HP Pascal/HP-UX Programmer's Guide

HEWLETT
PACKARD

HP Part No. 92431-90006
Printed in U.S.A. August 1992

Sixth Edition
E0892

Printing History

New editions are complete revisions of the manual. Update packages may be issued between editions.

The software code printed alongside the date indicates the version level of the software product at the time the manual 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: 31502A.00.03
Second Edition	November 1987	MPE XL: 31502A.01.06 HP-UX: 92431A.01.09
Third Edition	January 1988	MPE XL: 31502A.01.06 HP-UX: 92431A.01.12
Fourth Edition	October 1988	MPE XL: 31502A.01.19 HP-UX: 92431A.03.04
Fifth Edition	January 1991	MPE XL: 31502A.03.10 HP-UX: 92431A.08.00
Sixth Edition	June 1992 August 1992	MPE/iX: 31502A.04.05 HP-UX: 92431A.09.07

Preface

This *HP Pascal/HP-UX Programmer's Guide* for the Hewlett-Packard HP Pascal/iX and HP Pascal/HP-UX programming languages is intended for programmers with at least six months of Pascal programming experience, but no HP Pascal/iX or HP Pascal/HP-UX programming experience. It discusses selected HP Pascal/iX and HP Pascal/HP-UX topics in detail, and explains statement interaction where necessary. It does not explain every feature of HP Pascal/iX or HP Pascal/HP-UX, as the *HP Pascal/HP-UX Reference Manual* does.

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, the operating system for HP 3000s. MPE V is not based on the PA-RISC architecture; however, MPE V software can be run on the PA-RISC (Series 900) HP 3000s in what is known as *compatibility mode*.

Throughout this manual, the term *HP Pascal* refers to both HP Pascal/iX and HP Pascal/HP-UX. The following is a short description of each chapter and appendix.

Chapter 1 Describes HP Pascal/iX and HP Pascal/HP-UX and explains their relationship to HP Standard Pascal and its subsets.

Chapter 2 Describes HP Pascal program structure in terms of syntax and compilation units, and explains how your program can interface with its external environment.

Chapter 3 Explains how program input/output works.

Chapter 4 Gives the ranges of the predefined data types of HP Pascal and explains the types which HP Pascal does not share with older Pascal implementations.

Chapter 5 Explains how HP Pascal allocates space for and aligns static data structures.

Chapter 6 Explains dynamically allocated HP Pascal data structures.

Chapter 7 Discusses HP Pascal parameters.

Chapter 8 Explains procedure options, which allow routines to have optional parameters and default parameter values.

Chapter 9 Explains how your program can use external routines.

Chapter 10	Explains how your program can use intrinsics.
Chapter 11	Explains how to write error recovery code that allows your program to handle its own run-time errors. Explains how to debug your program.
Chapter 12	Explains how to use the optimizer to improve your program.
Appendix A	Explains how HP Pascal/iX works on the MPE/iX operating system.
Appendix B	Explains how HP Pascal/HP-UX works on the HP-UX operating system.

Refer to the following manuals for further information on HP Pascal:

- *HP Pascal/HP-UX Reference Manual* (92431-90005)
- *HP Pascal/HP-UX Migration Guide* (92431-90004)

This manual also refers to the following non-HP Pascal manuals:

- *ALLBASE/SQL Pascal Application Programming Guide* (36216-90007)
- *HP C Programmer's Guide* (92434-90002)
- *HP Link Editor/XL Reference Manual* (32650-90030)
- *HP System Dictionary/XL General Reference Manual* (32256-90004)
- *HP TOOLSET/XL Reference Manual* (36044-90001)
- *HP-UX Floating-Point Guide* (B2355-90024)
- *HP-UX Reference* (B2355-90004)
- *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)
- *Programming on HP-UX* (B2355-90026)
- *Switch Programming Guide* (32650-90014)
- *Trap Handling Programmer's Guide* (32650-90026)
- *TurboIMAGE/XL Reference Manual* (30391-90001)
- *Using VPLUS/V: Introduction to Forms Designs* (32209-90004)

If you have suggestions for improving this manual, please send us the Reader Comment Card, located at the front of this manual.

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`: $\texttt{COMMAND} \left\{ \begin{array}{l} \texttt{ON} \\ \texttt{OFF} \end{array} \right\}$
`[ ]`	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. $\texttt{COMMAND}\ \mathit{FileName} \left[\begin{array}{l} \texttt{OPTION} \\ \mathit{Parameter} \end{array} \right]$

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.

    $\left\{\begin{array}{c}\texttt{A}\\ \texttt{B}\end{array}\right\}$ | ... |

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

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

    (*Parameter*)△(*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.

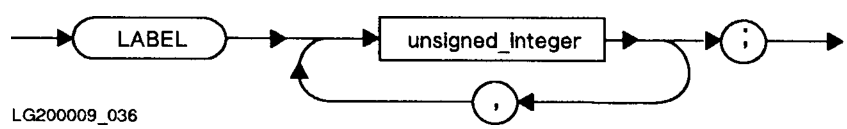
`For Syntax Diagrams:`

- 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.

Contents

Figures

Tables

1

Introduction

HP Pascal/iX and HP Pascal/HP-UX are supersets of HP Standard Pascal, the Pascal language that runs on all HP computers. HP Pascal/iX runs on the MPE/iX operating system and HP Pascal/HP-UX runs on the HP-UX operating system. Both operating systems run on HP PA-RISC computers, and both achieve ISO and ANSI validation. HP Pascal takes advantage of the architecture of these computers and has system programming extensions to HP Standard Pascal.

As a superset of HP Standard Pascal, HP Pascal accepts the syntax of the HP Standard Pascal subsets ISO Pascal and ANSI Standard Pascal. You can instruct the HP Pascal compiler to accept only the syntax of an HP Pascal subset. Refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual* for information on the STANDARD_LEVEL compiler option.

Figure 1-1 shows the relationship between HP Pascal, HP Standard Pascal, ISO Pascal, and ANSI Standard Pascal.

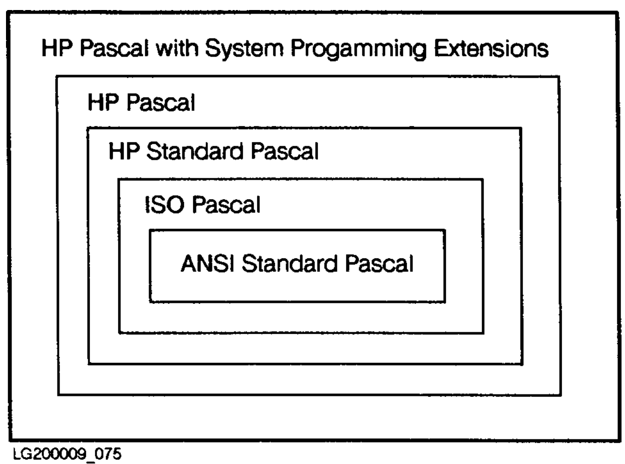


Figure 1-1. Relationship Between HP Pascal and Other Pascals

HP Pascal can interface with any subsystem that can be accessed through intrinsics. Some of the HP subsystems HP Pascal can interface with are listed below:

Subsystem	Description of Subsystem	Reference
TurboIMAGE/XL	Network database management system. Your HP Pascal program accesses TurboIMAGE/XL routines with intrinsic calls.	*TurboIMAGE/XL Reference Manual*
SQL	Relational database management system whose Pascal preprocessor has macros that generate calls to SQL.	*ALLBASE/SQL Pascal Application Programming Guide*
HP System Dictionary/XL	Dictionary of MPE/iX data elements.	*HP System Dictionary/XL General Reference Manual*
VPLUS	Forms generator. Your HP Pascal program accesses VPLUS routines with intrinsic calls.	*Using VPLUS/V: Introduction to Form Designs*

HP Pascal can interface with several system debuggers. Some of the debuggers are listed below:

Subsystem	Description of Subsystem	Reference
HP Symbolic Debugger	A symbolic debugger available on both the MPE/iX and HP-UX operating systems. It supports HP Pascal features.	*MPE/iX Symbolic Debugger User's Guide*
DEBUG	MPE/iX System Debugger.	*MPE/iX System Debug Reference Manual*
HP TOOLSET/XL	A programming environment for developing programs. It provides source management, a symbolic debugger, and an editor. The symbolic debugger in HP TOOLSET/XL does not support all the features of HP Pascal.	*HP TOOLSET/XL Reference Manual*

2

Program Structure

This chapter summarizes program structure—in terms of syntax and in terms of compilation units. For complete syntactic definitions of programs and their components, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

Syntactic Structure

Syntactically, every HP Pascal program is composed of two major parts: the program heading and the program block. The program block contains an optional declaration part and a statement (executable) part.

Figure 2-1 illustrates the syntactic structure of an HP Pascal program. For the exact syntax of a program and its components, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

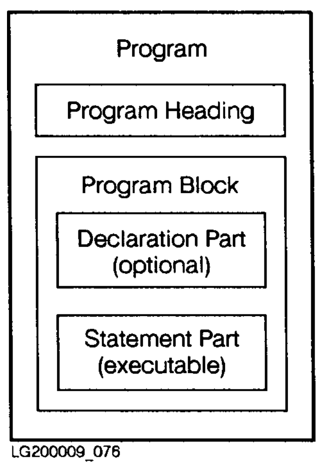


Figure 2-1. Syntactic Structure of a Program

Program Heading

The program heading contains the keyword PROGRAM, the program name, and any program parameters. The program name can be any identifier. If your program uses the standard textfiles *input* and *output* (the default sequential I/O files), these textfiles must be program parameters.

Program parameters—except the standard textfiles *input*, *output*, and *stderr*—must also be declared in the declaration part of the program block.

Example

See the example in the section "Program Block".

For more information about program parameters, see Appendix A and Appendix B.

Program Block

The program block consists of an optional declaration part and a statement (executable) part.

The declaration part defines whatever labels, constants, data types, variables (including program parameters), procedures, functions, or modules you want. It can also redefine standard constants, data types, variables, and routines in the declaration part; however, if you do redefine them, you cannot use their original definitions. You cannot redefine reserved words. For a list of HP Pascal reserved words, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

The statement part is a compound statement (for the definition of *compound statement,* see the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation).

Example

```
PROGRAM prog  (input,   {Required for read}
               output,  {Required for writeln}
               ff);     {Program parameter}


LABEL
   123;

CONST
   c = 35; {Defines constant}
  pi = 3;

TYPE
   t = 1..c;            {Defines data type}
   integer = 1..100;     {Redifines standard data type}

VAR
   vi : t;              {Defines variable}
   ff : FILE of real;   {Redefines program parameter}

FUNCTION f : integer;    {Defines function}
BEGIN {f}                {of type integer}
   f := 30;              {as redefined above}
END; {f}

PROCEDURE p; {Defines procedure}
BEGIN {p}
    .
    .
    .
END; {p}

FUNCTION sqr (n : real) {Redefines}
                : real; {standard function}
BEGIN {sqr}
    .
    .
    .
END; {sqr}


BEGIN {prog}
   123 : read(vi);
       rewrite(ff)
       write(ff,sqr(vi)+pi+f); {sqr and pi are as
                                redefined above}

       p;
       writeln('Done');
END. {prog}
```

Program Heading

Declaration Part

Statement Part

Block

LG200009_108

Compilation Unit Structure

A *compilation unit* is a unit of source code that can be compiled independently of other code (for example, a program is a compilation unit; a block is not).

