregards the extraneous END.
	ACTION	Ensure all BEGINs and ENDs match along with ENDs for CASEs. Make sure a BEGIN has not been commented out or fix any syntax errors.

216	MESSAGE	BAD CONSTANT PARAMETER (216)
CT	CAUSE	This string constant parameter to BINARY, OCTAL, or HEX either contains an invalid character or represents a value outside the range minint..maxint.
	ACTION	Fix the character construct.
	CAUSE	This parameter to SUCC is a constant value equal to the maximum value of an ordinal type. This parameter to PRED is a constant value equal to the minimum value of an ordinal type.
	ACTION	Fix the constant value.

217	MESSAGE	PROCEDURE OR FUNCTION NOT IN INTRINSIC FILE (217)
CT	CAUSE	An incorrect intrinsic file was specified prior to the declaration of the procedure or function.
	ACTION	Check the name of the SYSINTR file.
	CAUSE	The INTRINSIC name differs slightly from the procedure or function name declared INTRINSIC.
	ACTION	Either use the ALIAS option or correct the spelling of the ALIAS parameter.
	CAUSE	The procedure has never been put into the intrinsic file.
	ACTION	Either check the spelling or list the intrinsic file (or rebuild the intrinsic file if it is not the standard intrinsic file.)

---

218 MESSAGE INTRINSIC FILE NOT CHECKED (218)

CT CAUSE Due to a prior error, the intrinsic file was never opened. Thus, no attempt was made to look up this procedure or function.

ACTION Fix the previous error and try again.

---

219 MESSAGE "STRING" IS NOT ALLOWED AS A VALUE PARAMETER (219)

CT CAUSE A string formal value parameter must have a specified maximum length.

ACTION Make the declaration a VAR parameter or make the type a specific string type.

---

220 MESSAGE FUNCTION "! " NOT ASSIGNED TO (220)

CT CAUSE A function of a simple type has no assignment to the result in the function body.

A function of a structured type has no assignment to any component of the result in the function body.

ACTION Make an assignment to the function result.

---

221 MESSAGE DECLARED FUNCTION TYPE DOES NOT MATCH INTRINSIC TYPE (221)

CT CAUSE The HP Pascal type of the return of a function declared INTRINSIC does not match the type of the value returned by the intrinsic.

ACTION Change the type so it matches the value of the intrinsic type.

---

222 MESSAGE VARIABLE PARAMETER REQUIRED HERE (222)

CT CAUSE An expression appears as an actual reference parameter instead of a variable.

A constant appears as an actual reference parameter instead of a variable.

A component of a structured constant appears as an actual reference parameter instead of a variable.

ACTION Check the parameter; it must be a variable and not an expression or constant.

---

223 MESSAGE ILLEGAL PARAMETER FORM (223)

CT CAUSE The integer parameter to a string procedure/function is not compatible with a 32 bit integer.

The actual parameter is a procedure or function identifier, but the corresponding formal parameter is not a procedure or function heading.

The parameters of the actual procedural or functional parameter are not congruent with the parameters of the formal procedural

		or functional parameter. The parameter of a call to WADDRESS or SIZEOF is a component of a packed structure. The parameter of a call to BADDRESS is a component of a packed structure other than a PAC. Either the third parameter of a call to ASSERT is not a procedure identifier or the parameter of such a procedure is not an integer value parameter.
	ACTION	Check the types of the actual and formal parameters.
224	MESSAGE	SYSTEM ADDRESSING LIMIT EXCEEDED (224)
CT	CAUSE	The storage limit for variables at run time is exceeded.
	ACTION	Reduce the number of variables or make the structured variables, such as arrays or strings, smaller.
225	MESSAGE	INCONSISTENT ALIAS IN FORWARD PROCEDURE OR FUNCTION (225)
CT	CAUSE	The ALIAS in the definition differs from the ALIAS in the FORWARD declaration of a procedure or function.
	ACTION	Use the same ALIAS in both the declarations.
226	MESSAGE	INCONSISTENT OPTIONS IN FORWARD PROCEDURE OR FUNCTION (226)
CT	CAUSE	The routine options specified in the definition differs from the one in the FORWARD declaration of the procedure or function.
	ACTION	Use the same routine options in both declarations.
227	MESSAGE	INCONSISTENT COMPILER OPTIONS IN FORWARD PROCEDURE OR FUNCTION (227)
CT	CAUSE	The compiler options specified in the definition differ from the one in the FORWARD declaration of the procedure or function.
	ACTION	Use the same compiler options in both declarations.
228	MESSAGE	VARIABLE OR EXPRESSION NOT WITHIN STRING LIMITS (228)
CT	CAUSE	The bounds of a subrange variable used as a string index do not overlap the bounds of the string type.
	ACTION	Use a variable of the proper type.
	CAUSE	The constant expression used as a string index lies outside the bounds of the string type.
	ACTION	Use a constant expression within the string bounds.

229	MESSAGE	INCONGRUENT FORMAL PARAMETER SECTIONS (229)
	CAUSE	The formal parameter sections of the actual routine being passed as a parameter are not congruent with the formal parameter sections of the procedural or functional parameter of the called routine.
	ACTION	Alter one of the formal parameter sections so that it is congruent with the other. Raise the STANDARD_LEVEL to HP_PASCAL.
230	MESSAGE	INVALID CONTROL VARIABLE IN FOR STATEMENT (230)
CT	CAUSE	The control variable of the FOR loop is a record field. The control variable of the FOR loop is defined in a scope containing the current scope. The control variable of the FOR loop is a formal parameter of a procedure or function containing the FOR statement. The identifier used as the control variable of the FOR is not a variable.
	ACTION	Use a local ordinal variable for the loop control variable.
231	MESSAGE	CONTROL VARIABLE NOT AN ORDINAL TYPE (231)
CT	CAUSE	The control variable of the FOR loop is not an ordinal type.
	ACTION	Use a local ordinal variable for the loop control variable.
232	MESSAGE	EXPRESSION NOT COMPATIBLE WITH CONTROL VARIABLE (232)
CT	CAUSE	The expressions for the initial and final values are not type compatible with the control variable of a FOR loop.
	ACTION	Check expressions and make sure the types are compatible.
233	MESSAGE	INITIAL AND FINAL EXPRESSIONS NOT COMPATIBLE (233)
CT	CAUSE	The types of the expressions for the initial and final values of the FOR loop are not type compatible.
	ACTION	Change the types of the initial and final value expressions or of the loop control variable as appropriate.
240	MESSAGE	MULTIPLE MODULE IMPLEMENTATIONS NOT PERMITTED (240)
CT	CAUSE	Only one MODULE is permitted for each module.
	ACTION	Remove duplicate MODULE.
241	MESSAGE	MISSING EXPORT SECTION FOR THIS MODULE (241)

CT CAUSE Every module must have at least one EXPORT.

ACTION Declare or define at least one 'object' in the EXPORT section.

---

242 MESSAGE INVALID IMPORT MODULE IDENTIFIER (242)

CT CAUSE The given identifier is not defined.

The given identifier is not the name of a module in the current $SEARCH$ list.

ACTION Check the name of the IMPORT module identifier. If the PASLIB file in which the module is defined is not in the current search list, add the file to it.

---

243 MESSAGE NOT AN IMPORTED MODULE (243)

CT CAUSE The identifier is not the name of an import module or the module currently being defined

ACTION If the name is misspelled, correct the spelling. Otherwise, import the module in question.

---

250 MESSAGE DUPLICATE CASE LABEL (250)

CT CAUSE The CASE label is the same as a CASE label that appeared previously in the same construct.

The CASE label is contained in a previous CASE label subrange in the same construct.

The CASE label subrange contains at least one CASE label that appeared previously in the same construct.

ACTION Remove the duplicate label from the code.

---

251 MESSAGE CASE LABEL OF INCORRECT TYPE (251)

CT CAUSE The type of the CASE label is not the same as the type of the tag or the select expression.

ACTION Change the label or selecting expression as appropriate.

---

252 MESSAGE CASE LABEL TYPE NOT SAME AS PREVIOUS CASE LABEL (252)

CT CAUSE There was a detected error in the tag type or select expression, so the CASE labels are checked against each other. The type of the current CASE label does not match the type of previous CASE labels.

ACTION Make sure that all case labels in a CASE statement are of the same type.

---

270 MESSAGE INVALID LABEL - MUST BE AN INTEGER BETWEEN 0 AND 9999 (270)

CT CAUSE This label is not an integer.

A colon (:) appears or was inserted by the compiler where no label was desired.

ACTION Check to ensure that the label is an integer between 0 and 9999.

---

271 MESSAGE LABEL HAS NOT BEEN DECLARED (271)

CT CAUSE This label marks a statement, but never appeared in a LABEL declaration for this block.

ACTION Declare the label.

---

272 MESSAGE LABEL DECLARED MORE THAN ONCE (272)

CT CAUSE This label already appeared in this LABEL section or in a LABEL section in an enclosing scope.

ACTION Delete the duplicate label declaration.

---

273 MESSAGE SAME LABEL NOT ALLOWED ON MORE THAN ONE STATEMENT (273)

CT CAUSE This label has already marked a statement.

ACTION Remove/correct the duplicate definition.

---

274 MESSAGE LABEL '!' NOT USED (274)

CAUSE The label is referenced in a GOTO statement, but is not used to mark a statement.

ACTION Mark a target statement with the label.

---

275 MESSAGE LABEL REFERENCED BY GOTO OUTSIDE STRUCTURED STATEMENT (275)

CT CAUSE This label appears in a component statement of a structured statement and was previously referenced by a GOTO statement:

(a) preceding the structured statement.

(b) in a preceding component statement of the same structured statement.

(c) contained in an inner procedure or function.

ACTION Remove either the label or the GOTO from the code.

---

276 MESSAGE GOTO REFERENCES LABEL INSIDE STRUCTURED STATEMENT (276)

CT CAUSE The label referenced in a GOTO statement appears in a component statement of a structured statement and the GOTO statement appears:

(a) after the structured statement.

(b) in a later component statement of the same structured

statement.

ACTION Remove either the label or the GOTO from the code.

---

293 MESSAGE TSAM INTRINSIC ERROR "! " (293)

CT CAUSE An error was encountered when reading a TSAM (toolset format) file.

ACTION The error number replacing "! " refers to Toolset error messages if 900 or above. Look them up in a Toolset manual. Please report other numbers to your local HP representative.

---

294 MESSAGE $ INCLUDE NOT ALLOWED HERE WHEN SYMBOLIC DEBUG IS ENABLED (294)

CT CAUSE $INCLUDE of a file in executable code must be on a Pascal statement boundary if symbolic debug is enabled.

---

370 MESSAGE IMPORTED MODULE ‘!‘ WAS NOT REFERENCED (370)

N CAUSE The specified module was imported and no references to it were found.

ACTION Either remove the module from the IMPORT statement or cause the module to be referenced.

---

371 MESSAGE USE OF AN INLINED ROUTINE (371)

N CAUSE An inlined routine has been expanded in the current statement.

ACTION No action required. This message is for your information only.

---

373 MESSAGE ASSUME "! " IS VALID, USE $ASSUME$ (373)

N CAUSE The given optimizer assumption is valid, and should be used in the routine's declaration to get the most out of optimization.

ACTION Use the $ASSUME$ compiler option.

---

374 MESSAGE BIT32 Type CONVERTED TO LONG INTEGER (374)

N CAUSE Using bit32 requires that it be converted to a long integer.

ACTION Use type coercion to obtain a signed or unsigned 32 bit operation.

---

377 MESSAGE CODE GENERATED TO VERIFY CORRECT POINTER ALIGNMENT (377)

N CAUSE Checking code will be generated to ensure that the pointer being coerced has an alignment that allows it to be used as the coerced pointer type.

ACTION Use $RANGE OFF$ to eliminate the extra code.

378	MESSAGE	WHICH IS A COMPONENT OF ' ! '(378)
N	CAUSE	This message accompanies message #379.
	ACTION	See message 379.

379	MESSAGE	THE FIELD / AN ELEMENT OF ' ! ', CROSSES A WORD BOUNDARY (379)
N	CAUSE	Accesses of ordinal data items split across word boundaries are relatively inefficient.
	ACTION	Use $ALIGNMENT$ to start ordinal data items on word boundaries.

380	MESSAGE	TYPE COERCION MAY ACCESS INVALID DATA (380)
N	CAUSE	Informational message - the referenced type-coercion may cause uninitialized/invalid data to become accessible.
	ACTION	Ensure that the data referenced is valid.

381	MESSAGE	MACHINE DEPENDENT REPRESENTATION USED IS NOT CONSISTENT WITH PACKING(381)
N	CAUSE	A real type, such as a real or longreal, is used with $HP3000_16$.
	ACTION	Don't mix $HP3000_16$ and $HP3000_32$ modes in data declarations.

382	MESSAGE	SIZE OF MACHINE DEPENDENT TYPE IS NOT CONSISTENT WITH PACKING (382)
N	CAUSE	A machine dependent type such as a pointer, string, or file, is used with $HP3000_16$.
	ACTION	Don't mix $HP3000_16$ and $HP3000_32$ modes in data declarations.