You can design your program in two ways:

- As a single compilation unit. In this case you must compile the entire program at once.
- As two or more compilation units. In this case you can compile one unit at a time, or you can compile in groups. This is also known as *separate compilation*.

If your program is small, design it as a single compilation unit; it will compile quickly because it is small. (The example program in the section "Program Block" is a single compilation unit.) If your program is large, design it as two or more compilation units. This saves compilation time over the course of program development because you can correct and recompile one unit without recompiling the whole program.

The recommended design for a program with separate compilation units is *modular*; in other words, it is composed of separate compilation units called *modules*. For compatibility with Pascal/V, HP Pascal also supports *global* and *external compilation units.* You can design your program using these separate compilation units, if you prefer. You can mix modules and global and external compilation units.

Modules

A *module* is a compilation unit that defines whatever constants, data types, variables, functions, and procedures you want. A program or another module can *import* the module, thereby gaining access to the definitions that the module *exports*. The definitions that the module does not export are accessible only to the module itself.

Figure 2-2 illustrates the syntactic structure of a module. For the exact syntax of a module and its components, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

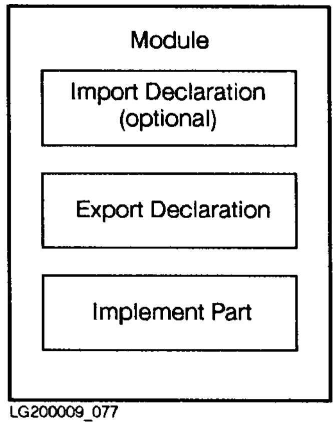


Figure 2-2. Syntactic Structure of a Module

A module's *import declaration* specifies the other modules that it imports. It can access items in the imported modules' export declarations. The import declaration can also be used to specify export of entire modules a second time.

A module's *export declaration* specifies the constants, data types, variables, functions, and procedures that it exports to the modules or programs that import it. A module defines its exportable routines in its implement part.

A module's *implement part* defines constants, data types, variables, and routines. The routines are accessible only to the module itself, unless they are exported in the export declaration.

Example

```
MODULE Module3;
$SEARCH 'mylib.o'$
IMPORT                                          Import Declaration
   Module1,
   Module2;

EXPORT
    CONST
       max = 100;
    TYPE
       ti = 1..max;                             Export Declaration
    VAR
       vi : ti;
    PROCEDURE pi (i : integer);
    FUNCTION fi (i : integer) : integer;

IMPLEMENT
    CONST
       min = 0;
    TYPE
       t2 = min..max;
    VAR
       v2 : ti;
       v3 : t2;
    PROCEDURE pi
    BEGIN  {exported}
       .
       .
       .
    END;
    FUNCTION f1 : integer;
    BEGIN  {exported}                           Implement Part
       .
       .
       .
    END;
    PROCEDURE p2 (i : integer);
    BEGIN  {not exported} -- hidden}
       .
       .
       .
    END;
    FUNCTION f2 (i : integer) : integer;
    BEGIN  {not exported -- hidden}
       .
       :
    END;
END. {Module 3}
```

LG200009_112

Figure 2-3. shows what a module can access.

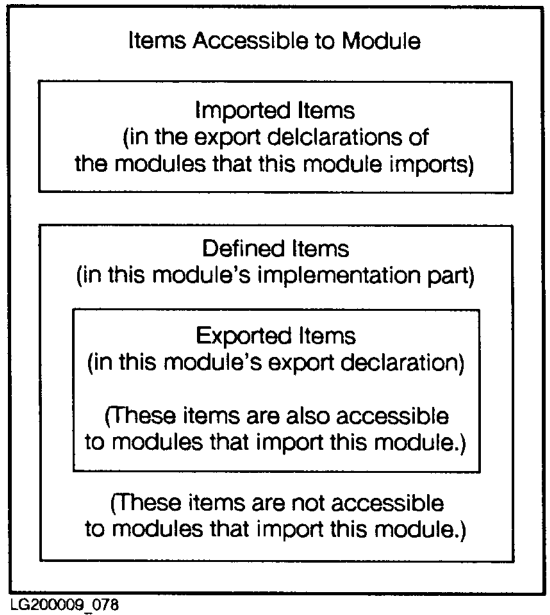


Figure 2-3. What a Module Can Access

A module must be compiled before a program or another module imports it (therefore, two modules cannot import each other).

For the compiler to compile a module with a program, the program must define the module in its declaration part. After defining this module, the program can import it.

When compiling a module independently of a program, the compiler stores the compiled module in the object file or in an alternate file named in the MLIBRARY option (if the MLIBRARY option is specified).

When compiling modules separately or with a program, the placement of the compiler output depends on whether the MLIBRARY option is used. If MLIBRARY is used, the module-text (in the IMPORT and EXPORT declaration) is placed in the file specified with the MLIBRARY option.

If MLIBRARY is not used, the module-text is placed into the object file along with the object code. The module-text present in object files also occurs in RLs (archive libraries), shared libraries, XLs, and program files that were created from these object files unless stripped or the Linkeditor's NODEBUG option is used. Even though the module-text is an unloadable space, it does take up room in the file.

The compiler can extract the module-text from Mlibraries or from any of the binary files discussed above.

Note The compiler may not be able to extract this information if the file is loaded.

The importing program uses the compiler option SEARCH to tell the compiler where to find the module. The compiler options MLIBRARY and SEARCH cannot specify the same library.

For more information on MLIBRARY and SEARCH, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

A program can define a module with the same name as a module in the library that SEARCH specifies. In that case, the program imports the module that it defines, rather than the library module with the same name. If a library contains two modules with the same name, the second one overrides the first. The compiler does not warn you when you are about to override an existing module.

When a program imports a module, the module and its exported items (including the module's exported modules) belong to the global scope of the program. The items that the module does not export (those in its implement part) also exist for the same lifetime as the exported items that were compiled at the simultaneously, even though the program cannot access them.

These non-exported items will not be put in the global symbol table if each module is separately compiled.

Note An exception to this rule occurs if any INLINE routines are exported. In this case all items in the *implement part* are placed in the module-text and the symbol table when imported. This includes any references to intrinsics, even those not used by the INLINE routines. This also means that any `$SYSINTR$` option used by the imported module must also be present in the importing module or program, along with the intrinsic file itself. Because of this, you may want to create multiple smaller modules, one of which will contain the inline routines, but without any intrinsics declared.

Example

Independently compiled modules (to be compiled together in a single compilation unit):

```
MODULE Mod1;  {Mod1 is in Mod1.o}
EXPORT
   .
   :
IMPLEMENT
   .
   :
END; {Mod1}
MODULE Mod2;  {Mod2 is in Mod1.o}
IMPORT
   Mod1;      {Mod2 imports Mod1}
EXPORT
   .
   :
IMPLEMENT
   .
   :
END; {Mod2}
MODULE Mod3;  {This Mod3 is in Mod1.o}
EXPORT
   .
   :
END. {Mod3}
```

Program (to be compiled as a compilation unit that does not contain the above modules -- the program imports the modules from the above compilation unit):

```
PROGRAM prog;
   .
   :
MODULE Mod3;  {The program defines this Mod3}
   .
   :
END; {Mod3}
$SEARCH 'Mod1.o'$
IMPORT
   Mod2,      {Mod2 comes from the library Mod1.o}
   Mod3;      {Mod3 is the one that the program defined}
BEGIN
   .
   :
END.
```

Global, Subprogram, and External Compilation Units

A *global compilation unit* defines global constants, data types, and variables within a Pascal program. It also contains the body of the main program. Syntactically, it is a program that begins with the GLOBAL compiler option. For more information on the GLOBAL compiler option, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

A *subprogram compilation unit* defines subprogram constants, data types, and variables within a Pascal program. Syntactically, it is a program that begins with the SUBPROGRAM compiler option. For more information on the SUBPROGRAM compiler option, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

An *external compilation unit* declares the global variables that it needs and defines routines that the *global compilation* unit and other external compilation units can access using the EXTERNAL directive. Syntactically, it is a program that begins with the EXTERNAL compiler option and has an empty outer block.

Note The EXTERNAL directive and the EXTERNAL compiler option are not the same. For more information, see Chapter 9 in this manual and the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

You must compile global and external compilation units separately. For more information on program preparation see Appendix A and Appendix B.

For more information on the EXTERNAL compiler option, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

Separate Compilation

Separate compilation is the process of separating the source for a large program into pieces that can be compiled independently of other pieces.

There are several reasons why compiling pieces of a program separately is practical:

- When the program is too long to compile.
- When the program is too complex to manage.
- When the program is being worked on by more than one programmer or by a team of programmers.

There are four methods used for separate compilation. They are performed by using modules and by using the compiler options SUBPROGRAM, GLOBAL, and EXTERNAL.

Using modules is the preferred method for separate compilation from a structured programming point of view. However, using modules does have certain limitations, as does using SUBPROGRAM, GLOBAL, and EXTERNAL. You must decide which method works in the way you prefer for your specific situation.

The remainder of this section addresses separate compilation using modules and each compiler option. The uses, advantages, and disadvantages of each method are discussed to help you determine which one to use.

For detailed information on SUBPROGRAM, GLOBAL, and EXTERNAL, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation. For more information on modules, see "Using Modules" in this chapter.

Using Modules

Once a *module* is created, the *import* statement makes that module available to any other program or module. The importing compilation unit receives the constant, type, variable, procedure, and function declarations that are exported by the imported module.

When to Use Modules

Use modules for separate compilation when you have an extremely large program, when you would like easy accessibility to libraries, or when you are building shared or executable (XLs) libraries.

Advantages of Using Modules

Some advantages to separate compilation with modules are:

- Many modules can exist within an executable program, but with only one main program.
- When a module changes, you only need to recompile units that refer to the module.
- You can import types and variables from a module without distributing the source. For example, you can extract information from an object file, archive library, or MLIBRARY.
- The types and the object code are in sync. There's no possibility of a mismatch.
- The constant, type, variable, procedure, and function declarations that are not exported are hidden.
- Modules can be placed in shared libraries or XLs.

Using SUBPROGRAM

The *SUBPROGRAM* compiler option turns a Pascal program into a subprogram compilation unit.

For separate compilation, SUBPROGRAM must be included in all compilation units, except the compilation unit containing the outer block. No code is generated for the outer block if used.

When to Use SUBPROGRAM

SUBPROGRAM is recommended for use in compilation units where the global variables won't change much.