383	MESSAGE	FEATURE MAY NOT BE SUPPORTED FOR OTHER TARGET MACHINES (383)
N	CAUSE	Informational message - the referenced feature may not be supported on other machines.
	ACTION	No action is required.

384	MESSAGE	MOVE PROCEDURE IN STATEMENT "! " USES A SIMULATED FOR LOOP (384)
N	CAUSE	The MOVE predefined procedure is implemented with a FOR loop to move the elements.
	ACTION	No action is required.

385	MESSAGE	POSSIBLE NON-ALIGNED OVERLAPPING SOURCE/TARGET IN STATEMENT ! (385)
N	CAUSE	The source and target of the MOVE predefined procedure may overlap and generate scrambled results.
	ACTION	You may need to use MOVE_R_TO_L or MOVE_L_TO_R.

400	MESSAGE	INVALID FILENAME (400)
CT	CAUSE	The filename given in the INCLUDE, SYSINTR, or SPLINTR option is not a legal filename.
	ACTION	Correct the filename to conform to the format required by the operating system.

401	MESSAGE	ILLEGAL NAME IN ALIAS OR SUBPROGRAM OPTION (401)
CT	CAUSE	The procedure or function name in an ALIAS option is not a valid identifier. The procedure or function name in a SUBPROGRAM option is not a valid HP Pascal identifier.
	ACTION	Make sure the name is a valid HP Pascal identifier.

402	MESSAGE	NOT A LEGAL LOCALITY NAME (402)
CT	CAUSE	The name for a locality is illegal.
	ACTION	Check the name and make sure it is legal.

403	MESSAGE	IF EXPRESSION CAN NOT BE EVALUATED (403)
CT	CAUSE	The expression in an IF has a syntax error in it.
	ACTION	Check the source code and fix the syntax error.-

404	MESSAGE	UNMATCHED $ENDIF$ FOUND (404)
CT	CAUSE	An $ENDIF$ compiler option was found without a preceding IF option. This may happen if either the compiler rejects an IF because it was out of place, or the IF is not in the code.
	ACTION	Check for a missing or commented IF.

405	MESSAGE	A BOOLEAN EXPRESSION IS REQUIRED INSIDE STRING (405)
CT	CAUSE	A blank string was found as part of an IF.
	ACTION	Remove the IF or add a string.

406	MESSAGE	EXPECTED TRUE/FALSE AFTER "= " (406)

CT	CAUSE	Misspelled true/false after "= " in SET
	ACTION	Correct spelling.
	CAUSE	Missing true/false after "= " in SET
	ACTION	Add TRUE or FALSE.

408	MESSAGE	UNMATCHED $ENDIF$ OR $ELSE$ FOUND (408)
CT	CAUSE	$ENDIF$/$ELSE$ compiler option was found without a preceding IF option. This may happen either if the compiler rejects an IF because it was out of place, or if the IF is not in the code.
	ACTION	Check for a missing or misplaced IF.

409	MESSAGE	EXCEEDED MAXIMUM NESTING LEVEL FOR IF (409)
CT	CAUSE	The nesting of IF exceeded the maximum allowable nesting level.
	ACTION	Remove the offending IF from the code.

410	MESSAGE	ILLEGAL IDENTIFIER IN SET or IF (410)
CT	CAUSE	An identifier is misspelled.
		Expected an identifier and one was not found.
	ACTION	Provide a legal identifier or correct the spelling of the identifier.

411	MESSAGE	$PUSH$ NESTING TOO DEEP, OPTIONS NOT SAVED (411)
CT	CAUSE	Too many $PUSH$ compiler options encountered.
	ACTION	Remove the offending $PUSH$ option.

412	MESSAGE	NOTHING TO POP, OPTIONS NOT CHANGED (412)
CT	CAUSE	Too many POP compiler options for the number of preceding $PUSH$ options.
	ACTION	Remove POP options so that those remaining have matching $PUSH$ options.

413	MESSAGE	INVALID INTRINSIC FILE (413)
CT	CAUSE	The file specified in the intrinsic option is not a valid SYSINTR file.
	ACTION	Check the name and make sure the file is an intrinsic file and has not been corrupted.

---

414 MESSAGE NLS NOT INSTALLED OR SYSTEM VARIABLE NOT SET (414)

CT CAUSE NLS (Native Language Support) is not installed or the JCW 'GETUSERLANG" is not set (MPE/iX) or the environment variable 'LANG' is not set (HP-UX).

ACTION Determine which of the above applies and correct the situation.

---

415 MESSAGE $INCLUDE FILENAME IS NULL (415)

CT CAUSE The file specified in the include option is empty.

ACTION Place a valid file name in the quotes.

---

425 MESSAGE COMPILER ERROR "! " COMPILE TERMINATED (425)

CT CAUSE (1..999) A run-time error was detected by the run-time support library during compiler execution.

(1000..1031) A run-time error was detected in an arithmetic operation during compiler execution.

(2000..2999) A run-time error was detected by a system intrinsic during compiler execution.

ACTION Check that there is no previous syntax error. If there is one, fix the error and recompile. Otherwise, report this as a bug.

(3000..3999) A run-time code trap (an addressing exception or an illegal instruction, for example) occurred during compiler execution.

(5000..5999) A user of internal code generation error.

(6000..6999) An optimizer error.

(7000..7999) A user or internal code generation error.

---

426 MESSAGE SYSTEM RESOURCE EXHAUSTED "! " COMPILE TERMINATED (426)

CT CAUSE The compiler ran out of space in the heap.

ACTION Break the code into smaller compilation units.

CAUSE The compiler ran out of space in one of its data areas or the compiler could not acquire one of its data areas (especially if the parameter is 2).

ACTION Reduce the size or the number of structured constants or number of identifiers or increase the size of data areas. In the following examples, the parameter is the number of pages:

On MPE/iX:

```
SETJCW PASXDATA 200
```

On HP-UX:

```
export PASXDATA=200     for ksh
```

PASXDATA=200; export PASXDATA *for sh or ksh*

setenv PASXDATA 200 *for csh*

---

461 MESSAGE PARSER STACK OVERFLOW - TOO MANY NESTED CONSTRUCTS (461)

CT CAUSE An internal compiler limit on nested structures has been reached. A common cause is a long list of ELSE-IFs.

ACTION Break up a nested structure. Use a balanced IF-THEN-ELSE structure.

---

500 MESSAGE OPTION NOT YET IMPLEMENTED (500)

W CAUSE This compiler option is not yet implemented.

ACTION Remove any references to this compiler option from the source code.

---

501 MESSAGE UNRECOGNIZED COMPILER OPTION (501)

W CAUSE A compiler option with this name is not recognized.

ACTION Check the spelling of this option.

---

502 MESSAGE THIS OPTION IS NOT ALLOWED HERE (502)

W CAUSE The option appears in an illegal location in the source code. For example, the GLOBAL option appears anywhere except before the PROGRAM heading.

ACTION Remove the option from an illegal location in the source code and place it in a legal location.

---

503 MESSAGE TEXT AFTER INCLUDE OR SKIP TEXT IGNORED (503)

W CAUSE Anything on the source line after INCLUDE was ignored.

Anything on the source line after a $SKIP_TEXT ON$ is treated as a comment. Anything on the source line after an IF that evaluates to FALSE is ignored.

ACTION Remove the extra text.

---

504 MESSAGE INTEGER OUT OF RANGE, VALUE NOT CHANGED (504)

W CAUSE LINES requires an integer greater than 20

WIDTH requires an integer in the range 10..132.

$CHECK_ACTUAL_PARM$ and $CHECK_FORMAL_PARM$ require an integer in the range 0..3.

ACTION Correct the option argument; check the compiler option syntax.

---

505 MESSAGE STRING PARAMETER IS REQUIRED, OPTION IGNORED (505)

W CAUSE This option requires information in a string literal parameter.

ACTION Check the option argument; check the compiler option syntax.

---

506 MESSAGE I/O FAILED ON FILE !, ! (506)

CAUSE I/O on a file failed. The compiler feature that uses that file has been disabled for the remainder of the compilation.

ACTION Check the named file for invalid file equations, links, size restrictions, and locking by other processes. Also check for disk space.

---

507 MESSAGE BOTH $GLOBAL$ AND $EXTERNAL$ NOT ALLOWED (507)

W CAUSE The option $GLOBAL$ occurred after the option $EXTERNAL$ was specified. Since only one is allowed, $GLOBAL$ was ignored.

The option $EXTERNAL$ occurred after the option $GLOBAL$ was specified. Since only one is allowed, $EXTERNAL$ was ignored.

ACTION Remove $GLOBAL$ or $EXTERNAL$, whichever is appropriate.

---

508 MESSAGE A "$ " IS REQUIRED HERE - ONE INSERTED (508)

W CAUSE Compiler option doesn't end with a $ on the same line.

ACTION Add a "$ " to the code.

---

509 MESSAGE EXPRESSION WILL CAUSE A RUN-TIME OVERFLOW (509)

W CAUSE The result of an expression will exceed maxint at run time. This is detected for:

(a) +, -, * when the types of the operands are such that the expression overflows. For example:

```
VAR
     A: maxint-10..maxint;
```

Then the expression A + A would never be less than 2 * maxint - 10, which is greater than maxint.

(b) -minint

(c) the addition, subtraction, or multiplication of two constants resulting in an overflow.

ACTION Correct the expression.

---

510 MESSAGE EXPRESSION WILL CAUSE A RUN-TIME UNDERFLOW (510)

W CAUSE The result of an expression will be less than minint at run time. This is detected for:

(a) +, -, * when the types of the operands are such that the

expression underflows. For example:

```
VAR
     A: maxint - 10..maxint;
     B: minint..minint + 10
```

Then the expression B - A would be less than minint + 10 - maxint, which is less than minint.

(b) the addition, subtraction, or multiplication of two constants resulting in an underflow.

ACTION Correct the expression.

---

511 MESSAGE MOD DIVISOR WILL CAUSE A RUN-TIME ERROR (511)

W CAUSE In an expression A MOD B, B will be <= 0 at run time.

In a constant expression A MOD B, B is <= 0.

ACTION Correct the expression.

---

512 MESSAGE RUN TIME DIVISION BY ZERO (512)

W CAUSE In an expression A DIV B, B = 0.

In a constant expression A DIV B, B = 0.

ACTION Correct the expression.

---

513 MESSAGE EMPTY INCLUDE FILE (513)

W CAUSE The INCLUDE file had no text in it.

ACTION Verify that the filename is correct.

---

514 MESSAGE $ NOT ALLOWED IN INFO PARAMETER (514)

W CAUSE The INFO parameter of a :PASXL, :PASXLLK, or :PASXLGO command is interpreted as a compiler option with the $ assumed as the leading and trailing character. The $ cannot appear in the INFO string itself.

ACTION Do not supply '$' in the INFO string.

---

515 MESSAGE NO DISC SPACE FOR XREF (515)

W CAUSE A file error occurred trying to open the file needed to do the cross reference. This could be any file error, but OUT OF DISK SPACE is the most likely. A temporary file with the name PASXRFdd, where d is a digit, is another possible cause.

ACTION Check for a duplicate file name and for enough file disk space.

---

516 MESSAGE NO VARIANT FOR TAG VALUE (516)

W	CAUSE	A NEW was called specifying a tag constant that did not appear in the case list in the variant part. The maximum space for the record is allocated.
	ACTION	Remove the variant identifier from the source code or correct its spelling.

519	MESSAGE	BOOLEAN EXPRESSION FOLDED TO '!' (519)
W	CAUSE	The compiler has folded an expression with IN, AND, or OR and constant operands or, in the case of IN, with a left operand that is a constant appearing in the set list. The compiler has folded an expression with =, <>, <=, >=, or > and operands that are non-set constants. With $PARTIAL_EVAL ON$, the compiler has folded an expression with OR when TRUE is an operand, or an expression with AND when FALSE is an operand.
	ACTION	Check the operands to ensure that they are correct.

520	MESSAGE	NON-OVERLAPPING TYPES - EXPRESSION FOLDED (520)
W	CAUSE	Two sets with ranges that do not overlap were intersected. The compiler folded the expression to the empty set. An arithmetic comparison was done with operands of types with ranges that do not overlap. The compiler folded the expression. For example, if A: 0..3 and B: 5..7, then A = B is folded to false.
	ACTION	Check the operands to ensure that they are correct.

521	MESSAGE	BODY OF FOR LOOP WILL NEVER EXECUTE (521)
W	CAUSE	Values of the initial and final expressions will prevent the body of the FOR loop from ever executing. Non-overlapping subranges for the types of the initial and final expressions prevent the body of the FOR loop from ever executing.
	ACTION	Check the values and types for the initial and final expressions.

522	MESSAGE	CASE LABEL NOT WITHIN TAG OR SELECT EXPRESSION RANGE (522)
W	CAUSE	The CASE label value or subrange is not within the range of the tag type and can never be specified in a call to NEW or assigned to the tag field. The CASE label value or subrange is not within the range of the select expression and can never be selected.
	ACTION	Check the possible values of the CASE selection expression and the values of the CASE labels.