SUBPROGRAM Advantages

Using SUBPROGRAM results in smaller object files and less link time. You also get faster access to the first 8K bytes of globals. The SUBPROGRAM option can also be specified with a list of routines to compile as few as one procedure, if RLFILE is used.

SUBPROGRAM Limitations

The variables must be in the exact same order and must be declared with the same types. Otherwise, at run time the global variables used in one compilation unit may not match the actual memory that matches the global variables in a different compilation unit.

To avoid this problem, place all global variable, type, and constant declarations in a file and include (`$INCLUDE$`) those files in all compilation units. If you do not ensure that the variable, type, and constant declarations match in all compilation units, your execution results will be incorrect, but no error will occur at compile time or at link time. Note that global variables, compiled with this option, can not be placed in shared libraries or XLs.

Using GLOBAL/EXTERNAL

The *GLOBAL* and *EXTERNAL* compiler options turn Pascal programs into global and external compilation units. The compiler options must precede the reserved word *program*.

The GLOBAL compiler option:

- Generates symbolic definitions for the global variables in the compilation units.
- Generates code for the outer block and any procedures.

The EXTERNAL compiler option:

- Generates symbolic references for the global variables in the compilation unit.
- Prevents the compiler from generating storage for global variables.
- Does not generate code for the outer block and prevents the compiler from generating an outer block. If there are statements in an outer block, they are ignored.

When to Use GLOBAL/EXTERNAL

Use GLOBAL/EXTERNAL when sharing global information with another language, when the number of global variables are too large to recompile each time, and when building shared libraries or XLs.

GLOBAL/EXTERNAL is also useful when global variables will change often.

GLOBAL/EXTERNAL Advantages

The following are some advantages of using GLOBAL/EXTERNAL:

- When you use GLOBAL/EXTERNAL for separate compilation, the global variables do not need to be listed in the same order.
- Because the variables are matched by name, only as many globals as used need to be declared when using EXTERNAL.
- The storage for globals does not take up space in the program file.
- The global variables can be placed in shared libraries.

GLOBAL/EXTERNAL Limitations

The following are some limitations of using GLOBAL/EXTERNAL:

- All global variables must be declared in the GLOBAL compilation unit.
- Using GLOBAL/EXTERNAL results in slower link time.
- Code that references global variables is not as efficient as code that does not use GLOBAL/EXTERNAL.

Using SUBPROGRAM with GLOBAL

The SUBPROGRAM with GLOBAL compiler options result in Pascal programs that are a mixture of subprogram and global compilation units. These compiler options must precede the reserved word PROGRAM.

Global variables declared here can be referenced in external compilation units.

When to Use SUBPROGRAM with GLOBAL

Use SUBPROGRAM with GLOBAL to allow multiple declarations of additional global variables instead of using just the outer block.

SUBPROGRAM with GLOBAL Advantages

When you use SUBPROGRAM with GLOBAL, you do not have to recompile the outer block if you are not using GLOBAL. This method of separate compilation is similar to using modules.

You do not have to share all variables with other languages, you can share only a few variables, if you wish.

If any of the global variables change, you only need to recompile the units that refer to them.

You can use this to put globals into shared libraries or XLs.

External Interfaces

Your program can interface with its external environment (other routines and files supported by the operating system) by using physical files, external routines, and intrinsics.

A *physical file* is a program-independent entity that the operating system maintains. It can be a permanent file on a disk or other medium, or it can be an interactive file created at a terminal. Your program can manipulate a physical file by associating it with a logical file (a file that the program declares). Chapter 3, "Input/Output," explains physical and logical files, which HP Pascal programs use for input/output.

An *external routine* is a routine that is not in the compilation unit that calls it. Its source language can be HP Pascal, HP C, HP COBOL II/XL, HP FORTRAN 66/V, HP FORTRAN 77, or SPL. Your program can access an external routine by declaring it with the EXTERNAL directive. Chapter 9 explains external routines.

An *intrinsic* is an external routine that can be called by a program written in any language that the operating system supports. An intrinsic can be written in any supported language, but its formal parameters must be of types that have counterparts in all the other supported languages. Your program can access an intrinsic by declaring it with the INTRINSIC directive. You need not declare the intrinsic's entire parameter list, and your program can use an intrinsic function as either a function or a procedure. Refer to Chapter 10 for more information on intrinsics.

3

Input/Output

Input/output depends on files: your program reads input from files and writes output to files. The terms that describe the three varieties of input/output—*sequential, textfile,* and *direct*—also describe the associated files.

This chapter:

- Gives general information about files.
- Explains the predefined file-opening procedures and how they determine whether files are sequential or direct, for input or for output.
- Defines *sequential* as it applies to input/output and files, and explains the predefined routines that support sequential I/O.
- Explains textfile input/output and files, which are subsets of sequential I/O and files (respectively), and explains the routines peculiar to them.
- Defines *direct* as it applies to input/output and files, and explains the predefined routines that support direct I/O.
- Gives the conditions under which files are closed, and tells what happens when a file closes.

Figure 3-1 illustrates the relationships between sequential, textfile, and direct input/output and sequential files, textfiles, and direct files.

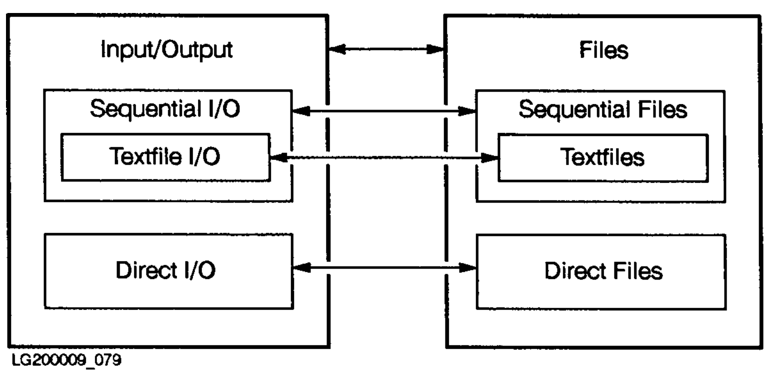


Figure 3-1. Relationships Between I/O Varieties and File Types

Input/output also depends upon the procedures that manipulate files and the functions that return information about them.

Table 3-1 categorizes the predefined input/output routines two ways: by purpose (for example, input or output) and by I/O type (sequential, textfile, or direct).

Table 3-1. Categories of Input/Output Routines

	Sequential I/O	**Textfile I/O**	**Direct I/O**
Opening Procedures	*reset* *rewrite* *append*	*reset* *rewrite* *append*	*open*
Input Procedures	*get* *read*	*get* *read* *readln*	*get* *read* *readdir*
Output Procedures	*put* *write*	*put* *write* *writeln* *page* *prompt* *overprint*	*put* *write* *writedir*
Positioning Procedure	None	None	*seek*
Association Procedures	*associate* *disassociate*	*associate* *disassociate*	*associate* *disassociate*
Status Functions	*eof* *position*	*eof* *eoln* *linepos*	*eof* *lastpos* *maxpos* *position*
Closing Procedure	*close*	*close*	*close*

General File Information

You need the general file information in this section to understand the rest of this chapter. Examine Figure 3-2, and then read the explanations of the entities in italics, whose relationships it shows.

Figure 3-2 illustrates the relationship between physical files (in the operating system environment) and logical files (in the program environment). It also shows how logical files, textfiles, and the standard textfiles *input* and *output* are related.

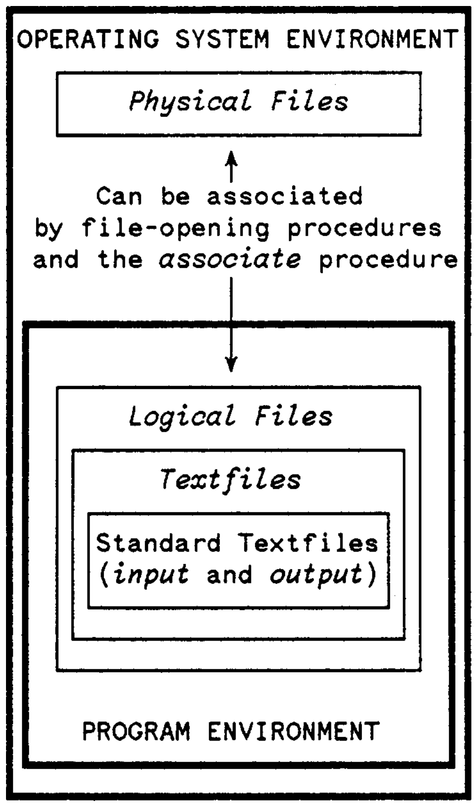


Figure 3-2. File Relationships

Physical Files

A *physical file* is a program-independent entity that the operating system controls. It can be a file on a disk or other medium, or an interactive file created at a terminal (refer to your operating system manual for information on creating and controlling physical files).

Your program can manipulate a physical file if the physical file is associated with one of the program's logical files. In this case, the physical file assumes the characteristics of the logical file.

Logical Files

A *logical file* is a data structure that a program declares and controls. It is a sequence of components of the same type.

The declaration of a logical file determines the type of its components but not their number. A logical file that is declared FILE OF *x* has components of type *x*. File operations can change the number of file components.

A logical file does not exist outside the main program or routine that declares it. If it is associated with a physical file, however, anything that happens to the logical file within the program also happens to the physical file. This is how a program can manipulate its external environment.

Note In subsequent sections of this chapter, the term *file* refers to a logical file unless otherwise noted.

Textfiles

A *textfile* is a logical file that is subdivided into lines, each of which ends with an end-of-line marker. The components of a textfile are of type *char*, but a textfile declaration specifies the type *text*, not FILE OF *char*.

The standard files *input* and *output* are textfiles. If you declare them in the program header, they are the default file parameters for all of the sequential input and output procedures, respectively.

Example

```
PROGRAM prog (input,output);

VAR
   tfile : text;
       c : char;

BEGIN
   .
   .
   .
   read(tfile, c);   {Reads from tfile}
   read(c);          {Reads from input}
   write(c);         {Writes to output}
END.
```

The preceding program has three textfiles: the standard textfiles `input` and `output`, and the file `tfile`.

End-of-line markers are not file components, and are not of type char. The predefined procedure *writeln* writes them to the file (see "Textfile Input/Output"). An end-of-line marker always precedes the end-of-file mark in a textfile, whether *writeln* wrote the last component to the file or not.

Current Position Indexes

Every logical file has a *current position index* that indicates either its current component, an end-of-file marker, or (in a textfile) an end-of-line marker. This index is an integer—the ordinal number of the current component or marker. A file's *current component* is the component that the next I/O operation on that file will input or output.

Figure 3-3 illustrates the relationship between *current position index* and *current component.*

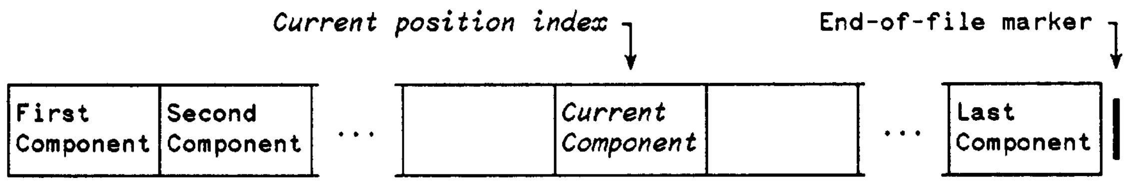


**Figure 3-3.
Relationship Between Current
Position Index and Current Component**

File Buffer Variables and Selectors

Every logical file has a *file buffer variable*, or *buffer*, which is a variable of the same type as the file components. Some file operations assign the value of the current component to the buffer; other operations leave the buffer undefined.

When the buffer is defined, you can access its value with its *file buffer selector*. The file buffer selector for the file `f` is `f^` or `f@`.

Accessing an undefined buffer causes an error.

Opening Files

Except when using *input* and *output* files, your program must open files before it can use them. A call to a predefined file-opening procedure has the following syntax and parameters.

Syntax

$$\left\{ \begin{array}{l} \texttt{reset} \\ \texttt{rewrite} \\ \texttt{append} \\ \texttt{open} \end{array} \right\} (\textit{logical\_file} \left[, \textit{physical\_file} \left[, \textit{open\_options} \right] \right])$$

Parameters

`reset`, `rewrite`, `append`, `open`	The names of the predefined file-opening procedures. See Table 3-2 for more information on them.
logical_file	The name of the logical file to be opened.
physical_file	A string or PAC expression whose value is the name of the physical file to be opened. The syntax of the file name is system-dependent (see Appendix A for the MPE/iX operating system or Appendix B for the HP-UX operating system).
open_options	A string or PAC expression whose value is a list of file attributes. The syntax of the list is system-dependent (see Appendix A for the MPE/iX operating system or Appendix B for the HP-UX operating system).

Example 1

```
reset(logfile);
rewrite(logfile2,physfile2);
append(lfile1,pfile1,'SHARED');  {HP-UX operating system ignores 'SHARED'}
open(lfile1);
```

If you specify *physical_file*, the system associates it with *logical_file*. If *logical_file* was previously associated with another physical file, the system closes the other physical file with its data intact and opens a new physical file.

Example 2

```
PROGRAM prog;

VAR
   datafile : FILE OF integer;

BEGIN
   open (datafile, 'file1');  {Logical file datafile is associated with
                               physical file file1.}

   open (datafile, 'file2');  {Physical file file1 is closed.
                               Logical file datafile is associated with
                               physical file file2.}
END.
```

If *logical_file* is not a program parameter, and *physical_file* is not specified, *logical_file* remains associated with its previously associated physical file. If *logical_file* was not previously associated with a physical file, the system associates *logical_file* with a temporary, nameless physical file.

Example 3

```
PROGRAM prog;  {Logical files logfile1 and logfile2 are not
                program parameters}
VAR
   logfile1,
   logfile2 : text;

BEGIN
   reset(logfile1,'file1'); {Logical file logfile1 is associated with
                             physical file file1.}

   rewrite(logfile1); {No physical file is specified, so logical file
                       logfile1 remains associated with physical file file1.}

   rewrite(logfile2); {No physical file is specified, and logical file
                       logfile2 is not associated with a physical file,
                       so logfile2 is associated with a temporary,
                       nameless physical file.}
END.
```

If *logical_file* is a program parameter, and *physical_file* is not specified, the system opens the physical file that has the same name as *logical_file* (with the lowercase letters upshifted—see Appendix B for HP-UX implications). If no such physical file exists, the result depends on whether either *append* or *rewrite* opened the logical file. If so, the system creates the physical file. If not, it is an error.

Example 4

For this example, assume that the physical file `file1` exists, but the physical file `file2` does not.

```
PROGRAM prog (file1,file2);  {Logical files file1 and file2
                              are program parameters.}
VAR
   file1,
   file2 : text;

BEGIN
   rewrite(file1);  {Logical file file1 is associated with the
                     physical file file1.}

   rewrite(file2);  {Logical file file2 is associated with a
                     physical file file2.  }
END.
```

A temporary, nameless physical file cannot be saved. It becomes inaccessible when the main program or routine that declared *logical_file* terminates, or when you associate *logical_file* with a new physical file.

Your program does not need to open the standard textfiles *input* and *output*. When they are program parameters, the operating system opens them with *reset* and *rewrite,* respectively.

The standard textfiles *input* and *output* are bound to specific system files. For the MPE/iX operating system, see Appendix A; for the HP-UX operating system, see Appendix B.

Table 3-2 summarizes the characteristics of the four predefined file-opening procedures.

Table 3-2. Characteristics of File-Opening Procedures

Procedure	*Reset*	*Rewrite*	*Append*	*Open*
Type of file That it Can Open	Any			Any except textfile
State in Which it Opens File	Read-only	Write-only		Read-Write
Manner in Which file Can Be Accessed	Sequentially			Directly
Purpose for Which it Opens File	Input	Output over old contents	Output at end of old contents	Input and output
Where it Puts Current Position Index *	First component	Before first component	After last component	Before first component
Value of *eof* for File *	*False*	*True*		*False*
Erases Old File Contents	No	Yes	No	
File Buffer Variables *	Contains value of first component	Undefined		

* For a nonempty file. For an empty file, every file-opening procedure puts the current position index before the [nonexistent] first component, *eof* returns *true*, and the file buffer variable is undefined.

Associate Procedure

The predefined procedure *associate* associates a logical file with an open physical file, and puts the current position index at the first component.

Syntax

`associate` (*logical_file, file_number, open_options*)

Parameters

logical_file — The name of the logical file.

file_number — The file number of the open physical file. The physical file must have been opened with a direct call to an operating system routine or a non-Pascal routine. You cannot call the *associate* procedure with the file number of a closed file or a file that was opened with the Pascal procedure `append`, `associate`, `open`, `reset`, or `rewrite`.

open_options — One of the following options. It must be a string literal:

'READ'	Associate with sequential access file with read-only access.
'WRITE'	Associate with sequential access file with write-only access.
'READ,DIRECT'	Associate with direct access file with read-only access.
'WRITE,DIRECT'	Associate with direct access file with write-only access.
'READ,WRITE,DIRECT'	Associate with direct access file with read-write access.
'DIRECT'	Associate with direct access file with read-write access (same as 'READ, WRITE, DIRECT').
'NOREWIND'	Associates with a file without changing the current file position.

You must specify one of the above strings for *open_options*. The system-dependent open options listed in Appendix A (for MPE/iX) and Appendix B (for HP-UX) apply to the file-opening procedures `append`, `open`, `reset`, and `rewrite`. Pascal ignores them when they are used with `associate`.

You cannot specify read access if the physical file is not open for read access, or to specify write access if it is not open for write access. If you associate a logical file with an empty physical file, for read access, the next read causes an error.

Table 3-3 summarizes the characteristics of the predefined procedure *associate*.

Table 3-3. Characteristics of Associate Procedure

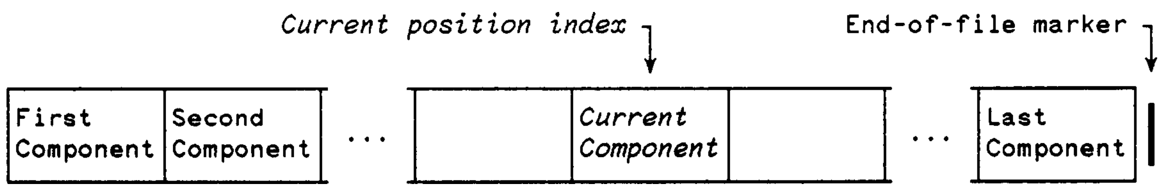
Type of File That it Can Open	Any.
State in Which it Opens File	Specified in *open_options*.
Manner in Which File Can Be Accessed	Either—Defined by characteristics of physical file.
Purpose for Which it Opens File	Input, output or both.
Where it Puts Current Position Index	Before first component.
Value of *eof* for File *	False unless opened for write, in which case *eof* returns true despite possible old data after the current component.
Erases Old file Contents	No.
File Buffer Variables *	First component for a textfile that is open for reading; undefined otherwise.

* For a nonempty file. For an empty file, every file-opening procedure puts the current position index before the [nonexistent] first component, *eof* returns *true*, and the file buffer variable is undefined.

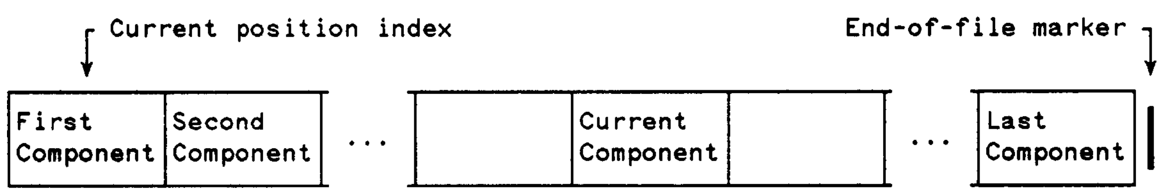
If the physical file is not empty, the first reference to its file buffer variable loads its file buffer with its first component. If the physical file is empty, the first reference to its file buffer variable causes an error.

Figure 3-4 illustrates the effect of the *associate* procedure on the open file whose file number is *file_num*:

Condition of file:



After *associate(examp_file,file_num,'READ')*, the file is open in the read-only state and looks like this:



Now *examp_file* is open in the read-only state.