---

523 MESSAGE INTEGER CONSTANT IS REQUIRED - OPTION IGNORED (523)

W CAUSE This compiler option requires an integer parameter; such as WIDTH. The compiler has ignored this option.

ACTION Check the syntax and insert an integer where necessary.

---

524 MESSAGE SUBPROGRAM "! " SPECIFIED, BUT NOT FOUND (524)

W CAUSE A procedure or function name specified in the SUBPROGRAM option was not found in the source.

ACTION Check the spelling of the procedure or function.

---

525 MESSAGE ANY EXTERNAL GOTO TO THIS LABEL IS AN ERROR (525)

W CAUSE This label marks a component statement of a structured statement. This label cannot be referenced by a GOTO statement contained in an external procedure or function, but that error will not be detected until the program is prepared or executed.

ACTION Make sure no nonlocal GOTOs branch to this label.

---

526 MESSAGE EXPRESSION FOLDED TO THE EMPTY SET (526)

W CAUSE The compiler has determined that a set expression results in an empty set and folded that expression to empty. This warning appears in case the user expected side effects or made some kind of error that caused the folding. Folding occurs when an intersection is performed with the empty set, the empty set occurs on the left side of the set difference operator, or two empty sets appear in a set operation.

ACTION Check to see if expression should fold to the empty set.

---

527 MESSAGE 'ON' OR 'OFF' IS REQUIRED HERE (527)

W CAUSE The word ON or OFF is required after this compiler option name; for example, $LIST$.

ACTION Correct the option argument and the compiler option syntax.

Messages 528-814

---

528 MESSAGE PREVIOUS VERSION OF '!' INACTIVATED (528)

W CAUSE A procedure or function by the same name already exists in the USL file and has been inactivated.

CAUSE If PRIVATE_PROC was ON, then two level 1 procedure or function names are not unique within the first 15 characters, or a copy from a previous compilation is being replaced.

CAUSE If PRIVATE_PROC was OFF, then either duplicate non-level 1 procedure or function names exist (they are not unique within 15 characters) or duplicate procedure or function names have been introduced due to separate compilation of procedures or

functions wi th names which are identical within the first 15 characters.

---

530 MESSAGE EXPRESSION WILL CAUSE A RUN-TIME SET RANGE ERROR (530)

W CAUSE Evaluation of a set construction in which an element of the set list will necessarily fall outside the bounds of the set construction will cause this error.

ACTION Check the source code and fix the expression.

---

532 MESSAGE THE SPECIFIED WORKSPACE FOR TOOLSET IS INVALID (532)

W CAUSE The file is not a valid TSAM root file or the file cannot be opened.

ACTION Determine why the file is invalid.

---

533 MESSAGE BAD FONT OPTION GIVEN (533)

W CAUSE The call to FDeviceControl returned an error condition.

ACTION Ensure that the font number specified exists in the font file specified in the file equation for PASLIST.

---

534 MESSAGE CONTROL VARIABLE HAS BEEN ASSIGNED TO NON-LOCALLY (534)

W CAUSE The control variable may be modified by a non-local reference from a routine invoked in the body of the FOR loop.

ACTION Make sure that there are no non-local references to the control variable.

---

535 MESSAGE "! " ACCESSED, BUT NOT INITIALIZED (535)

W CAUSE A simple variable appears in an expression, as a value parameter, or in some other accessing reference and it has never appeared in an assigning reference, such as a reference parameter, on the left side of an assignment statement.

Some component of a structured variable appears in an accessing reference, but no component of that variable has yet appeared in an assigning reference.

ACTION Make sure the variable is initialized.

---

536 MESSAGE LABEL "! " DECLARED, BUT NOT USED TO MARK ANY STATEMENT (536)

W CAUSE The label appears in a LABEL declaration, but is not used to mark any statement.

ACTION Remove label from LABEL declaration.

---

537 MESSAGE THIS PREVIOUSLY UNIMPLEMENTED FEATURE IS NOW IMPLEMENTED (537)

W	CAUSE	New functionality has been added of which the user should be aware.
	ACTION	None.

538	MESSAGE	THIS FEATURE REQUIRES $OS "! " (538)
W	CAUSE	The current $OS is not one that allows the feature.
	ACTION	Use the $OS specified in the message or remove the feature.

539	MESSAGE	THIS FEATURE REQUIRES $ STANDARD_LEVEL "! " (539)
W	CAUSE	The current standard_level is lower than that required for this feature.
	ACTION	Use the standard_level in the message or remove the feature.

540	MESSAGE	THIS FEATURE REQUIRES $STANDARD_LEVEL$ "! " AND $OS "! " (540)
W	CAUSE	The current standard_level is lower than that required for this feature and the $OS specified is wrong.
	ACTION	Use the standard_level and $OS specified in the message or remove the feature.

541	MESSAGE	FURTHER MESSAGES SUPPRESSED FOR THIS LINE (541)
W	CAUSE	Only 5 messages will be printed for any single input source line. If more than 5 messages are issued, then this will be the sixth and last message.
	ACTION	Remove the causes of the first 5 messages on this line, so that the next message can be printed.

549	MESSAGE	MISSING SEPARATOR, TEXT IGNORED UNTIL NEXT SEPARATOR (549)
W	CAUSE	When two or more compiler options are on the same line, the options must be separated by a semicolon or comma.
	ACTION	Add a separator if there are two or more compiler options.

550	MESSAGE	LOWER BOUND GREATER THAN UPPER, FOLDED TO EMPTY SUBRANGE (550)
W	CAUSE	Assigning or comparing a constant subrange with the lower bound greater than the upper bound results in an assignment or comparison of an empty set.
	ACTION	Correct the bounds.

551	MESSAGE	OBSOLETE !, USE '!' (551)
W	CAUSE	A feature supported by a previous release is now obsolete.

	ACTION	Change the source to use the recommended features and recompile.

552	MESSAGE	SYSTEMS LANGUAGE VARIABLE NOT SET (552)
W	CAUSE	JCW 'NLUSERLANG' or environment variable 'LANG' not set.
	ACTION	Set the system variable to the desired language (-LANG on HP-UX, NLUSERLANG on MPE/iX.)

553	MESSAGE	'!' and '!' ARE INCOMPATIBLE COMPILER OPTIONS (553)
W	CAUSE	These options are not compatible. The second was ignored.
	ACTION	Delete one of the options.

554	MESSAGE	DUPLICATE $SET FOR '!'; ITS VALUE IS NOW '!' (554)
W	CAUSE	This identifier was previously set by a $SET (or, on HP-UX, a -D on the command line). The value last seen takes effect.
	ACTION	Decide which value you want this identifier to have and remove all other $SETs (or, on HP-UX, remove the -D option on the command line if it is in error).

555	MESSAGE	VOLATILE VARIABLE PASSED BY REFERENCE (555)
W	CAUSE	A variable declared as volatile was used as an actual parameter in a routine call for which the formal parameter was a reference parameter.
	ACTION	The compiler cannot guarantee that the parameter will be properly updated, so you must ensure that it is.

556	MESSAGE	DEFAULT_PARM VALUES DO NOT MATCH THOSE IN FORWARD DECLARATION (556)
W	CAUSE	The values of constants for OPTION DEFAULT_PARMS do not match the corresponding values declared in a previous FORWARD declaration. The values used are those that were specified in the FORWARD declaration.
	ACTION	Ensure that the values are the same or leave off the formal parameter list from the routine heading when the routine is defined.

557	MESSAGE	PARAMETER TO PROCEDURE "NEW"MAY CAUSE A RUN-TIME ERROR (557)
W	CAUSE	The pointer argument to NEW is not aligned on a four-byte boundary. If the type of a pointer was defined with the ALIGNMENT compiler option and a component of a structured type contains this pointer type, a variable declared with this structured type may cause the pointer to be aligned improperly.
	ACTION	Create the variable so that the component used as an argument to NEW is four-byte aligned. This can be done by removing the

ALIGNMENT option from the type declaration for the pointer or by rearranging the fields of the record containing the pointer.

---

558 MESSAGE FILES APPEAR IN THE VARIANT PART OF A RECORD (558)

W CAUSE Fields of a file type or a structure containing a file type appear in the variant part of a record. When this variant becomes inactive, all fields in that variant are undefined. Furthermore, files corresponding to such fields are not guaranteed to be closed when the variant becomes inactive.

ACTION Make sure that such files are closed before deactivating the variant.

---

559 MESSAGE INVALID $SHLIB_VERSION DATE STRING (559)

W CAUSE The date string passed to the $SHLIB_VERSION compiler option is invalid.

ACTION The date string should be in the form: month/year. The year may be a 2 or 4 digit value. A month/year value representing a date earlier than January 1990 is invalid.

---

560 MESSAGE $HP_DESTINATION'ARCHITECTURE'$ IGNORED; FIRST SEEN TAKES EFFECT (560)

W CAUSE The compiler encountered more than one $HP_DESTINATION'ARCHITECTURE' option; only the first one seen will take effect.

ACTION Remove extra $HP_DESTINATION'ARCHITECTURE'$ options from the source file.

ACTION If you are specifying the +DA option to the pc command, remove the compiler option $HP_DESTINATION'ARCHITECTURE'$ from your source file.

---

561 MESSAGE $HP_DESTINATION'SCHEDULER'$ IGNORED; FIRST SEEN TAKES EFFECT (561)

W CAUSE The compiler encountered more than one $HP_DESTINATION'SCHEDULER' option; only the first one seen will take effect.

ACTION Remove extra $HP_DESTINATION'SCHEDULER'$ options from the source file.

ACTION If you are specifying the +DS option to the pc command, remove the compiler option $HP_DESTINATION'SCHEDULER'$ from your source file.

---

562 MESSAGE $OPTIMIZE 'BASIC_BLOCKS *num* '$: *num* was omitted; using zero. (562)

W CAUSE You inadvertently omitted the number which specifies the threshold of basic blocks in a procedure which you want optimized at level 2. Zero was inserted by the compiler, which effectively disables the basic blocks feature; every procedure is optimized at level 2.

	ACTION	Specify the *num* in the $OPTIMIZE 'BASIC_BLOCKS *num* '$ directive.
	ACTION	On HP-UX, specify the *num* on the command-line with +Obb*num*.
	ACTION	On HP-UX, specify 0 as *num* to guarantee the "old" -0 behavior (that is, not ever dropping down to level 1 optimization).

568	MESSAGE	'+' IS NOT ALLOWED HERE (568)
W	CAUSE	A '+' was specified as part of a $SEARCH compiler option, but it did not precede all the file names in the search list.
	ACTION	Correct the compiler option.

569	MESSAGE	NO ASSEMBLY FILE FOUND. LIST_CODE NOT PERFORMED (569)
W	CAUSE	The compiler could not find the file with the assembly listing (usually "???.s" on HP-UX or "PASASSM" on MPE/iX). On MPE/XL;, this usually happens because the PASASSM file is too small to hold the assembly output (the default size is 40,000 records). Another possible reason is that you have hit some file system limit like total file space or number of files.
	ACTION	If you have run against a system limit, get around it and recompile. On MPE/iX, if you can determine that you have not run afoul of a system limit, try the following file equation: FILE PASASSM;DISC=100000 Modify the parameter of the "DISC=" option according to how big you think your compilation unit is.

570	MESSAGE	PARAMETER TYPE NOT SUPPORTED BY EXTERNAL LANGUAGE (570)
W	CAUSE	An ANYVAR or READONLY parameter is used with an EXTERNAL C or EXTERNAL FTN77 directive. These types of parameters are not supported by these languages.
	ACTION	Remove parameter or change TYPE to VAR.

571	MESSAGE	INCOMPATIBLE COMPILER OPTIONS PFA AND OPTIMIZE (571)
W	CAUSE	Both OPTIMIZE and PFA options are present. The options are mutually exclusive.
	ACTION	(1) If PFA is desired, remove OPTIMIZE. (2) If OPTIMIZE is desired, remove PFA.

572	MESSAGE	INCOMPATIBLE COMPILER OPTIONS OPTIMIZE AND SYMDEBUG (572)
W	CAUSE	Both OPTIMIZE and SYMDEBUG 'XDB' options are present. The options are mutually exclusive.
	ACTION	(1) If SYMDEBUG 'XDB' is desired, remove OPTIMIZE. (2) If OPTIMIZE is desired, remove SYMDEBUG 'XDB'.

573	MESSAGE	INCOMPATIBLE COMPILER OPTIONS SYMDEBUG AND OPTIMIZE (572)

W	CAUSE	Both OPTIMIZE and SYMDEBUG 'TOOLSET' options are present. The options are mutually exclusive.
	ACTION	(1) If SYMDEBUG 'TOOLSET' is desired, remove OPTIMIZE. (2) If OPTIMIZE is desired, remove SYMDEBUG 'TOOLSET'.

575	MESSAGE	VALUE OF ESCAPECODE IS QUESTIONABLE HERE (575)
W	CAUSE	The value returned by ESCAPECODE outside a RECOVER construct is undefined.
	ACTION	Store off escape code into a local variable from inside the RECOVER construct, and use the local variable outside it.