Figure 3-4. Effect of Associate Procedure on Open File

Example 1

This example applies to HP Pascal on the MPE/iX operating system only. For a description of the MPE/iX intrinsic FOPEN, refer to the *MPE/iX Intrinsics Reference Manual*.

```
PROGRAM test;

TYPE
   pac100 = PACKED ARRAY [1..100] OF char;

VAR
   f      : FILE OF integer;  {f is not a textfile}
   buffer : pac100;
   name   : pac100;
   fnum   : integer;
   j      : integer;
   e,g,h  : text;

FUNCTION FOPEN : shortint; INTRINSIC;  {MPE/&XL; file-opening intrinsic}

BEGIN
   fnum := FOPEN(,0,octal('44'),-4); {open direct access read-write temp. file}
   associate(f,fnum,'READ,WRITE,DIRECT'); {associate with file for
                                           read-write direct access}
   writedir(f,3,5);
   readdir(f,3,j);

   rewrite(e,'UDC');                   {create file 'UDC'}
   writeln('This is a test');
   close(e,'SAVE');                    {close file 'UDC'}
   name := 'UDC';
   fnum := FOPEN(name,octal('40'));    {open 'UDC' for sequential read access}
   associate(g,fnum,'READ');           {associate with 'UDC' for seq. read access}
   read(g,buffer);

   fnum := FOPEN(,4,octal('101'));     {open write access sequential temp. file}
   associate(h,fnum,'WRITE');          {associate for sequential write access}
   writeln(h,'This is a test');
END.
```

Example 2

This example applies to HP Pascal on the HP-UX operating system only. For descriptions of the HP-UX routines `tmpnam` and `open`, refer to the *HP-UX Reference* manual.

```
PROGRAM test;

TYPE
   pac100 = PACKED ARRAY [1..100] OF char;
VAR
   f      : FILE OF integer;  {f is not a textfile}
   buffer : pac100;
   name   : pac100;
   mode   : integer;
   fnum   : integer;
   j      : integer;
   e,g,h  : text;
   option : integer;

{External HP-UX routine that returns a unique file name}
PROCEDURE tmpnam (VAR fpathname : pac100); EXTERNAL;

{External HP-UX routine that opens a file}
FUNCTION file_open $ALIAS 'open'$ {use alias to avoid conflict w/Pascal open}
   (VAR fpathname : pac100;
        foption   : integer;
        mode      : integer) : integer; EXTERNAL;
BEGIN
   tmpnam(name);                          {get unique name for temporary file}
   mode := octal('666');                  {read-write access for file}
   option := octal('402');                {specify read-write access}
   fnum := file_open(name,option,mode);   {open the file}
   associate(f,fnum,'READ,WRITE,DIRECT');{associate with file for
                                           read-write direct access}
   writedir(f,3,5);
   readdir(f,3,j);

   rewrite(e,'UDC');                       {create text file 'UDC'}
   writeln('This is a test');              {write to file}
   close(e,'SAVE');                        {close text file 'UDC'}
   name := 'UDC'#0;                        {open the same file through HP-UX}
   mode := octal('666');
   fnum := file_open(name,0,mode);
   associate(g,fnum,'READ');               {associate with 'UDC' for seq. read access}
   read(g,buffer);
   tmpnam(name);                           {open text file through HP-UX}
   mode := octal('666');
   option := octal('401');                 {specify write access}
   fnum := file_open(name,option,mode);
   associate(h,fnum,'WRITE');              {associate for sequential write access}
   writeln(h,'This is a test');
END.
```

Disassociate Procedure

The predefined procedure *disassociate* removes the logical-physical file association that was previously created with the standard procedure `associate`. As a result, you can no longer use the file `f` with Pascal input and output routines.

Syntax

```
disassociate (f)
```

Parameters

f A variable of type file.

Normally, a file is closed on exit from the block in which it is declared. A disassociated file, however, remains open until it is closed with a direct call to an operating system routine.

`Disassociate` is useful on a file that is opened by a non-Pascal routine that is passed to a Pascal routine and must remain open on exit from the Pascal routine.

Sequential Input/Output

Sequential input/output is input/output that is performed with sequential files; that is, files whose current position indexes advance one component at a time. Sequential input comes from read-only files that the procedure *reset* opened. Sequential output goes to write-only files that the procedure `rewrite` or `append` opened.

Table 3-4 summarizes the characteristics of the predefined sequential input/output procedures.

Table 3-4. Characteristics of Sequential I/O Procedures

Procedure	`get`	`read`	`put`	`write`
State that file must be in *	Read-only or read-write		Write-only or read-write	
Assigns value of	Current component		Buffer	Specified variable
To	Buffer	Specified variable	Current component	
Advances current position index	To next component **			
After call, buffer is undefined	No		Yes	

* For sequential I/O, the state must be read-only or write-only. The state read-write is included here because these sequential I/O procedures work the same way on direct (read-write) files (see "Direct Input/Output").

** For all the procedures except *get*, the current position index is advanced to the component after the assignment. See the explanation of *deferred get* that follows this table.

The procedures `get` and `read` assign values to the buffer with *deferred get*. Deferred get allows HP Pascal to maintain the original Pascal definition of `get` while avoiding unexpected behavior with input from interactive I/O devices (such as terminals).

The procedure `get` advances the current position index to the next component and moves the next component into the buffer variable.

The procedure `reset` opens a file for sequential input, positions the file at the first component, and performs a `get`.

If the `get` (Pascal definition) is performed after a `reset` to a terminal, a physical read is required to fill the buffer variable. Consequently, a program is paused for input from the terminal before the program requests an input operation.

The deferred get avoids this problem. With deferred get, the procedure `get` advances the current position index to the next component and, on the next reference to the buffer variable, moves the current component into the buffer variable. The reference to the buffer variable can be explicit (`f^`) or implicit. For example, `read(f,v)` or `eof(f)`.

Example 1

```
PROGRAM prog;

TYPE
   seqfile = FILE OF char;

VAR
      f1,f2,f3 : seqfile;
         c1,c2 : char;

BEGIN
   reset(f1);     {Opens f1 for sequential input.
                   First component of f1 becomes its current component.}

   c1 := f1^;     {Assigns f1's first component to f1's buffer.
                    Assigns f1's buffer (first component) to c1.}

   get(f1);       {Advances f1's current position index.
                   Second component of f1 becomes its current component.}

   read(f1,c2);   {Implicit reference to f1's buffer --
                   deferred get from get(f1) assigns
                   f1's current (second) component to f1's buffer.
                   Read(f1,c2) assigns f1's current (second) component to c2
                   and advances f1's current position index.
                   Third component of f1 becomes its current component.}

   rewrite(f2);   {Opens f2 for sequential output (write-only).
                   Erases old contents.
                   Leaves f2's buffer undefined.}
   get(f2);       {Illegal -- rewrite(f2) made f2 write-only.}

   f2^ := c1;     {Assigns c1 to f2's buffer.}

   put(f2);       {Assigns f2's buffer (c1) to f2's current (first) component.
                   Advances f2's current position index to position two,
                   where its second component will be after write(f2,c2).}

   write(f2,c2); {Assigns c2 to f2's current (second) component.
                   Advances f2's current position index to position three,
                   where its third component will be.}

   append(f3);    {Opens f3 for sequential output (write only).
                   Does not erase old contents, which end with component n.
                   Leaves f3's buffer undefined.}
```

(*Example is continued on next page.*)

```
    get(f3);      {Illegal -- append(f3) made f3 write-only.}

    f3^ := c1;    {Assigns c1 to f3's buffer.}

    put(f3);      {Assigns f3's buffer (c1) to f3's current (n+1st) component.
                   Advances f3's current position index to position n+2,
                   where its n+2nd component will be after write(f3,c2).}

    write(f3,c2); {Assigns c2 to f3's current (n+2nd) component.
                   Advances f3's current position index to position n+3,
                   where its n+3rd component will be.}
  END.
```

The preceding program reads values from the first and second components of the file `f1` into the variables `c1` and `c2` (respectively). Then it writes `c1` and `c2` to the first and second components of the file `f2` (respectively), and appends them to the file `f3`.

The `get` associated with `read` is implicit; your program need not call `get` explicitly. If it does, a component is skipped.

Example 2

```
PROGRAM prog;

TYPE
   intfile = FILE OF integer;

VAR
       f : intfile;
   x,y,z : integer;

BEGIN
   reset(f);   {Opens f for sequential input.
                First component becomes current component.}

   read(f,x);  {Implicit reference to f's buffer -- deferred get
                from reset(f), above -- assigns current (first)
                component to buffer.  Then read(f,x) assigns
                current (first) component to x.
                Second component becomes current component.}

   read(f,y);  {Implicit reference to buffer --
                deferred get from read(f,x) assigns
                current (second) component to buffer.
                Read(f,y) assigns current (second) component to y
                and advances current position pointer.
                Third component becomes current component.}

   get(f);     {Explicit reference to buffer --
                because get(f) follows read(f,y),
                it advances the current position pointer.
                Fourth component becomes the current component.}

   read(f,z);  {Implicit reference to buffer --
                deferred get from get(f) assigns current (fourth)
                component to buffer.
                Read(f,z) assigns current (fourth) to z.
                Fifth component becomes the current component.}
END.
```

The preceding program assigns the first, second, and fourth components of the file `f` to the variables `x`, `y`, and `z`, respectively. The program skips the third component.

Table 3-5 gives the characteristics of the predefined sequential file functions.

Table 3-5. Characteristics of Sequential File Functions

Function	*Eof*	*Position*
Returns:	*True* if the current position index is at the end-of-file marker; *false* otherwise (always *true* for a write-only file).	Current position index (an integer).
Effect on buffer:	If *eof* returns false, and the buffer does not have a value, then *eof* assigns the value of the current component to the buffer; otherwise, no effect.	None.

Trying to read from file `f` when `eof(f)` is *true* causes a run-time error. You can prevent it by calling `eof(f)` before attempting to read from `f`, and taking appropriate action if `eof(f)` is *true*.

Example 3

```
PROGRAM prog;

TYPE
   seqfile = FILE OF real;

VAR
   f : seqfile;
   i : integer;
   a : ARRAY [1..100] OF real;

BEGIN
   reset(f);  {Open f}
   i := 1;
   WHILE not eof(f) AND (i<=100) DO  {Read array values from f}
      BEGIN
        read(f,a[i]);
        i := i+1;
      END;
   END;
END.
```

If *f* is a terminal, the appropriate action for *eof* is a device read. The next *read* or *readln* of *f* accesses the component in the buffer, without performing another device read.

Example 4

```
PROGRAM prog (input);  {for this example, assume input is from terminal}

TYPE
   readbuf = PACKED ARRAY [1..80] OF char;  {for device read}

VAR
   x : char;
   i : 1..100;
   a : readbuf;

BEGIN
  i := 1;
  WHILE (NOT eof) AND (i <= 100) DO
  BEGIN
    readln(a);                    {perform device read}
    i := i + 1;
  END;
END.
```

By default, `eof` and `readln` apply to the standard textfile `input`. The user running the program terminates input by pressing (RETURN). An input line can have up to 80 characters.

Textfile Input/Output

Textfile input/output is sequential input/output that is performed with textfiles (a subset of sequential files). The program reads textfile input from read-only textfiles opened by the procedure `reset`, or from the standard textfile *input*. The program writes textfile output to write-only textfiles opened by the procedure `rewrite` or `append`, or to the standard textfile `output`.

Table 3-6 summarizes the characteristics of the predefined textfile input/output procedures.

Table 3-6. Characteristics of Textfile I/O Procedures

Procedure	`readln`[1]	`writeln`[2]		`page`	`overprint`	`prompt`
State that file must be in	Read-only	Write-only				
Writes or Reads	Value of current component	Specified expression	End-of-line marker	Page-eject character [3]	Line-feed suppression character[4]	Buffer
To/after	To specified variable	To current component	After current component	After current component		To output device
Advances current position index	To beginning of next line	To beginning of next line		To next component	To beginning of same line	No
After call, buffer is undefined	No	Yes				

1. `readln` and `read` perform implicit data conversion if the specified variable is of any simple type other than *char* (see the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual* for details).
2. `writeln` and `write` format the specified variable (see the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual* for details).
3. The page-eject character causes devices to skip to the top of the next page when it prints the textfile.
4. The line-feed suppression character prevents the device from moving to the next line after it prints the parameter of *overprint*; thus the sequence

   ```
   overprint('ABC');
   writeln('XYZ');''
   ```

   prints `ABC` and then prints `XYZ` on top of it.