576	MESSAGE	POINTER FIELD IN OTHER VARIANT NOW UNDEFINED (576)
W	CAUSE	An integer field overlaying a pointer field has been assigned to making the pointer undefined.
	ACTION	None - informational message only.

577	MESSAGE	ASSUME "! " IS NOT VALID, REMOVE $ASSUME$ (577)
W	CAUSE	A construct is used that invalidates the given assumption which the compiler ignores.
	ACTION	Remove the $ASSUME$ option that is invalid.

582	MESSAGE	$HP3000_32$ NOT RECOGNIZED, OPTION IGNORED (582)
W	CAUSE	$HP3000_32$ is not recognized because $HP30000_16$ has not been set.
	ACTION	Remove $HP3000_32$

584	MESSAGE	INVALID MODULE LIBRARY NAME SPECIFIED (584)
W	CAUSE	Specified a module library which cannot be opened by the system.
	ACTION	Check the name of the module library.

586	MESSAGE	INVALID ALIGNMENT VALUE, OPTION IGNORED (586)
W	CAUSE	The alignment specified was not one of 1, 2, 4, 8, 16, 32, 64, or 2048 bytes.
	ACTION	Correct the alignment value.

587	MESSAGE	UNSUPPORTED VARIABLE ALIGNMENT REQUESTED (587)
W	CAUSE	The type declaration specified an alignment value that is not supported for static variables.

	ACTION	Correct the alignment value.

588	MESSAGE	POSSIBLE USE OF UNINITIALIZED FIELD '!' OF '!' (588)
W	CAUSE	The field of the local variable mentioned in the message may be uninitialized when used in this procedure or function.
	ACTION	Ensure that the field is initialized before use.

590	MESSAGE	IDENTIFIER ‘!' OVERLOADED BY IMPORTED MODULE(S) (590)
W	CAUSE	An identifier with the same spelling is exported by an earlier imported module.
	ACTION	Rename one of the identifiers. If you do not rename an identifier, the identifier in the last imported module will be used.

591	MESSAGE	COUNT IS NEGATIVE; NO DATA WILL BE MOVED (591)
W	CAUSE	The move count parameter to a MOVE procedure will always be negative, thus no data will be moved.
	ACTION	Make sure that the count is supposed to be negative.

592	MESSAGE	LONG TO SHORT POINTER CONVERSION EMITTED IN STATEMENT "! " (592)
W	CAUSE	A 64 bit address was converted to a 32 bit address. Only addresses that are in space registers 4 through 7 can be converted without an error.
	ACTION	A run-time trap will occur if the address is not valid.
	ACTION	On MPE/iX, make sure short addresses in SR4 are not passed to an executable library (XL). They may not trap until dereferenced.

593	MESSAGE	TYPE COERCION ALTERS NUMBER OF STORAGE UNITS (593)
W	CAUSE	Source and target types require a different number of storage units; thus, code generated as a result of this type coercion may not behave as expected.
	ACTION	Remove the type coercion expression or define the TYPE_COERCION level to be 'NONCOMPATIBLE'.

594	MESSAGE	IMPLEMENT MISSING FOR MODULE "! " (594)
W	CAUSE	No IMPLEMENT appeared in the given MODULE.
	ACTION	Supply an IMPLEMENT section.

595	MESSAGE	EXPORT QUALIFICATIONS NOT IMPLEMENTED (595)

W	CAUSE	EXPORT qualifiers currently have no effect.
	ACTION	No action is required.

596	MESSAGE	DUPLICATE IMPORTED MODULE (596)
W	CAUSE	<IDENT1> ! <IDENT2> is the same as <IDENT2>.
	ACTION	Rename one of the modules.

597	MESSAGE	POSSIBLE USE OF UNINITIALIZED VARIABLE '!' (597)
W	CAUSE	The local variable mentioned in the message may be uninitialized when used in this procedure or function.
	ACTION	Ensure that the variable is initialized before use.

598	MESSAGE	RESULTS OF $GLOBAL/$RLFILE/$SUBPROGRAM IS DIFFERENT ON MPE V (598)
W	CAUSE	If a compilation has $GLOBAL, $RLFILE, and $SUBPROGRAM set, the result of the compile will be different than if it was done on MPE V (no outer block information is output).
	ACTION	Remove either $GLOBAL or $SUBPROGRAM.

599	MESSAGE	POSSIBLE PARAMETER ADDRESS ALIGNMENT MISMATCH (599)
W	CAUSE	A VAR parameter of unknown alignment is being passed as a reference parameter to an INTRINSIC which has a strict alignment requirement for that parameter. If the actual parameter has a less restrictive alignment than that required by the intrinsic, an AD DRESS ALIGNMENT error will occur.
	ACTION	Ensure that the actual parameter has the alignment required by the INTRINSIC.

600	MESSAGE	INSUFFICIENT HEAP AREA TO ALLOCATE VARIABLE (PASCERR 600)
RT	CAUSE	The heap is full.
	ACTION	Increase the amount of heap space for the program or decrease the storage used by the program.

601	MESSAGE	INVALID DISPOSE PARAMETER (PASCERR 601)
RT	CAUSE	The pointer parameter to DISPOSE is NIL.
		The pointer parameter to DISPOSE does not identify any area allocated by NEW.
	ACTION	Initialize the pointer with NEW before disposing.
	CAUSE	The pointer parameter to DISPOSE identifies an area previously deallocated by release.
	ACTION	Do not DISPOSE a pointer that has been released.

602	MESSAGE	REPEATED USE OF DISPOSE ON GIVEN PARAMETER (PASCERR 602)
RT	CAUSE	The pointer parameter to dispose identifies an area previously deallocated by dispose.
	ACTION	Do not DISPOSE a pointer that has been released.
603	MESSAGE	DISPOSE PARAMETER ALLOCATED AS DIFFERENT VARIANT (PASCERR 603)
RT	CAUSE	The pointer parameter to dispose identifies an area allocated by new with a different sequence of case constants.
	CAUSE	The pointer parameter to dispose includes case constants, but it identifies an area allocated by new without any case constants.
	CAUSE	The pointer parameter to dispose does not include case constants, but it identifies an area allocated by new with case constants.
	ACTION	Make sure that any tags associated with DISPOSE match the tags on NEW. Also check for heap corruption.
604	MESSAGE	DISPOSE PARAMETER CONTAINS AN OPEN SCOPE (PASCERR 604)
RT	CAUSE	The pointer parameter to dispose identifies an area containing an actual variable parameter, an element of the record variable list of a WITH statement, or both.
	ACTION	Make sure that the identifier does not reference such an area.
605	MESSAGE	INVALID RELEASE PARAMETER (PASCERR 605)
RT	CAUSE	The parameter to RELEASE was not set by a previous call to MARK.
	ACTION	Initialize the parameter with MARK.
	CAUSE	The parameter to RELEASE was set by a call to MARK, but a previous call to RELEASE has been made with this parameter.
	ACTION	Get rid of one of the uses of MARK.
	CAUSE	The parameter to RELEASE was set by a call to MARK, but that call to MARK was preceded by a call to MARK with a different parameter that has already been used as a parameter to RELEASE.
	ACTION	Don't use RELEASE on already released space.
606	MESSAGE	RELEASE PARAMETER ENCLOSES AN OPEN SCOPE (PASCERR 606)
RT	CAUSE	The parameter to release identifies an area containing an actual variable parameter, an element of the record variable list of a WITH statement, or both.
	ACTION	Make sure that the identifier does not reference such an area.

607 MESSAGE RELEASE PARAMETER ENCLOSES GETHEAP AREA(S) (PASCERR 607)

RT CAUSE The parameter to release identifies an area containing areas the user allocated with GETHEAP procedure, but which have not yet been deallocated with the RTNHEAP procedure.

ACTION RTNHEAP must be used to release areas allocated by GETHEAP.

---

608 MESSAGE HEAP INTEGRITY LOST / HEAP DATA LOST (PASCERR 608)

RT CAUSE The internal data structures of the heap have become inconsistent. The most likely causes are:

1. A field has been assigned to in a variant different than the one specified in a call to new.
2. A pointer to a disposed area (for example, a dangling pointer) has been dereferenced in an assignment.
3. An SPL routine has directly accessed the DL-DB area outside of a region allocated by the GETHEAP procedure.
4. The DLSIZE intrinsic has been called.
5. The RTNHEAP procedure was unable to return an area.

ACTION Verify that none of the likely causes have occurred.

---

609 MESSAGE BAD ALIGNMENT (PASCERR 609)

RT CAUSE A call to new or dispose passed a bad value for the alignment parameter; for example, the type to which the pointer points has an alignment which is not recognized by NEW or DISPOSE. The only legal values for the alignment are 1, 2, 4, 8, 16, and 2048.

ACTION Ensure that the type to which the pointer points has an alignment which is one of the above.

CAUSE A call to P_GetHeap or P_RtnHeap passed a bad value for the alignment parameter.

ACTION Give a correct alignment value.

---

610 MESSAGE BAD SIZE (PASCERR 610)

RT CAUSE A call to new or dispose passed a bad value for the size of the area.

ACTION None. Usually an internal error.

CAUSE A call to GetHeap or RtnHeap passed a bad value for the size of the area.

ACTION Change the size parameter.

---

611 MESSAGE HEAP INTEGRITY LOST / HEAP DATA LOST (PASCERR 608)

RT CAUSE The internal data structures of the heap have become inconsistent. The most likely causes are:

1. A field has been assigned to in a variant different than the one specified in a call to new.

2. A pointer to a disposed area, such as a dangling pointer, has been dereferenced in an assignment.

3. There is a mismatch of data types. Check to see that the routine calling NEW or GETHEAP uses the same declaration for the pointer as the routine which makes an assignment through it (for separate compilations).

ACTION According to above causes.

---

620 MESSAGE VALUE NOT WITHIN SUBRANGE (PASCERR 620)

RT CAUSES The value of an ordinal expression is outside of the subrange of the target of an assignment statement.

The value of an ordinal expression appearing as an actual parameter is outside the subrange of the formal value parameter.

The value of an ordinal expression appearing in an array selector is outside of the subrange of the index type.

ACTION Ensure that the value is within the subrange.

---

621 MESSAGE NO CASE LABEL FOR SELECTOR VALUE (PASCERR 621)

RT CAUSE The value of the CASE select expression does not match any of the specified CASE constants and no OTHERWISE clause appears.

ACTION Add a CASE to handle the value that caused the error, or add an OTHERWISE clause to handle the value, or change the program logic so the value of the selector corresponds with one of the CASE labels.

---

622 MESSAGE INVALID POINTER (PASCERR 622)

RT CAUSE A pointer with the value of NIL was dereferenced.

A pointer with an undefined value was dereferenced.

A pointer set by MARK was dereferenced.

A pointer identifying an area previously deallocated was dereferenced.

ACTION Correct the program logic.

---

623 MESSAGE VALUE OF PRED UNDEFINED (PASCERR 623)

RT CAUSE The minimum value of an ordinal type or subrange was the parameter to PRED. The result is undefined.

ACTION Do not call PRED with the lowest value of an ordinal type.

---

624 MESSAGE VALUE OF SUCC UNDEFINED (PASCERR 624)

RT	CAUSE	The maximum value of an ordinal type or subrange was the parameter to SUCC. The result is undefined.
	ACTION	Do not call SUCC with the highest value of an ordinal type.

625	MESSAGE	SET RANGE ERROR (PASCERR 625)
RT	CAUSE	An attempt was made to assign a set to a set variable when the set contains an element not within the set range of the variable. An attempt was made to pass a set to a formal parameter when the set contains an element not within the set range of the parameter.
	ACTION	Correct the program logic.

626	MESSAGE	ATTEMPT TO DO MOD BY A VALUE LESS THAN OR EQUAL TO ZERO (PASCERR 626)
RT	CAUSE	An attempt was made to perform the MOD operation when the right operand is zero or negative.
	ACTION	Correct the program logic error that has caused the invalid value to be used. Note that MOD is not the remainder operator.

627	MESSAGE	SQRT CALLED WITH NEGATIVE ACTUAL PARAMETER (PASCERR 627)
RT	CAUSE	The value passed to the SQRT function is less than zero.
	ACTION	Only call SQRT with non-negative values.

628	MESSAGE	LN CALLED WITH NON-POSITIVE ACTUAL PARAMETER (PASCERR 628)
RT	CAUSE	The value passed to the LN function is less than or equal to zero.
	ACTION	Only call LN with positive values.

640	MESSAGE	BAD PROCEDURAL PARAMETER (PASCERR 640)
RT	CAUSE	A nonlevel 1 procedure or function was passed as a procedural or functional parameter to an external, non-HP Pascal routine.
	ACTION	Only level 1 procedures/functions can be passed.

650	MESSAGE	STRING OVERFLOW (PASCERR 650)
RT	CAUSE	An attempt was made to index beyond the maximum length of the string.
	ACTION	Correct the string operation, standard procedure or function call arguments, or the program logic.

651	MESSAGE	STRING INDEX EXCEEDS CURRENT LENGTH (PASCERR 651)

RT	CAUSE	An attempt was made to index beyond the current length of the string.
	ACTION	Correct the argument or the program logic.

652	MESSAGE	DESIGNATED CHARACTER POSITION(S) OUTSIDE STRING (PASCERR 652)
RT	CAUSE	The specified offset is greater than the current length of the string, or less than 1.
	ACTION	Correct either the argument or the program logic.

653	MESSAGE	DESIGNATED CHARACTER POSITION(S) OUTSIDE PAC (PASCERR 653)
RT	CAUSE	The specified offset is greater than the upper bound of the PAC.
	ACTION	Correct the program logic that has caused the invalid value to be used; change the value that has caused the error to a legitimate value. Also check the type definition.

654	MESSAGE	ATTEMPT TO READ PAST END OF STRING (PASCERR 654)
RT	CAUSE	Attempt was made to read beyond the maximum length of the string.
	ACTION	Correct the problem that is causing the read past the end of the string.

655	MESSAGE	INVALID NUMBER OF CHARACTERS SPECIFIED (PASCERR 655)
RT	CAUSE	The number of characters to be copied, moved, or deleted in the predefined string procedure STRMOVE is less than zero.
	ACTION	Correct the problem that is generating the negative count.

670	MESSAGE	INVALID CHARACTER FOR HEX DIGIT (PASCERR 670)
RT	CAUSE	The character was not in the set 0..9, A..F, or a..f.
	ACTION	Correct the argument to the numeric conversion function to contain only valid characters in the particular base.

671	MESSAGE	INVALID CHARACTER FOR OCTAL DIGIT (PASCERR 671)
RT	CAUSE	The character was not in the set 0..7.
	ACTION	Correct the argument to the numeric conversion function to contain only valid characters in the particular base.

672	MESSAGE	INVALID CHARACTER FOR BINARY DIGIT (PASCERR 672)
RT	CAUSE	The character was not in the set 0..1.
	ACTION	Correct the argument to the numeric conversion function to

contain only valid characters in the particular base.

---

673 MESSAGE NUMBER OF SIGNIFICANT DIGITS CAUSED OVERFLOW (PASCERR 673)

RT CAUSE The number of significant digits was more than 32 for the standard function BINARY, 11 for the function OCTAL, or 8 for the function HEX.

ACTION Correct the argument to the numeric conversion function to be a representable value.

---

690 MESSAGE OPEN ERROR: PHYSICAL FILE COULD NOT BE CLOSED (PASCERR 690)

RT CAUSE An attempt was made to open a file, but the logical file was already associated with a physical file and this physical file could not be closed prior to opening another physical file.

ACTION Find out why the file could not be closed.

---

691 MESSAGE OPEN ERROR: MISMATCH OF LOGICAL/PHYSICAL FILES (PASCERR 691)

RT CAUSE The characteristics of the logical file are not compatible with those of the associated physical file. For example, a physical file with variable length records may not be opened for direct access.

ACTION Check to make sure that the file characteristics are compatible.

---

692 MESSAGE FILE OPEN ERROR (PASCERR 692)

RT CAUSE An unsuccessful attempt was made to open a file. The file was absent or exclusively accessed, or you did not have permission to access the file.

ACTION Check for file's presence and its access protections, and also the state of the file when the open is attempted.

---

693 MESSAGE ERROR OCCURRED WHILE READING FROM FILE (PASCERR 693)

RT CAUSE File system failure or corrupted Pascal FILE variable.

ACTION Correct the file system problem, or correct program error that corrupted Pascal FILE variable, such as array reference out of bounds with RANGE OFF or dereferencing an invalid pointer.

---

694 MESSAGE ATTEMPT TO READ PAST EOF (PASCERR 694)

RT CAUSE The current position is past the last component of the file.

ACTION Correct the program logic to check EOF before reading file data or checking EOLN status. For a direct access file, check that disk record to be read is not greater than MAXPOS.

695	MESSAGE	ERROR OCCURRED WHILE WRITING TO FILE (PASCERR 695)
RT	CAUSE	A Pascal FILE variable has been corrupted.
	ACTION	Correct the file system problem or program error that is corrupting the HP Pascal file such as an array out of bounds with RANGE OFF or dereferencing an invalid pointer.
	CAUSE	An attempt is made to write past the physical unit of the file.
	ACTION	Increase the file's physical limit.

696	MESSAGE	WRITE ON READ-ONLY FILE (PASCERR 696)
RT	CAUSE	An attempt was made to perform an output operation on a file opened for input access only.
	ACTION	Correct the program logic so it doesn't write to the file or open the file in a way that permits writing (such as REWRITE, APPEND, or OPEN.) Scratch files can only be created by opening them in a way that permits writing.

697	MESSAGE	OPEN ERROR: UNABLE TO INITIALIZE POSITION (PASCERR 697)
RT	CAUSE	A request was made to open a logical file already associated with the physical file. However, the file pointer was unable to be repositioned at the beginning of the physical file.
	ACTION	See if program logic is corrupting the Pascal FILE variable.

698	MESSAGE	OPEN ERROR: UNABLE TO EMPTY FILE (PASCERR 698)
RT	CAUSE	REWRITE was unable to empty the file of its previous contents.
	ACTION	Check if program logic is corrupting the Pascal FILE variable. Otherwise, it is a file system problem.

699	MESSAGE	UNABLE TO CLOSE FILE (PASCERR 699)
RT	CAUSE	The file could not be closed as requested.
	ACTION	Check if you have save permission on your system or make sure you have used the CLOSE command to close the file.

700	MESSAGE	ERROR OCCURRED DURING DIRECT ACCESS I/O (PASCERR 700)
RT	CAUSE	An error occurred during a file operation on a direct access file.
	ACTION	Check if you are specifying a record beyond the file's physical limit.

701	MESSAGE	ILLEGAL CHARACTER IN NUMBER (PASCERR 701)
RT	CAUSE	An attempt was made to read a number from a text file, but an illegal character was found before a valid number.

	ACTION	Correct the input.

702	MESSAGE	INPUT VALUE OVERFLOW (PASCERR 702)
RT	CAUSE	The numeric value read is too large for the type of the variable.
	ACTION	Correct the input.

703	MESSAGE	ATTEMPT TO WRITE PAST PHYSICAL BOUNDS OF FILE (PASCERR 703)
RT	CAUSE	The current record position is past the physical limit of the file.
	ACTION	Create a larger size file and re-run the program.

704	MESSAGE	READ ATTEMPTED FROM OUTPUT FILE (PASCERR 704)
RT	CAUSE	An attempt was made to perform an input operation on a file opened only for output.
	ACTION	Correct the program logic so it doesn't read from the file or open the file in a way that permits reading (such as RESET or OPEN.)

705	MESSAGE	FILE NOT OPENED FOR DIRECT ACCESS (PASCERR 705)
RT	CAUSE	An attempt was made to perform a direct access file operation on a file not opened for direct access with the OPEN procedure.
	ACTION	A nontext file must be opened for direct access with OPEN to use SEEK, READDIR, WRITEDIR, or POSITION.

706	MESSAGE	FILE NOT OPENED (PASCERR 706)
RT	CAUSE	An attempt was made to access an unopened file.
	ACTION	Correct the program logic so it doesn't read from the file or open the file in a way that permits reading (such as RESET or OPEN.)

707	MESSAGE	INVALID OPEN OPTION (PASCERR 707)
RT	CAUSE	An invalid option was found in the third parameter to one of the file opening procedures.
	ACTION	Correct the option.

708	MESSAGE	COULD NOT OPEN FILE FOR APPEND ACCESS (PASCERR 708)
RT	CAUSE	A file system failure or corrupted Pascal FILE variable prevented opening a variable length record file for append access.
	ACTION	Either correct the file system problem or correct the program

		error that corrupted the Pascal FILE variable (such as array reference out of bounds with RANGE OFF or dereferencing an invalid pointer.)
709	MESSAGE	FIELD WIDTH LESS THAN ZERO (PASCERR 709)
RT	CAUSE	The field width in a formatted write of a nonnumeric expression was less than zero.
	ACTION	Correct the program logic so it doesn't use negative values for the field width or decimal position.
710	MESSAGE	FIELD WIDTH LESS THAN 1 (PASCERR 710)
RT	CAUSE	The field width in the formatted write of a numeric expression was less than 1.
	ACTION	Correct the width specified.
711	MESSAGE	NO DIGITS AFTER DECIMAL POINT (PASCERR 711)
RT	CAUSE	No digits occur after the decimal point in a formatted write of a real or longreal expression.
	ACTION	Correct the input.
712	MESSAGE	INPUT VALUE UNDERFLOW (PASCERR 712)
RT	CAUSE	The value read is too small to be represented in the variable.
	ACTION	Correct the input.
713	MESSAGE	FIELD TOO SMALL TO PRINT NUMBER (PASCERR 713)
RT	CAUSE	This is an internal HP PASCAL error.
	ACTION	Contact Hewlett-Packard.
714	MESSAGE	INVALID CLOSE OPTION (PASCERR 714)
RT	CAUSE	An invalid disposition option was found in the second parameter to CLOSE.
	ACTION	Correct the option.
715	MESSAGE	INVALID ENUMERATED IDENTIFIER FOR INPUT (PASCERR 715)
RT	CAUSE	An attempt was made to read an enumerated identifier from a textfile, but either a valid HP Pascal identifier was not found or the identifier found was not an identifier of that enumerated type.
	ACTION	Correct the input.

---

716 MESSAGE CANNOT WRITE ENUMERATED VALUE (PASCERR 716)

RT CAUSE An attempt was made to write an enumerated variable to a textfile, but the current ordinal value of the variable is not within the range of the enumerated type.

ACTION Check the program's logic.

---

717 MESSAGE INVALID BOOLEAN READ (PASCERR 717)

RT CAUSE An attempt was made to read a Boolean value from a textfile, but a non-boolean value was found.

ACTION Correct the input.

---

718 MESSAGE INVALID FLOATING POINT NUMBER REPRESENTATION (PASCERR 718)

RT CAUSE An attempt was made to read a real or longreal number from a textfile, but an invalid floating point number was found.

ACTION Correct the program's logic to read the real or longreal from the correct place in the file or string, verify that the correct file or string is being accessed, or correct the corrupted file or string.

---

719 MESSAGE INVALID CALL TO EOLN (PASCERR 719)

RT CAUSE The EOLN function was called for a file positioned at end-of-file. An end-of-line marker precedes the end-of-file in every text file, but this final end-of-line marker had already been read past.

ACTION Check for end-of-file before calling EOLN.

---

720 MESSAGE UNABLE TO LOCK FILE (PASCERR 720)

RT CAUSE An attempt was made to lock a file without specifying the lock option in the call to open. This error should never occur since in HP Pascal the only way to lock a file is by specifying this lock option.

ACTION None

---

721 MESSAGE WRITE FIELD WIDTH TOO LARGE (PASCERR 721)

RT CAUSE Either an attempt was made to write a number with a field width greater than 254 characters, or an attempt was made to write a longreal in fixed point format which would result in an excessive number of digits being printed.

ACTION Reduce the field width if it is greater than 254 characters. Write large longreals in floating point format.

---

722 MESSAGE CANNOT "ASSOCIATE" FILE OPENED BY A PASCAL ROUTINE (PASCERR

		722)
RT	CAUSE	An attempt was made to associate a file that was not opened with a system provided open routine. Instead, the file was opened with a PASCAL open routine.
	ACTION	Open the file with a system provided open routine such as MPE/iX "FOPEN" or HP-UX "OPEN" before using "associate."
723	MESSAGE	MISSING OPTIONS TO "ASSOCIATE" (PASCERR 723)
RT	CAUSE	The option string passed to the associate routine was empty.
	ACTION	Pass the appropriate options to the associate routine.
724	MESSAGE	INVALID OPTIONS TO "ASSOCIATE" (PASCERR 724)
RT	CAUSE	An illegal combination of options were passed to "associate."
	ACTION	Pass a legal set of options to "associate."
725	MESSAGE	LOGICAL FILE PREVIOUSLY ASSOCIATED OR OPENED (PASCERR 725)
RT	CAUSE	An attempt was made to associate a logical file name to a physical file number. However, the file name is already on the Pascal open file list. It was placed on the list during a previous "associate" or "open." If the file is not disassociated or close d, any subsequent attempt to associate it will fail.
	ACTION	Close the file using the Pascal "close" routine or disassociate the file using the Pascal "disassociate" routine.
799	MESSAGE	INVALID OPERATING SYSTEM I/O (PASCERR 799)
RT	CAUSE	An attempt was made to perform some kind of I/O which is illegal on this Operating System. This error will never occur for normal users.
	ACTION	Contact Hewlett-Packard.
808	MESSAGE	COERCION REQUIRES $TYPE_COERCION 'STRUCTURAL'$ (808)
CT	CAUSE	The current $TYPE_COERCION 'string'$ is insufficient to permit this type coercion.
	ACTION	Set the type_coercion level to that given in the message.
809	MESSAGE	COERCION REQUIRES $TYPE_COERCION 'REPRESENTATION'$ (809)
CT	CAUSE	The current $TYPE_COERCION 'string'$ is insufficient to permit this type coercion.
	ACTION	Set the type_coercion level to that given in the message.