The file-opening procedures `rewrite` and `append` and the textfile output procedures `writeln`, `page`, `overprint`, and `prompt` leave the buffer undefined.

Example 1

```
PROGRAM prog (in,out);

VAR
   in,out : text;
   w,x,y,z : char;

BEGIN
   reset(in);          {Open in for textfile input}
   rewrite(out);       {Open out for textfile output}
   readln(in,x,y,z);   {Read x, y, and z from in}
   write(out,x);       {Write x to out}
   overprint(out);     {Write buffer and line-feed suppression to out}
   writeln(out,y);     {Write y to out and advance to next line}
   page(out);          {Write page-eject character to out}
   writeln(out,z);     {Write z to out and advance to next line}
   prompt(out,'?');    {Write '?' to out, without carriage control}
   readln(in,w);       {Read user's answer to '?' from in}
   writeln(out,w);     {Write user's answer to out}
END.
```

When a device prints the file *out,* it prints the value of `y` over the value of `x`, and it prints the values of `z` and `w` on the next page.

Table 3-7 summarizes the characteristics of the predefined textfile functions.

Table 3-7. Characteristics of Textfile Functions

Function	*Eoln*	*Linepos*	
State that file must be in	Read-only	Read-only	Write-only
Returns	*True* if the current position index is at an end-of-line marker; *false* otherwise.	Number of characters read from file since last end-of-line marker (excluding character in buffer). After *readln*, or when current position index is at end-of-line marker, this number is zero.	Number of characters written to file since last end-of-line marker (excluding character in buffer). After *writeln*, or when current position index is at end-of-line marker, this number is zero.
Effect on buffer	If *eoln* returns *true*, it assigns a blank character to the buffer	None	

Example 2

```
PROGRAM prog (infile,outfile,output);

VAR
   infile,
   outfile : text;
         i : integer;
         c : char;

BEGIN
   reset(infile);     {Open infile for input}
   rewrite(outfile);  {Open outfile for output}

   WHILE not(eof(infile)) DO BEGIN  {If infile is not at end-of-file}
      IF eoln(infile) THEN BEGIN    {but the current line of in has ended,}
         writeln(linepos(infile));  {print the number of characters read
                                     from the current line of infile,}
         readln(infile);            {and advance to the next line.}
         writeln(linepos(outfile)); {Also, print the number of characters
                                     written to outfile,}
         writeln(outfile);          {and start a new line of outfile.}
      END {IF}             {If the current line of infile has not ended,}
      ELSE BEGIN
         read(in,c);       {read the next character of infile,}
         write(out,c);     {and write it to outfile.}
      END;
   END; {WHILE}
END.
```

The preceding program copies the textfile `infile` to the textfile `outfile`, writing the values of `linepos(infile)` and `linepos(outfile)` to the standard textfile `output` whenever `eoln(infile)` is *true*.

Except for the *position* function, every sequential I/O procedure and sequential file function applies to textfiles (see "Sequential Input/Output"). Sequential files work the same way, except that for textfiles, `read` (like `readln`) sometimes performs implicit data conversion, and `write` (like `writeln`) can format the output value. See the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for information on implicit data conversion and formatting output values.

Direct Input/Output

Direct input/output is input/output that is performed with direct files; that is, files whose current position indices can be manipulated directly by the program. Direct input and output come from read-write files opened by the procedure `open` (they cannot be textfiles). Your program can use the same direct file for input and output.

Table 3-8 summarizes the characteristics of the predefined direct I/O procedures. (The I/O procedures in Table 3-3 also work on direct access files.)

Table 3-8. Characteristics of Direct I/O Procedures

Procedure	*Readdir*	*Writedir*	*Seek*
State that file must be in	Read-write		
Assigns value of	Specified component	Specified variable	Not applicable
To	Specified variable	Specified component	Not applicable
Advances current position index	To component following specified component		To specified component
After call, buffer is undefined	No	Yes	

The procedures `readdir`, `writedir`, `seek`, `read`, and `write` have this relationship:

This	Is equivalent to this
`readdir(f,i,x);`	`seek(f,i);` `read(f,x);`
`writedir(f,i,x);`	`seek(f,i);` `write(f,x);`

Example 1

```
PROGRAM prog;
TYPE
   dirfile = FILE OF integer;
VAR
             f : dirfile;
   i1,i2,i3,i4 : integer;
BEGIN
   open(f);           {Opens f for direct input/output}

   {READ TWO SPECIFIC COMPONENTS USING readdir AND read}
   readdir(f,50,i1);  {Puts the current position index at component 50.
                       Assigns component 50 to i1.
                       Advances the current position index.
                       Component 51 becomes the current component.}
   read(f,i2);        {Assigns component 51 to i2.}

   {READ TWO SPECIFIC COMPONENTS USING seek AND read}
   seek(f,70);        {Puts the current position index at component 70.}
   read(f,i3);        {Assigns component 70 to i3.
                       Advances the current position index.
                       Component 71 becomes the current component.}
   read(f,i4);        {Assigns component 71 to i4.}

   {WRITE TWO SPECIFIC COMPONENTS USING writedir AND write}
   writedir(f,10,i1); {Puts the current position index at component 10.
                       Assigns i1 to component 10.
                       Advances the current position index.
                       Component 11 becomes the current component.}
   write(f,i2);       {Assigns i2 to component 11.}

   {WRITE TWO SPECIFIC COMPONENTS USING seek AND write}
   seek(f,30);        {Puts the current position index at component 30.}
   write(f,i3);       {Assigns i3 to component 30.
                       Advances the current position index.
                       Component 31 becomes the current component.}
   write(f,i4);       {Assigns i4 to component 31.}
END.
```

All of the sequential I/O procedures work the same way on direct files; that is, they treat them like sequential files. If you use both sequential and direct I/O procedures on a file, the following guidelines apply:

- After the sequential input procedure *read*, any reference to the buffer—even an explicit assignment to the buffer such as `f^ := 30`—assigns the value of the next component to the buffer.
- Because the components of a direct file can be written in any order, your program can skip components when it writes to a file directly. If your program reads the file sequentially later, the values of the skipped components are unpredictable.

- The file-opening procedure `open` and the direct I/O procedures `seek` and `writedir` leave the buffer undefined. After calling one of these procedures, your program must call `get`, `read`, or `readdir` before referencing the buffer implicitly (with a sequential I/O procedure) or explicitly.

Table 3-9 summarizes the characteristics of the predefined direct file functions.

Table 3-9. Characteristics of Direct File Functions

Function	*Lastpos*	*Maxpos*	*Eof*
State that file must be in	Read-write		
Returns	Position number of highest-numbered component that you can read (the last component ever written)	Position number of highest-numbered component that you can write	Returns *true* if current position index is after lastpos; *false* otherwise

All of the sequential file functions work the same way on direct files, except for a subtle difference in the `eof` function (compare Table 3-5 and Table 3-9).

Example 2

```
PROGRAM prog;

TYPE
   cfile = FILE OF char;

VAR
   f : cfile;
   c : char;

BEGIN
   reset(f);                          {Opens file for sequential input.}
   WHILE not(eof(f)) DO read(f,c); {Reads until eof is true.}
   read(f,c);                         {ERROR -- cannot read when eof is true.
                                       This statement would abort the program.}

   open(f);                           {Opens file for direct input/output.}

   IF lastpos(f) < maxpos(f) THEN BEGIN
      seek(f,lastpos(f)+1);           {Puts current position index beyond
                                       last component, making eof true.}

      read(f,c);                      {ERROR -- cannot read beyond lastpos(f).}

      write(f,c);                     {Writes beyond last component.
                                       The component written becomes the last.}
   END;
END.
```

Closing Files

When your program closes a file, it breaks the association between the logical file and the physical file; therefore, it cannot access the file or file buffer variable. It must reopen the file before attempting to operate on it in any other way, or it is a run-time error. One way to close a file is with the predefined procedure `close`. A call to `close` has the following syntax and parameters.

Syntax

`close (`*logical_file* `[,` *close_option*`])`

Parameters

logical_file The name of the logical file to be closed.

close_option A string or PAC expression whose value is one of the following:

`SAVE` or `LOCK`	The file is saved permanently.
`TEMP` or `NORMAL`	The file is saved temporarily. What happens to the temporary file when the current session or job ends is system-dependent. For the MPE/iX operating system, see Appendix A; for HP-UX, see Appendix B.
`CRUNCH`	The effect of this option on the space after the end-of-file marker is system-dependent. See Appendix A (MPE/iX) or Appendix B (HP-UX).
`PURGE`	The file is removed.

A program also closes a logical file and its associated physical file when the program:

- Terminates.
- Exits the routine that declares the file, either because the routine ends, because it executes a *goto* statement that transfers control to a routine outside its scope, or it calls the predefined procedure `escape` because of a run-time error Chapter 11 explains `escape`).
- Reopens the file (in which case the file is closed before it is reopened).

Also, a program closes a file that is stored on the heap when it deallocates the file's heap space by calling the predefined procedure `dispose` or `release` with the appropriate parameter (see Chapter 6).

A program closes a pre-existing physical file (one that it did not create) in the same state that it was in before the program opened it. If a program creates a file, however, it can specify the state in which the `close` procedure closes it.

Example

```
PROGRAM prog;

LABEL
   9999;

TYPE
   ftype = FILE OF integer;

VAR
   f1 : ftype;

PROCEDURE p;
VAR
   f2 : ftype;
BEGIN
   reset(f2);  {Opens f2}
   goto 9999;  {Closes f2 and f3}
END;

PROCEDURE q;
VAR
   f3 : ftype;
BEGIN
   open(f3);  {Opens f3}
   p;
   {p never returns here}
END;

BEGIN
   rewrite(f1);       {Opens f1}
   q;
   9999 : reset(f1);  {Closes and reopens f1}
   close(f1);         {Closes f1}
END.
```

4

Predefined Pascal Constants, Data Types, and Modules

This chapter:

- Gives the value of each predefined constant.
- Gives the range of each predefined data type.
- Explains in detail the predefined data types *bit16*, *bit32*, *bit52*, *longint*, and *shortint*, which are unique to HP Pascal.
- Explains each predefined module.

Values of Predefined Constants

HP Pascal's two predefined constants and their values are:

Constant	Value
minint	-2147483648
maxint	2147483647

When you wish to use the minimum integer, you must use the predefined constant *minint* and not the actual value.

Ranges of Predefined Data Types

Table 4-1 gives the range and size of each predefined data type available to HP Pascal. The data types are in alphabetical order and the sizes are in bits. To get a size in bytes, divide the number of bits by eight.