810	MESSAGE	COERCION REQUIRES $TYPE_COERCION 'STORAGE'$ (810)
CT	CAUSE	The current $TYPE_COERCION 'string'$ is insufficient to permit this type coercion.
	ACTION	Set the type_coercion level to that given in the message.

811	MESSAGE	COERCION REQUIRES $TYPE_COERCION 'NONCOMPATIBLE'$ (811)
CT	CAUSE	The current $TYPE_COERCION 'string'$ is insufficient to permit this type coercion.
	ACTION	Set the type_coercion level to that given in the message. This is very dangerous coding practice.

813	MESSAGE	MULTIPLE DEFINITIONS FOR THIS MODULE (813)
CT	CAUSE	A definition for this module identifier has already been compiled within this compilation unit.
	ACTION	Delete extra module definition from the compilation unit.

814	MESSAGE	MISSING EXPORT SECTION (814)
CT	CAUSE	A module must have an EXPORT section.
	ACTION	Define an EXPORT section for this module.

Messages 816-7999

816	MESSAGE	INVALID IMPORT MODULE SPECIFIED (816)
CT	CAUSE	The IMPORT module specified could not be found.
	ACTION	Check $SEARCH path for missing files or check the module name.
	CAUSE	The module name is a duplicate of an identifier previously defined.
	ACTION	Rename either the module name or the identifier.

817	MESSAGE	INVALID MODULE IDENTIFIER (817)
CT	CAUSE	The identifier is not a module identifier.
	ACTION	Check identifier for misspellings.

818	MESSAGE	NOT EXPORTED BY THE QUALIFYING IMPORTED MODULE (818)
CT	CAUSE	The identifier was not exported by the qualifying imported module or defined in the module currently being defined.
	ACTION	Check the identifier for misspellings.

819	MESSAGE	TYPE COERCION PERMITTED FOR DATA ITEMS ONLY (819)

CT	CAUSE	There was an attempt to type coerce NIL. There was an attempt to type coerce a procedure name.
	ACTION	Remove the type coercion.

820	MESSAGE	BIAS IS LESS THAN MINIMUM ARRAY INDEX (820)
CT	CAUSE	The bias parameter to a MOVE procedure will always cause an index range error before the move is completed.
	ACTION	Fix the bias parameter or count parameter.

821	MESSAGE	BIAS + COUNT IS GREATER THAN MAXIMUM ARRAY INDEX (821)
CT	CAUSE	The bias and move count parameters to a MOVE procedure will always cause an index range error before the move is completed.
	ACTION	Fix the bias parameter or count parameter.

822	MESSAGE	BIAS IS NOT ASSIGNMENT COMPATIBLE WITH ARRAY INDEX TYPE (822)
CT	CAUSE	A bias parameter of a type that is not assignment compatible to the index type of an array parameter to a MOVE procedure was specified.
	ACTION	Fix the bias parameter to be of the same type as the index of the array.

823	MESSAGE	TARGET ELEMENT TYPE DOES NOT MATCH SOURCE ELEMENT TYPE (823)
CT	CAUSE	Element type of the source and target parameters to a MOVE procedure must be identical.
	ACTION	Use a different mechanism to move data.

824	MESSAGE	ACTUAL PARAMETER MUST BE AN ARRAY (824)
CT	CAUSE	The source or target parameter to a MOVE procedure is not an array type, which it must be.
	ACTION	Declare the type as an array or coerce the parameter to an array type.

825	MESSAGE	A CRUNCHED STRUCTURE IS REQUIRED HERE (825)
CT	CAUSE	Any structures nested within a crunched structure must also be crunched.
	ACTION	Declare the inner structure "crunched".

826	MESSAGE	INVALID TYPE FOR COMPONENT OF A CRUNCHED STRUCTURE (826)
CT	CAUSE	Crunched structures may only have components of certain types.

	ACTION	Consult the *HP Pascal Reference Manual* for details.

828	MESSAGE	MISSING DEFAULT VALUE FOR "! " (828)
CT	CAUSE	This parameter requires a default value to be specified.
	ACTION	Supply a default value in the "default_parms" option.

831	MESSAGE	ROUTINE OPTION NOT COMPATIBLE WITH PREVIOUS ONE(S) (831)
CT	CAUSE	A routine was declared with two routine options that are incompatible.
	ACTION	Re-evaluate the requirements for the routine options.

832	MESSAGE	PROCEDURE NESTING TOO GREAT FOR THIS ROUTINE OPTION (832)
CT	CAUSE	A level 2 or greater routine was declared with a routine option that is illegal at a level greater than 1.
	ACTION	Either make the routine level 1 or remove the routine option.

833	MESSAGE	INVALID ROUTINE OPTION (833)
CT	CAUSE	A routine option was declared that is not a known routine option.
	ACTION	Check the spelling.

834	MESSAGE	INVALID EXTENSIBLE PARAMETER COUNT (834)
CT	CAUSE	The count value in an Extensible routine is either less than "0" or greater than the number of parameters in the routine.
	ACTION	Provide a legitimate count.

835	MESSAGE	THIS FORM PERMITTED ONLY IN ROUTINE OPTION (835)
CT	CAUSE	A keyword value assignment to a formal parameter was used outside of a definition option.
	ACTION	Remove the keyword assignment and assign by position.

836	MESSAGE	THIS FORM NOT PERMITTED IN ROUTINE OPTION (836)
CT	CAUSE	An empty parameter was specified in a routine option or the parameter was an expression.
	ACTION	Either supply a value or replace the expression with a constant.

837	MESSAGE	INVALID FORMAL PARAMETER FOR THIS ROUTINE OPTION (837)

CT	CAUSE	A routine option specified a formal parameter that was not declared in the formal parameter list.
	ACTION	Check the formal parameter list.

838	MESSAGE	DUPLICATE FORMAL PARAMETER FOR THIS ROUTINE OPTION (838)
CT	CAUSE	A routine option specified a formal parameter twice.
	ACTION	Remove the duplicate specification.

839	MESSAGE	ROUTINE OPTION AND FORMAL PARAMETER ORDERING MISMATCH (839)
CT	CAUSE	The order of parameters in a routine option does not match the ordering of the formal parameters in the formal parameter list.
	ACTION	Fix the routine option or match the ordering.

841	MESSAGE	DEFAULT VALUE FOR VARIABLE FORMAL PARAMETER IS NOT NIL (841)
CT	CAUSE	A VAR formal parameter was assigned a default value that is not NIL.
	ACTION	Assign the value NIL to the VAR parameter.

842	MESSAGE	DEFAULT VALUE NOT COMPATIBLE WITH FORMAL PARAMETER (842)
CT	CAUSE	A parameter was assigned a default value whose type does not match the type of the formal parameter.
	ACTION	Fix the default value.

844	MESSAGE	ILLEGAL USE OF READONLY VARIABLE OR PARAMETER (844)
CT	CAUSE	A READONLY variable or parameter was used as the target of an assignment statement or was passed as a VAR parameter.
	ACTION	Remove the offending use of the READONLY variable or parameter.

845	MESSAGE	INVALID USE OF ROUTINE OPTION (845)
CT	CAUSE	The routine option is not allowed in this context.
	ACTION	Remove the routine option.

846	MESSAGE	NOT A FORMAL PARAMETER (846)
CT	CAUSE	A formal parameter specified in a routine option is not declared in the formal parameter list.
	ACTION	Check the spelling. Remove the parameter in the routine option. Add the parameter to the formal parameter list.

847	MESSAGE	NOT A VARIABLE DEFAULT FORMAL PARAMETER (847)
CT	CAUSE	A formal parameter to the Haveoptvarparm function is not a VAR or ANYVAR parameter.
	ACTION	Check the formal parameter list. Remove this call to Haveoptvarparm.

848	MESSAGE	NOT AN EXTENSION FORMAL PARAMETER (848)
CT	CAUSE	A formal parameter to the Haveextparm function is not an extensible parameter.
	ACTION	Remove this call or check the count on the Extensible routine option.

849	MESSAGE	THIS ROUTINE OPTION NOT VALID FOR FUNCTIONS (849)
CT	CAUSE	The specified routine option is not allowed for a function.
	ACTION	Remove this routine option.

850	MESSAGE	RECURSIVE USE OF INLINE PROCEDURE/FUNCTION NOT ALLOWED (850)
CT	CAUSE	A routine declared OPTION INLINE directly or indirectly calls itself recursively.
	ACTION	Remove the recursion or remove the OPTION INLINE.

851	MESSAGE	THIS DIRECTIVE NOT ALLOWED WITH ROUTINE OPTIONS (851)
CT	CAUSE	A routine directive was declared for a routine that has definition options.
	ACTION	Remove the directive or the option.

852	MESSAGE	NOT A DEFAULT FORMAL PARAMETER (852)
CT	CAUSE	A formal parameter supplied to the Haveoptvarparm function is not a default parameter.
	ACTION	Remove this call or check the list of default parameters.

856	MESSAGE	AN ADDRESS CAN NOT BE GENERATED FOR THIS VARIABLE (856)
CT	CAUSE	The parameter to ADDRESS, BADDRESS, or WADDRESS does not reside on a storage unit boundary, so a legal address can not be generated for it.
	ACTION	Do not take the address of this variable.

858	MESSAGE	THIS FEATURE IS NO LONGER VALID (858)
CT	CAUSE	The designated feature has been removed from the language

definition.

ACTION Remove the feature from the source code.

---

859 MESSAGE ANYPTR MAY NOT BE DEREFERENCED (859)

CT CAUSE Pointers of type ANYPTR may not be dereferenced.

ACTION Assign or type coerce the pointer before dereferencing it.

---

860 MESSAGE ADDRESS ALIGNMENT INCOMPATIBLE WITH DESIRED USE (860)

CT CAUSE The alignment of the value of the pointer being coerced is incompatible with the alignment implied by the type coercion.

ACTION Ensure that the target type's alignment is smaller than or equal to that of the source type.

CAUSE The alignment of an actual parameter prohibits its use due to the required alignment of the VAR or ANYVAR formal parameter.

ACTION Ensure that the actual parameter has an alignment larger than or equal to that of the formal parameter.

---

861 MESSAGE INCOMPATIBLE SOURCE AND TARGET TYPES FOR COERCION (861)

CT CAUSE The subrange of values for the type of the parameter to the type coercion does not overlap with the subrange of values for the target type of the type coercion. (ordinal coercion only)

ACTION None: A subrange variable cannot be coerced to another subrange type that does not have some overlap with its original type.

---

862 MESSAGE THIS TYPE COERCION NOT PERMITTED AS REFERENCE PARAMETER (862)

CT CAUSE Ordinal type coercions that require type conversion are not permitted as reference parameters.

Pointer type coercions that require type conversion such as short-to-long or long-to-short pointer conversion are not permitted as reference parameters.

ACTION Copy into a variable, and pass that as the reference parameter.

---

863 MESSAGE THIS FEATURE IS NOT IMPLEMENTED (863)

CT CAUSE The feature in use has not been implemented in the current compiler.

ACTION Remove this feature from the source code.

---

864 MESSAGE BYTE OFFSET NOT PERMITTED WITH PROCEDURE OR FUNCTION VAR (864)

CT CAUSE ADDR takes a second parameter only if the first parameter is not a procedure or function variable.

ACTION Remove the second parameter.

---

866 MESSAGE NO ANYVAR FOUND IN FORMAL PARAMETER LIST (866)

CT CAUSE A procedure or function declared with OPTION UNCHECKABLE ANYVAR must have an ANYVAR parameter in its formal parameter list.

ACTION Remove the option or supply an ANYVAR.

---

868 MESSAGE INTRINSIC MECHANISM ERROR "! ". (868)

CT CAUSE An error has occurred in accessing the intrinsic file.

ACTION [REV BEG]Check the status indicator returned from the Intrinsic Mechanism Access Routines. If the status indicator is one of the following values, correct the error.

Value		Description
1	OpenFail	The given IM could not be opened.
2	CloseFail	The IM could not be closed.
3	RetrieveFail	An access error occurred in attempting to read from the IM.
4	ReplaceFail	An access error occurred in attempting to write from the IM.
5	SpaceExhausted	Inadequate space remains in the IM to perform requested action.
14	BadIntrinsicFile	The file being accessed is not an intrinsic file.

If the status indicator is *not* one of the above values, report the error to your HP Service Representative.[REV END]

---

869 MESSAGE ARRAY ELEMENT SIZE MUST BE >= ONE BYTE. (869)

CT CAUSE Array parameter to Move_Fast must have elements with sizes greater than or equal to one byte.

ACTION Use another mechanism to perform the move.

---

870 MESSAGE ARRAY MUST BE ALIGNED ON A BYTE BOUNDARY. (870)

CT CAUSE Array parameter to Move_Fast must be aligned on a byte boundary.

ACTION Use another mechanism to move the array.

---

871 MESSAGE INVALID ARRAY PARAMETERS TO MOVE_FAST. (871)

CT CAUSE Both array parameters to Move_Fast must have elements with the same sizes.

ACTION Use some other mechanism to move the array.

	CAUSE	If only one of the parameters is crunched, then the elements must be packed in with no wasted space between elements.
	ACTION	Check the packing.