Table 4-1. Ranges and Sizes of Predefined HP Pascal Types

Type	Range	Unpacked Size in Bits
Bit16	`0..65535`	16
Bit32	$0..2^{32}-1$	32
Bit52	$0..2^{52}-1$	64
Boolean	FALSE or TRUE, where FALSE=0 and TRUE=1	8
Char	ASCII character set	8
Integer	$-2^{31}..2^{31}-1$	32
Longreal †	$-1.797693134862315*10^{308}..-4.940656458412466*10^{-324}$, 0, $4.940656458412466*10^{-324}..1.797693134862315*10^{308}$	64
Real †	$-3.402823*10^{38}..-1.401298*10^{-45}$, 0, $1.401298*10^{-45}..3.402823*10^{38}$	32
Shortint	`-32768..32767`	16
Longint	$-2^{63}..2^{63}-1$	64

† The range of values for longreal and real include denormalized numbers.

Note HP and IEEE floating point numbers are identical. HP3000_16 floating point numbers are different from HP and IEEE floating point numbers. For details, refer to the *Introduction to MPE XL for MPE V Programmers*.

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 65535 when stored.

To determine if a type `T` is assignment compatible with *bit16*, treat *bit16* as a subrange of integer:

- If variable `v` is of type `T` and variable `b16` is of type *bit16*, then the assignment `b16 := v` is legal if the value of `v` is within the range 0..65535.
- If the ranges of `T` and *bit16* do not overlap, the assignment `b16 := v` causes a compile-time error.
- If the ranges of `T` and *bit16* do overlap, but the value of `v` is outside the range of *bit16*, then the assignment `b16 := v` causes a run-time error.

Example

```
PROGRAM prog;

TYPE
   T1 = integer;     {overlaps bit16 range        }
   T2 = -32768..-1;  {does not overlap bit16 range}
   T3 = 0..65535;    {overlaps bit16 range        }

VAR
   v1 : T1;  {b16:=v1 may be legal, depending on value of v1}
   v2 : T2;  {b16:=v2 is never legal}
   v3 : T3;  {b16:=v3 is always legal}
  b16 : bit16;

BEGIN {prog}
   v1  := 65535;
   b16 := v1;       {legal}
   b16 := b16 + 5;  {legal; now b16 = (65540 MOD 65535) = 4}
   b16 := b16 - 5;  {legal; now b16 = 65535}

   v3 := 65535;
   v3 := v3 + 4;    {causes run-time error}
   v3 := 4;
   v3 := v3 - 5;    {causes run-time error}

   v1  := -20;
   b16 := v1;       {causes run-time error}

   v2  := -30;
   b16 := v2;       {causes compile-time error}
END. {prog}
```

Bit32

The predefined data type *bit32* is a subrange, $0..2^{32}$-1, that is stored in 32 bits. *bit32* is a unique HP Pascal type because arithmetic operations on *bit32* data are performed as unsigned 32-bit integers. Unsigned addition and subtraction do not overflow. Unsigned multiply can overflow unless the compiler option OVFLCHECK is used.

Note that there are no *bit32* constants in the compiler. Therefore, numbers in the range `maxint + 1..`2^{32-1} can not be expressed directly. The function `hex` can be used with the compiler options TYPE_COERCION and RANGE to provide *bit32* constants. The compiler option TYPE_COERCION is also needed when initializing a *bit32* constant field. In this case, `bit32()` is not used. When *bit32* is used in an executable statement, `RANGE OFF` must be used.

To determine if a type `T` is assignment compatible with *bit32*:

- If variable `v` is of type `T` and variable `b32` is of type bit32, then the assignment `b32 := v` is legal if the value of `v` is within the range $0..2^{32}$-1.
- If the ranges of `T` and bit32 do not overlap, the assignment `b32 := v` causes a compile-time error.
- If the ranges of `T` and bit32 do overlap, but the value of `v` is outside the range of bit32, then the assignment `b32 := v` causes a run-time error.

Example

```
$standard_level 'hp_modcal'$
program prog_bit32(output);

var i : integer;
    b : bit32;

type rec = record
           f1 : bit32;
           end;
$push; type_coercion 'conversion'$
const v_rec = rec[f1: hex('ffffffff')];      { bit32 constant field }
$pop$
begin
b := hex('ffffffff');   { compile-time error }
i := -1;

try
b := i;                 { run-time error }
recover ;

$push; type_coercion 'conversion'; range off$
b := bit32(i) + 1;      { zero is stored }

b := bit32(hex('ffffffff'));
$pop$

try
i := b;                 { run-time error }
recover ;

try
i := b + i;             { b and i are converted to longint and are }
                        { too big to fit back into i }
recover ;

i := hex('ffffffff');   { both b and i now have all bits on }

{ the following never prints since i is sign extended to longint and
  b is zero extended to longint }
if i = b then writeln('equal');
end.
```

Bit52

The predefined data type *bit52* is a subrange, 0..2^{52}-1, that is stored in 64 bits. *bit52* is a unique HP Pascal type because arithmetic operations on *bit52* data are performed as unsigned 64-bit integers. Unsigned addition and subtraction do not overflow. Unsigned multiply may overflow. The compiler option OVFLCHECK has no effect.

Note that there are no *bit52* constants in the compiler. Therefore, numbers in the range `maxint + 1..`$2^{52\ -1}$ can not be expressed directly. The function `hex` can be used with the compiler options TYPE_COERCION and RANGE to fill part of this range. The compiler option TYPE_COERCION is also needed when initializing a *bit52* constant field. In this case, `bit52()` is not used. When *bit52* is used in an executable statement, `RANGE OFF` must be used.

For number in the range of 2^{32}..2^{52-1}, a run-time computation must be done. If the numbers are all constants, they must be type coerced to *bit52* so they do not integer overflow.

Variant records can also be used to build up these large constants.

To determine if a type `T` is assignment compatible with *bit52*.

- If variable `v` is of type `T` and variable `b52` is of type *bit52*, then the assignment `b52 := v` is legal if the value of `v` is within the range 0..2^{52}-1.
- If the ranges of `T` and *bit52* do not overlap, the assignment `b52 := v` causes a compile-time error.
- If the ranges of `T` and *bit52* do overlap, but the value of `v` is outside the range of *bit52*, then the assignment `b52 := v` causes a run-time error.

Example

```
$standard_level 'hp_modcal'$
program prog_bit52(output);

var i : integer;
    b : bit52;

type rec = record
           f1 : bit52;
           end;
$push; type_coercion 'conversion'$
const v_rec = rec[f1: hex('fffffff')];      { bit52 constant field }
$pop$
begin
b := hex('fffffff');   { compile-time error }
i := -1;

try
b := i;                { run-time error }
recover ;
```

(*Example is continued on next page.*)

```
$push; type_coercion 'conversion'; range off$
b := bit52(i) + 1;      { zero is stored }

b := bit52(hex('ffffffff'));
$pop$

try
i := b;                 { run-time error }
recover ;

try
i := b + i;             { b and i are converted to longint and are }
                        { too big to fit back into i }
recover ;

i := hex('ffffffff');   { both b and i now have all bits on }

{ the following never prints since i is sign extended to longint and
  b is zero extended to longint }
$push; type_coercion 'conversion'$
if longint(i) = b then writeln('equal');
$pop$
end.
```

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.) The type *shortint* has the following uses:

- If you want to access an external non-Pascal routine that has a formal parameter of a type whose range is -32768..32767, and uses 16-bits of storage, you can declare a corresponding formal Pascal parameter of type *shortint*, and it will be compatible.
- For Pascal/V compatibility.

To determine whether a type `T` is assignment compatible with the type *shortint*, you can treat *shortint* as a subrange of *integer*. This means that you can assign a variable `v` of type `T` to a variable `sv` of type *shortint* if:

- The type `T` is *integer* or a subrange of *integer*.
- The value of `v` is within the range of *shortint* (-32768..32767).

  If the ranges of `T` and *shortint* do not overlap, the assignment `sv:=v` causes a compile-time error. If the ranges of `T` and *shortint* do overlap, but the value of `v` is outside the range of *shortint* the assignment `sv:=v` causes a run-time error.

Example

```
PROGRAM prog;

TYPE
   T1 = integer;       {overlaps shortint range}
   T2 = -10..40000;    {overlaps shortint range}
   T3 = 40000..50000;  {does not overlap shortint range}

VAR
   v1 : T1;            {sv:=v1 may be legal, depending on value of v1}
   v2 : T2;            {sv:=v2 may be legal, depending on value of v2}
   v3 : T3;            {sv:=v3 is never legal}
   sv : shortint;

BEGIN
   v1 := 10;
   sv := v1;      {legal assignment}

   v1 := 45000;
   sv := v1;      {causes run-time error}

   v2 := 400;
   sv := v2;      {legal assignment}

   v2 := 35000;
   sv := v2;      {causes run-time error}

   v3 := 40000;
   sv := v3;      {causes compile-time error}
END.
```

Longint

The predefined data type *longint* is an integer in the range $-2^{63}..2^{63}-1$ that is stored in 64 bits. The compiler option OVFLCHECK has no effect on 64 bit multiply.

Note that there are no *longint* constants in the compiler. Therefore, numbers outside of the range `minint .. maxint` can not be expressed directly. The compiler option TYPE_COERCION must be used with a run-time computation. If the numbers are constants, they must be typed coerced to *longint* so they do not integer overflow.

Example

```
$standard_level 'hp_modcal'$
program prog_longint(output);

var i : integer;
    b : longint;

type rec = record
           case integer of
           0:(l    : longint);
           1:(f1,f2: integer);
           end;
const v_rec = rec[f1: hex('1'),
                  f2: hex('ffffffff')];  { longint constant field }
begin
b := v_rec.l;
writeln(b);

try
i := b;                 { run-time error }
recover ;

$push; type_coercion 'conversion'$
b := longint(1000000) * 1000000;
$pop$

writeln(b);
end.
```

Output:

```
  8589934591
1000000000000
```

Predefined Modules

On both the MPE/iX and HP-UX operating systems, HP Pascal has these predefined modules:

- `stdinput`
- `stdoutput`

On the HP-UX operating system only, HP Pascal has these additional predefined modules:

- `stderr`
- `arg`
- `pas_hp1000`

In its import declaration section, your program can import any or all of the predefined modules supported by the operating system on which it runs.

This section shows the actual declarations in the predefined modules for your information only. Do not include these declarations in your program. Instead, import the predefined modules as shown on the following page.

stdinput

The `stdinput` module contains the declaration for the predefined global variable (standard textfile) *input*. It allows an independent module (which has no program header) to use *input*. Importing the `stdinput` module into an independent module is the same as declaring *input* in the program header of a program.

The content of the predefined module `stdinput` is:

```
VAR
   input : text;
```

stdoutput

The `stdoutput` module contains the declaration for the predefined global variable (standard textfile) *output*. It allows an independent module (which has no program header) to use *output*. Importing the `stdoutput` module into an independent module is the same as declaring *output* in the program header of a program.

The content of the predefined module `stdoutput` is:

```
VAR
   output : text;
```

stderr

The stdrrr module contains the declaration for the predefined global variable (standard textfile) *stderr*. It allows an independent module (which has no program header) to use *stderr*. Importing the stderr module into an independent module is the same as declaring *stderr* in the program header of a program.