872	MESSAGE	ARRAY ELEMENTS CANNOT BE CONFORMANT ARRAYS. (872)
CT	CAUSE	If an array parameter to one of the MOVE routines is a conformant array, then its elements must not themselves be conformant arrays. The size of the elements must be known at compile time.
	ACTION	Use a different mechanism like a FOR or WHILE loop to move the elements.

873	MESSAGE	INVALID MODULE LIBRARY SPECIFIED (873)
CT	CAUSE	Either the file that is to be used for the search of a module or the file that is the Module Library is not of the Module Library format.
	ACTION	Ensure that the file that was previously created is in Module Library format.

874	MESSAGE	INVALID IMPORT MODULE ENVIRONMENT (874)
CT	CAUSE	Trying to import a module which was compiled under a different compilation environment.
	ACTION	Recompile imported module on current machine.

875	MESSAGE	INTRINSIC DECLARATION NOT ENTERED INTO INTRINSIC FILE (875)
CT	CAUSE	Due to a previous error the intrinsic declaration was not entered into the intrinsic file.
	ACTION	Correct previous errors.

876	MESSAGE	INTRINSIC FILE OVERFLOW (876)
CT	CAUSE	The physical limit of the intrinsic file has been exceeded.
	ACTION	Build a larger intrinsic file using BUILD or a file equation.

877	MESSAGE	INVALID DEREFERENCING OF AN IMPORTED POINTER (877)
CT	CAUSE	Trying to dereference an imported pointer whose type is not defined.
	ACTION	Import the type that the pointer points to. Do not dereference the pointer in this module.

878	MESSAGE	INVALID USE OF AN INLINED ROUTINE (878)

CT CAUSE The address of an inlined routine is being requested. This happens in the following cases:

* The procedure is passed as a parameter to WAddress, BAddress, Addr or Assert (as the "assert procedure").

* The procedure is passed as the actual parm when the formal parm is a procedural/functional type.

ACTION Don't use option inline if the procedure is being used in the above contexts.

---

879 MESSAGE UNIMPLEMENTED USE OF AN INLINED ROUTINE "! " (879)

CT CAUSE An inline function appearing as an actual parameter to itself is an unimplemented feature.

ACTION Assign the function result to a local variable and pass the local variable as the parameter.

---

880 MESSAGE $ALIGNMENT$ CONFLICT (880)

CT CAUSE The $ALIGNMENT$ value on a record or array declaration is less than the minimum alignment for the record or array (because of the alignments of its fields/elements).

ACTION Specify an alignment for the record or array that is at least as large as the maximum alignment of any of its fields/elements.

CAUSE The type on the right hand side of a type declaration is a type identifier which has already been defined with $ALIGNMENT$.

ACTION Remove the conflicting $ALIGNMENT$.

CAUSE $ALIGNMENT$ is not allowed on string and file types.

ACTION Don't use $ALIGNMENT$ on string and file types.

---

881 MESSAGE MIXED MODE OPERATIONS NOT ALLOWED (881)

CT CAUSE An expression which mixes $HP3000_16$ and $HP3000_32$ operands is not allowed.

ACTION Don't mix modes in the expression.

CAUSE String parameters to predefined string procedures and functions and strings used in string expressions require $HP3000_16$.

ACTION Don't use $HP3000_32$ strings as parameters to string predefines or in string expressions.

CAUSE Real parameters to arithmetic functions require $HP3000_16$ reals.

ACTION Don't use $HP3000_32$ reals as parameters to arithmetic predefines.

---

882 MESSAGE MIXED MODE PACKING NOT ALLOWED (882)

CT CAUSE Mixing $HP3000_16$ and $HP3000_32$ in data type definitions is

not allowed.

ACTION Don't mix modes in data declarations.

---

883 MESSAGE COERCION REQUIRES $TYPE_COERCION 'CONVERSION'$ (883)

CT CAUSE The current $TYPE_COERCION$ level is insufficient to permit this coercion.

ACTION Set the $TYPE_COERCION$ level to that given in the message.

---

884 MESSAGE INVALID TYPE FOR INTRINSIC FORMAL PARAMETER NUMBER ! (884)

CT CAUSE The data type for the formal parameter specified is not an acceptable type for an intrinsic declaration (when building an intrinsic file using $BUILDINT$).

ACTION Use an appropriate language-independent type for the intrinsic parameter.

---

885 MESSAGE INVALID TYPE FOR INTRINSIC FUNCTION RETURN (885)

CT CAUSE The data type for the function return specified is not an acceptable type for an intrinsic declaration.

ACTION Specify the correct type.

---

886 MESSAGE RECURSIVE INCLUDE OF FILE (886)

CT CAUSE The file just specified in an $INCLUDE$ directive is currently being included (thus, this is an infinite recursion of includes; a fatal error).

ACTION Remove the recursive include.

---

887 MESSAGE INVALID FORMAL PARAMETER TYPE (887)

CT CAUSE A data type which is a $HP3000_32$ type is not allowed as a formal parameter when $HP3000_16$ is ON.

ACTION Declare the parameter to be of a $HP3000_16$ type.

---

888 MESSAGE STATEMENT ! INCOMPATIBLE WITH $ASSUME '!' (888)

CT CAUSE The code generated for the given statement conflicts with the given assume option. The compiler has detected incorrect code generation.

ACTION Use a correct ASSUME option or remove the $ASSUME option.

---

889 MESSAGE CONFORMANT ARRAYS NOT ALLOWED WITH $HP3000_16$ (889)

CT CAUSE Conformant arrays are not implemented when using $HP3000_16$.

ACTION Do not use this feature with $HP3000_16$.

---

890 MESSAGE CANNOT EXPORT AN IMPORTED MODULE IN THE OUTER BLOCK (890)

CT CAUSE The word EXPORT was seen after the module name on an import statement in the outer block.

ACTION Remove the word EXPORT.

---

891 MESSAGE LISTINTR FAILED TO COMPLETE SUCCESSFULLY (891)

CT CAUSE The listing of the intrinsic file terminated unexpectedly. Possible reasons are that the listing file could not be opened, or the file limit on the listing file was exceeded.

ACTION Make sure the intrinsic file is present and spelled correctly. If the file limit on the listing file was exceeded, build a larger file or use a file equation to specify a larger file.

---

892 MESSAGE UNABLE TO CLOSE FILE '!' (892)

CT CAUSE The compiler was unable to close the specified file. Possible reasons are that system file space is exhausted, or that an attempt is made to create a file across account boundaries (which is not allowed on MPE/iX).

ACTION Create enough system file space or specify a file within the account boundary.

---

893 MESSAGE I/O MODULE(S) NOT IMPORTED (893)

CT CAUSE A call to a standard procedure such as writeln, readln, write, or read was made in the implement section of a module that did not import the appropriate module STDINPUT or STDOUTPUT. As a result, the default file symbols input and/or output are unknown to the compilation unit.

ACTION Explicitly IMPORT the appropriate system-defined module STDINPUT, STDOUTPUT, or both.

---

894 MESSAGE INVALID USE OF MODULE IDENTIFIER (894)

CT CAUSE Module identifier can only be used with IMPORT.

ACTION Rename the identifier or remove the module identifier.

---

900 MESSAGE INCORRECT POINTER ALIGNMENT (900)

RT CAUSE Internal parameter to CHKA.

ACTION No action is required. Internal use only.

---

905 MESSAGE INVALID PROCEDURAL/FUNCTIONAL VALUE REFERENCED (905)

RT CAUSE The value does not denote any actual procedure or function.

		The static nesting level of the value does not correspond to the current state of the activation stack. The value is NIL. The procedure or function is uninitialized or contains a bad value.
	ACTION	Make sure the procedure or function has been initialized correctly.
908	MESSAGE	MOVE PROCEDURE PARAMETERS OUT OF RANGE (908)
RT	CAUSE	The range of the move for either the source or target exceeds the declared range of the source or target arrays.
	ACTION	Check that the expressions defining the start, offset, and count are producing correct values.
909	MESSAGE	ESCAPE PROCEDURE WITH NO ENCLOSING TRY-RECOVER (909)
RT	CAUSE	Escape was called by the user and no enclosing TRY-RECOVER was declared.
	ACTION	Use TRY-RECOVER to catch the escape.
910	MESSAGE	ESCAPE EXECUTED WITHOUT AN UNWIND DESCRIPTION FOR THE FRAME (910)
RT	CAUSE	An escape was executed, but one or more of the procedures in the program stack does not have an unwind descriptor.
	ACTION	Contact Hewlett-Packard.
911	MESSAGE	ESCAPE EXECUTED BUT CANNOT UNWIND DESCRIPTOR FOR THE FRAME (911)
RT	CAUSE	An Escape was executed, but one or more of the procedures in the program stack has a frame that is not unwindable.
	ACTION	Contact Hewlett-Packard.
912	MESSAGE	GOTO EXECUTED AND BOTTOM OF FRAME HIT; INTERNAL ERROR (912)
RT	CAUSE	Internal error occurred while executing a non-local GOTO statement.
	ACTION	Contact Hewlett-Packard.
913	MESSAGE	GOTO EXECUTED WITHOUT AN UNWIND DESCRIPTOR FOR THE FRAME (913)
RT	CAUSE	A non-local GOTO was executed, but one or more of the procedures in the program stack does not have an unwind descriptor.
	ACTION	Contact Hewlett-Packard.

914	MESSAGE	GOTO EXECUTED BUT CANNOT UNWIND DESCRIPTOR FOR THE FRAME (914)
RT	CAUSE	A non-local GOTO was executed, but one or more of the procedures in the program stack has a frame that is not unwindable.
	ACTION	Contact Hewlett-Packard.
5001	MESSAGE	GOTO OUT OF BLOCK TO MULTIPLE ENTRY PT. (5001)
W	CAUSE	Goto out of block to procedure with multiple entry points.
	ACTION	Warning only. No action required.
5002	MESSAGE	! (5002)
W	CAUSE	FSerr for other messages (see following messages).
	ACTION	Warning only.
5004	MESSAGE	UNINITIALIZED VARIABLE (SYMID = !) !. (5004)
W	CAUSE	Optimizer detected uninitialized variable, should have been initialized before its use.
	ACTION	Warning only.
5080	MESSAGE	PREVIOUS VERSION OF ENTRY ! WAS REPLACED (5080)
W	CAUSE	Code for the entry listed was replaced in the RL (iX only).
	ACTION	Warning only. No action required.
5104 to 5199	MESSAGE	INTERNAL COMPILER ERROR.
W	CAUSE	The compiler is in error.
	ACTION	Report error to your HP Service Representative.
5200	MESSAGE	INTERNAL REGISTER TABLE OVERFLOW; PROCEDURE TOO BIG (5200)
CT	CAUSE	Your procedure is too large for the compiler to handle at once.
	ACTION	Break your procedure into two or more pieces.
5202	MESSAGE	MAXIMUM AMOUNT OF LOCAL DATA ALLOWED EXCEEDED (5202)
CT	CAUSE	The maximum amount of local storage allowed has been exceeded.
	ACTION	Break your procedure into two or more pieces.

5207 MESSAGE MULTIPLE PROGRAM ENTRY POINTS (5207)

CT CAUSE Possible multiple main programs.

ACTION Make sure only one main program exists in the compilation unit.

5208 MESSAGE TOO MANY NESTED TRYS IN PROCEDURE (5208)

CT CAUSE The maximum number of nested TRYs allowed in a procedure is about thirty.

ACTION Break up your procedure by putting some of the inner TRY blocks into a nested procedure.

5209 MESSAGE CANNOT OPEN OBJECT FILE (5209)

CT CAUSE The compiler cannot open the object file. This may be because:

(a) You do not have write permission in the group (on MPE/iX) or directory (on HP-UX) that you are working in.

(b) You have exceeded some physical disk space limit.

ACTION

(a) Work in a group or directory in which you have write permission or get write permission in the current group or directory.

(b) Remove some unnecessary files to make room for your object file.

5210 MESSAGE CANNOT CLOSE OBJECT FILE (5210)

CT CAUSE The compiler could not close the object file. This may be because:

(a) You do not have write permission in the group (on MPE/iX) or directory (on HP-UX) that you are working in.

(b) You have exceeded some physical disk space limit.

ACTION

(a) Work in a group or directory in which you have write permission or get write permission in the current group or directory.

(b) Remove some unnecessary files to make room for your object file.

5211 MESSAGE INVALID FILE CODE FOR OBJECT FILE ! (5211)

CT CAUSE File code for object file is not NMOBJ or NMRL.

ACTION Change file code or use different object file.

5212 MESSAGE DUPLICATE LABELS ARE NOT ALLOWED (5212)

CT CAUSE A duplicate user or internal label exists.

	ACTION	Check for duplicate labels. If none are found, report this error to your HP Service Representative.

5213	MESSAGE	CANNOT OPEN ASSEMBLY FILE (5213)
CT	CAUSE	The compiler could not open the assembly file. This may be because: (a) You do not have write permission in the group (on MPE/iX) or directory (on HP-UX) that you are working in. (b) You have exceeded some physical disk space limit.
	ACTION	(a) Work in a group or directory in which you have write permission, or obtain write permission in the current group or directory. (b) Remove some unnecessary files to make room for your assembly file.

5214	MESSAGE	CANNOT CLOSE ASSEMBLY FILE (5214)
CT	CAUSE	The compiler could not close the assembly file. This may be because: (a) You do not have write permission in the group (on MPE/iX) or directory (on HP-UX) that you are working in. (b) You have exceeded some physical disk space limit.
	ACTION	(a) Work in a group or directory in which you have write permission or obtain write permission in the current group or directory. (b) Remove some unnecessary files to make room for your assembly file.

5380	MESSAGE	ATTEMPT TO OPEN FILE ! FAILED (5380)
CT	CAUSE	File could not be opened by compiler.
	ACTION	Check capabilities, access rights, and permissions of file in the group (on MPE/iX) or directory (on HP-UX).

5381	MESSAGE	FILE ! HAS INVALID FILE CODE; EXPECTED NMRL (5381)
CT	CAUSE	File code of object file should be NMRL.
	ACTION	Use different file for object, build file as NMRL, or do not use RL compile option.

5382	MESSAGE	ATTEMPT TO ADD MODULE(S) BEYOND MODULE LIMIT OF ! IN FILE ! (5382)
CT	CAUSE	Module cannot be added to named RL.

	ACTION	Clean up your RL or use a different file for the object.
5383	MESSAGE	FILE ! HAS AN INVALID RECORD SIZE. EXPECTED 128W RECORDS. (5383)
CT	CAUSE	The RL has an invalid record size.
	ACTION	Build a new RL file with a correct record size.
5400 to 5999	MESSAGE	INTERNAL COMPILER ERROR.
I	CAUSE	The compiler is in error.
	ACTION	Report error to your HP Service Representative.
6055	MESSAGE	OPTDRIVER: BAD OPTIMIZER OPTION; IGNORED. (6055)
W	CAUSE	Internal compiler error.
	ACTION	Report error to your HP Service Representative.
6056	MESSAGE	OPTDRIVER: CAN'T OPEN DEBUG FILE FOR OUTPUT; STDOUT USED. (6056)
W	CAUSE	Internal compiler error.
	ACTION	Report error to your HP Service Representative.
6057	MESSAGE	OPTDRIVER: BAD OPTIMIZATION LEVEL SPECIFIED; DEFAULT OF 0 USED. (6057)
W	CAUSE	Internal compiler error.
	ACTION	Report error to your HP Service Representative.
6058	MESSAGE	OPTDRIVER: BAD SCHEDULER ALGORITHM SPECIFIED, USED DEFAULT. (6058)
W	CAUSE	Internal compiler error.
	ACTION	Check argument to +DS, then report error to your HP Service Representative.
6059	MESSAGE	OPTDRIVER: !1 BASIC BLOCKS; DROPPING TO LEVEL 1 OPTIMIZATION FOR !2. (6059)
W	CAUSE	Procedure !2 contains more that 500 basic blocks and requires an inordinate amount of compile-time resources. Therefore, the optimizer will be run at level 1 for !2.
	ACTION	No action is necessary. However, on HP-UX, if level 2 optimization is desired in spite of a possibly lengthy compile time, this limit can be overridden by the use of the +Obb*num* option, where *num* is at least as large as the number given in

		this message.
	ACTION	Use the $OPTIMIZE 'BASIC_BLOCKS *num* $ compiler option.
6110 to 6199	MESSAGE	INTERNAL OPTIMIZER ERROR.
CT	CAUSE	The compiler is in error.
	ACTION	Report error to your HP Service Representative.
6200 to 6299	MESSAGE	ALIASER: OUT OF MEMORY.
CT	CAUSE	The optimizer ran out of virtual memory.
	ACTION	The easiest workaround is to break your compilation unit into two or more pieces. On HP-UX, this error may also be produced if the system runs out of swap space, so another possible work-around is to increase the amount of swap space available to the system (see your HP-UX system administrator about this). However, this action should be taken only as a last-resort. On MPE/iX, the compiler heap space can be increased by running PASCALXL.PUB.SYS with a larger NMHEAP: :RUN PASCALXL.PUB.SYS;NMHEAP=120000000 ...
6305	MESSAGE	RALLOC: OUT OF GENERAL REGISTERS. (6305)
CT	CAUSE	Possible overly complex expression.
	ACTION	Simplify large or complex expression.
6306	MESSAGE	RALLOC: OUT OF CALLEE SPACE REGISTERS. (6306)
CT	CAUSE	Long pointer expression too complex.
	ACTION	Simplify long pointer expressions.
6307	MESSAGE	RALLOC: OUT OF CALLER SPACE REGISTERS. (6307)
CT	CAUSE	Long pointer expression too complex.
	ACTION	Simplify long pointer expressions.
6308	MESSAGE	RALLOC: OUT OF CALLEE FLOATING POINT REGISTERS. (6308)
CT	CAUSE	Floating point expression too complex.
	ACTION	Simplify floating point expressions.

6309	MESSAGE	RALLOC: OUT OF CALLER FLOATING POINT REGISTERS. (6309)
CT	CAUSE	Floating point expression too complex.
	ACTION	Simplify floating point expressions.

6310 to 6365	MESSAGE	OUT OF MEMORY
CT	CAUSE	The optimizer ran out of virtual memory.
	ACTION	The easiest workaround is to break your compilation unit into two or more pieces. On HP-UX, this error may also be produced if the system runs out of swap space, so another possible work-around is to increase the amount of swap space available to the system (see your HP-UX system administrator about this). However, this action should be taken only as a last-resort. On MPE/iX, the compiler heap space can be increased by running PASCALXL.PUB.SYS with a larger NMHEAP: :RUN PASCALXL.PUB.SYS;NMHEAP=120000000 ...

6400 to 6999	MESSAGE	INTERNAL COMPILER ERROR.
I	CAUSE	The compiler is in error.
	ACTION	Report error to your HP Service Representative.

ON HP-UX, the following warnings are generated if you pass a model number that is not found in the /usr/lib/sched.models file.

7000	MESSAGE	MODEL NUMBER IS UNKNOWN; WILL DEFAULT TO *arch-rev* CODE GENERATION. (7000)
W	CAUSE	The model number given on a +DA option is not known to the compiler.
	ACTION	The default code generation is as specified in the warning. If this is not the desired target architecture revision, the version may be specified using an architecture revision (such as 1.1) instead of a model number on the +DA option.

7001	MESSAGE	MODEL NUMBER IS UNKNOWN; DEFAULT INSTRUCTION SCHEDULING IS USED. (7001)
W	CAUSE	The model number given on a +DS option is not known to the compiler.
	ACTION	The default instruction scheduling is based on the most recent processor implementation known to the compiler. If this is not what is desired, an alternate model number may be specified.

On HP-UX, the following warning will be generated if the file /usr/lib/sched.models cannot be found.

7002	MESSAGE	CANNOT OPEN /usr/lib/sched.models. (7002)
W	CAUSE	The file /usr/lib/sched.models does not exist or cannot be opened for reading.
	ACTION	Check protections on /usr/lib/sched.models. If it does not exist, contact your HP Service Representative.

On HP-UX, the following warning is generated if you pass arguments that do not conform to the expected format.

7003	MESSAGE	IMPROPER ARGUMENT TO +DA OR +DS OPTION. (7003)
W	CAUSE	An improper argument was given to the +DA or +DS option.
	ACTION	Check the reference manual for information on the correct form of the option.

7100 to 7109	MESSAGE	INTERNAL COMPILER ERROR.
I	CAUSE	The compiler is in error.
	ACTION	Report error to your HP Service Representative.

7110	MESSAGE	DEBUG INFORMATION MAY BE CORRUPT; "! " UNRESOLVABLE REFERENCE(S). (7110)[REV BEG]
I	CAUSE	User errors.
	ACTION	Correct all user errors or remove -g or $SYMDEBUG options and recompile. If there are no user errors, report error to your HP Service Representative.[REV END]

7200	MESSAGE	INTERNAL TABLE OVERFLOW (7200)
CT	CAUSE	Source file too large.
	ACTION	Split program up into smaller files.

7201	MESSAGE	NEW_SLC_BLOCK: OUT OF MEMORY. (7201)
CT	CAUSE	Compiler ran out of virtual memory.
	ACTION	See message 6200.

7202	MESSAGE	INIT_LINK: OUT OF MEMORY. (7202)

CT	CAUSE	Compiler ran out of virtual memory.
	ACTION	See message 6200.

7203	MESSAGE	ALLOCATE_BYTES: OUT OF MEMORY. (7203)
CT	CAUSE	Compiler ran out of virtual memory.
	ACTION	See message 6200.

7204	MESSAGE	ERROR IN WRITING TO OUTPUT FILE. (7204)
CT	CAUSE	I/O error writing to object file.
	ACTION	Check for full file system (HP-UX, MPE/iX) or an object file that too small (MPE/iX).

7205	MESSAGE	UNABLE TO ALLOCATE SPACE FOR OBJECT IN RL. (7205)
CT	CAUSE	I/O error writing to RL.
	ACTION	Check for an RL file that is too small (MPE/iX).

7206	MESSAGE	UNABLE TO ADD OBJECT TO RL. (7206)
CT	CAUSE	I/O error writing to RL.
	ACTION	Check for an RL file that is too small, write permission (HP-UX), or capability (MPE/iX).

7207	MESSAGE	OBJECT IS TOO BIG TO FIT INTO RL. (7207)
CT	CAUSE	Object size is too large for the RL requested.
	ACTION	Check for an RL file that is too small or split up object (MPE/iX).

7400 to 7999	MESSAGE	INTERNAL COMPILER ERROR.
I	CAUSE	The compiler is in error.
	ACTION	Report error to your HP Service Representative.

A-: 80

Appendix B ASCII Character Codes

Table B-1 maps each ASCII character to its decimal and hexadecimal code, its ASCII symbol, and its name. Each code is stored in eight bits; thus the decimal codes are between 0 and 255, and the hexadecimal codes are between 0 and FF.

Table B-1. ASCII Character Codes

Decimal Code	Hexadecimal Code	ASCII Symbol	Name
0	00	NUL	Null
1	01	SOH	Start of heading
2	02	STX	Start of text
3	03	EXT	End of text
4	04	EOT	End of transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal tab
10	0A	LF	Line feed
11	0B	VT	Vertical tab
12	0C	FF	Form feed
13	0D	CR	Carriage return
14	0E	SO	Shift out

Table B-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	ASCII Symbol	Name
15	0F	SI	Shift in
16	10	DLE	Data link escape
17	11	DC1	Device control 1
18	12	DC2	Device control 2
19	13	DC3	Device control 3
20	14	DC4	Device control 4
21	15	NAK	Negative acknowledgement
22	16	SYN	Synchronous idle
23	17	ETB	End of transmission block
24	18	CAN	Cancel
25	19	EM	End of medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File separator
29	1D	GS	Group separator
30	1E	RS	Record separator
31	1F	US	Unit separator

32	20	SP	Space
33	21	!	Exclamation mark
34	22	"	Quotation mark
35	23	#	Number sign

Table B-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	ASCII Symbol	Name
36	24	$	Dollar sign
37	25	%	Percent sign
38	26	&	Ampersand
39	27	'	Apostrophe
40	28	(	Left parenthesis
41	29	)	Right parenthesis
42	2A	*	Asterisk
43	2B	+	Plus sign
44	2C	,	Comma
45	2D	-	Minus sign
46	2E	.	Full stop
47	2F	/	Solidus

48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight

Table B-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	ASCII Symbol	Name
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less-than sign
61	3D	=	Equal sign
62	3E	>	Greater-than sign
63	3F	?	Question mark

64	40	@	Commercial "at" sign
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K
76	4C	L	Uppercase L
77	4D	M	Uppercase M

Table B-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	ASCII Symbol	Name
78	4E	N	Uppercase N
79	4F	O	Uppercase O

80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Left bracket
92	5C	\	Reverse solidus
93	5D	]	Right bracket
94	5E	^	Circumflex accent
95	5F	_	Underline
96	60	‘	Grave accent
97	61	a	Lowercase a
98	62	b	Lowercase b

Table B-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	ASCII Symbol	Name
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	68	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s

116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w

Table B-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	ASCII Symbol	Name
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Left brace
124	7C	|	Vertical line
125	7D	}	Right brace
126	7E	~	Tilde
127	7F	DEL	Delete

Appendix C Compiler Limits and Values

These compiler limits are maximum values that you cannot change:

Number of:	**Maximum Value**
Bits per structure	2147483600
Characters per identifier	132
Characters per source line	132
Characters per string	268435447
Elements per array	268435455
Elements per enumerated type	17367
Elements per set	2147483616
Nested IF options *	12
Nested INCLUDE options *	Operating system dependent
Nested PUSH options *	15
Nested TRY-RECOVER constructs	30

* If a program contains one INCLUDE option, the number of nested INCLUDE options is one. If the included file contains an INCLUDE option, the number of nested INCLUDE options is two, and so on. The definitions of the number of nested IF options and the number of nested PUSH options are analogous.

The following values are implementation defined:

minint
maxint
e
pi