The content of the predefined module stderr is:

```
VAR
   stderr : text;
```

The predefined module stderr is only available on the HP-UX operating system.

The main use of stdinput, stdoutput, and stderr is to allow a module to perform a read or write operation to either standard input files, standard output files, or, on HP-UX, standard error files. The module must import the corresponding stdinput, stdoutput, or stderror modules, and the program must have input, output, or stderr in the program header. A main program does not need to import these standard modules, but the corresponding program parameter must be present in the program header.

The following example shows a program importing a module that imports stdinput, stdoutput, and, on HP-UX, stderr.

```
MODULE A;
EXPORT
   Procedure getnum (var n:integer);
IMPLEMENT
IMPORT
   stdinput, stdoutput, stderr;
   Procedure getnum (var n: integer);
   BEGIN
      Writeln ('Enter a positive number')  {Writes to output.}
      Readln (n);                          {Reads from input.}
      IF n < 0 THEN
        Writeln (stderr, 'Incorrect input')  {Writes to stderr.}
   END;
END.
```

The program below shows how module A is imported. It is compiled into file A.o. The program parameters input, output, and stderr must be present since module A imports them. arg and pas_hp1000 do not need to be present if they are imported.

```
Program Test (input, output, stderr);
$search 'A.o'$ { search A.o for module A }
IMPORT A:
VAR
  m : integer;
BEGIN
  getnum(m);
 .
 .
 .
END.
```

arg

The `arg` module contains routines that access HP-UX command line arguments. (It also contains the types that these routines use, but only the routines are presented here.)

The routines in the predefined module `arg` are:

Routine	Effect and Declaration
`argc`	Returns the total number of arguments to the program (the name of the program is considered to be the first argument). **Declaration:** `FUNCTION argc : integer;`
`argv`	Returns a pointer to an array of pointers to the actual arguments. **Declaration:** `FUNCTION argv : argarrayptr;`
`argn`	A specific argument, in the form of a Pascal string. **Declaration:** `FUNCTION argn (argnum : integer) : String1024;`

The predefined module `arg` is only available on the HP-UX operating system.

pas_hp1000

The pas_hp1000 module contains routines that help you migrate Pascal/1000 programs to HP Pascal/HP-UX on the HP 9000 Series 700 or 800 machine. They emulate user-callable routines in the Pascal/1000 run-time library.

The routines in the predefined module pas_hp1000 are:

Routine — **Effect and Declaration**

pas_init_hp1000_args — Only for programs running under the RTE shell on the HP 9000 Series 700 or 800. Using command line arguments, it sets up an HP-UX-like argument array for use in argument-accessing routines.

Declaration:

```
PROCEDURE pas_init_hp1000_args;
```

pas_parameters — Returns a specific argument to the program as a Pascal PACKED ARRAY OF CHAR.

Declaration:

```
FUNCTION pas_parameters
   (       position  : shortint;
    ANYVAR Parameter : Pas_PAC80;  {any PAC}
           maxlen    : shortint
   ) : shortint;
```

pas_sparameters — Returns a specific argument to the program as a Pascal string.

Declaration:

```
FUNCTION pas_sparameters
   (    position  : shortint;
    VAR Parameter : String;  {Any string}
   ) : shortint;
```

pas_numericparms — Interprets the arguments to the program as an array of numeric strings and returns an array of numbers corresponding to these strings.

Declaration:

```
PROCEDURE pas_numericparms
   (ANYVAR ParmArray : Pas_ParmArray);
```

pas_getnewparms — Only for programs running under the RTE shell on the HP 9000 Series 700 or 800. Reinitializes the argument data structures when the program has been rescheduled after being suspended.

Declaration:

```
PROCEDURE pas_getnewparms;
```

`pas_filenamr` — Returns the name of the physical file associated with the specified logical file.

Declaration:

```
FUNCTION pas_filenamr
   (ANYVAR f : text) : pas_nametype;
```

`pas_timestring` — Returns the time of day as a 26-character PACKED ARRAY OF CHAR.

Declaration:

```
PROCEDURE pas_timestring
   (ANYVAR f : pas_timestringtype);
```

`pas_traceback` — Produces a stack trace of the program and writes it to `stderr`.

Declaration:

```
PROCEDURE pas_traceback
   (dummy : shortint);  {parameter is ignored}
```

`pas_stringdata1`
`pas_stringdata2` — Return pointers to the data portion of a string. Functionally identical; provided as different entry points for consistency with Pascal/1000 names.

Declarations:

```
FUNCTION pas_stringdata1
   (VAR s : String) : localanyptr;

FUNCTION pas_stringdata2
   (VAR s : String) : localanyptr;
```

The predefined module `pas_hp1000` is only available on the HP-UX operating system.

5

Allocation and Alignment

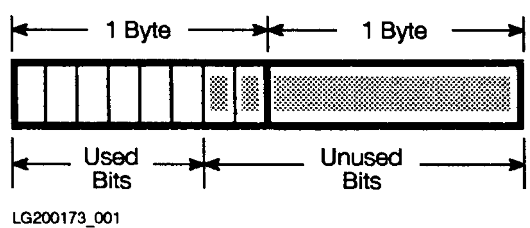
This chapter:

- Defines *allocation, alignment,* and *packing algorithm*.
- Shows how unpacked and packed variables are allocated and aligned.
- Tells how entire arrays and records are allocated and aligned (whether they are unpacked, packed, or crunched).
- Shows how array elements and record fields are allocated and aligned when they are unpacked, packed, and crunched.
- Explains how enumeration and subrange types are related and shows how they are allocated and aligned.
- Explains how files, sets, and strings are allocated and aligned.

Note This chapter applies to the HP Pascal packing algorithm, which is the default. On the MPE/iX operating system, the compiler option HP3000_16 specifies the Pascal/V packing algorithm instead. For information on the HP3000_16 compiler option, refer to the *HP Pascal/iX Reference Manual*. For information on the Pascal/V packing algorithm, see Appendix A in this manual.

In diagrams in this section, bold lines are byte boundaries and fine lines are bit boundaries. Unused bits and bytes are shaded.

Example

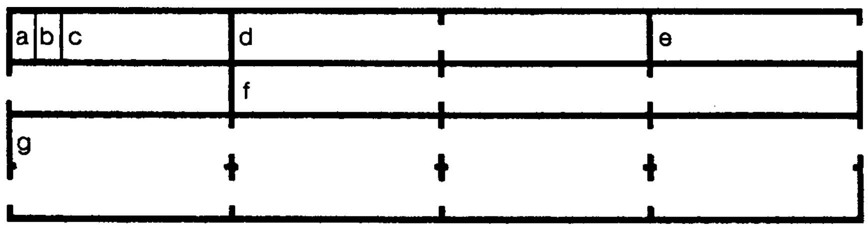


Note that:

- Zero represents the Boolean value FALSE, and one represents TRUE.
- The leftmost bit represents the sign of a signed integer value.

Byte boundaries are broken where a variable crosses them. Bit boundaries are omitted where a variable crosses them. A space that is allocated to a variable contains the variable's name. If the name does not fit the space, it is printed outside, with an arrow pointing to the space.

Example



LG200173_002

The variables `a` and `b` occupy one bit each, `c` occupies six bits, `d` and `e` occupy two bytes each, `f` occupies three bytes, and `g` occupies eight bytes.

Allocation, Alignment, and Packing Algorithm

Allocation is the assignment of memory to variables. When the compiler assigns one byte of memory to the variable `x`, you can say that both the byte and `x` are *allocated* (the byte is allocated to `x`, and `x` is allocated one byte of memory).

Alignment refers to the position at which a variable's share of memory begins. There are several types of alignment.

- Bit-aligned: If the byte that the compiler allocates to `x` can begin on a bit boundary.
- 1-byte-aligned: If the byte that the compiler allocates must begin on a byte boundary.
- 2-byte-aligned: If the byte that the compiler allocates must begin on a 2-byte boundary.
- 4-byte-aligned: If the byte that the compiler allocates must begin on a 4-byte boundary.
- 8-byte-aligned: If the byte that the compiler allocates must begin on a 8-byte boundary.

For the list of possible alignments, refer to "ALIGNMENT" in the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

Example

LG200173_003

The variables `c` and `d` are allocated one byte each, but `c` is bit-aligned and `d` is byte aligned.

A *packing algorithm* determines a variable's allocation and alignment, and the allocation and alignment of its elements or fields, if it has them. The HP Pascal packing algorithm uses the following factors to allocate and align a particular variable:

- Variable type.
- Whether the variable (if it is an array, record, or set) is unpacked, packed, or crunched.

When the compiler options TABLES or MAPINFO are ON, the program listing contains packing information. Refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for more information on compiler options.

Unpacked Variables

An *unpacked variable* is either not part of an array or record, or it is part of an unpacked array or record.

Table 5-1 shows how the HP Pascal packing algorithm allocates and aligns unpacked variables of each HP Pascal type. The variable types are in alphabetical order. Sections that Table 5-1 references are in this chapter.

Table 5-1. Allocation and Alignment of Unpacked Variables (HP Pascal Packing Algorithm)

Variable Type	Allocation	Alignment
`Anyptr`	`8 bytes`	`4-byte`
`Array`	See "Arrays"	
`Bit16`	`2 bytes`	`2-byte`
`Bit32`	`4 bytes`	`4-byte`
`Bit52`	`8 bytes`	`4-byte`
`Boolean`	`1 byte`	`Byte`
`Char`	`1 byte`	`Byte`
`Enumeration`	See "Enumerations and Subranges"	
`File`	See "Files"	`8-byte`
`Function`	`8 bytes`	`4-byte`
`Globalanyptr`	`8 bytes`	`4-byte`
`Integer`	`4 bytes`	`4-byte`
`Localanyptr`	`4 bytes`	`4-byte`
`Longint`	`8 bytes`	`4-byte`
`Longreal`	`8 bytes`	`8-byte`
`Pointer`	`4 bytes`	`4-byte`
`Procedure`	`8 bytes`	`4-byte`
`Real`	`4 bytes`	`4-byte`
`Record`	See "Records"	
`Set`	See "Sets"	
`Shortint`	`2 bytes`	`2-byte`
`String`	See "Strings"	`4-byte`
`Subrange`	See "Enumerations and Subranges"	

Packed Variables

A *packed variable* is the element of a packed array or the field of a packed record.

Table 5-2 shows how the HP Pascal packing algorithm allocates and aligns packed variables of each HP Pascal type. The variable types are in alphabetical order. The sections that Table 5-2 references are in this chapter.

Table 5-2.
Allocation and Alignment of Packed Variables
(HP Pascal Packing Algorithm)

Variable Type	Allocation	Alignment
`Anyptr`	`8 bytes`	`4-byte`
`Array`	See "Arrays" for information on entire array and "Packed Arrays" for information on elements.	
`Bit16`	`2 bytes`	`2-byte`
`Bit32`	`4 bytes`	`4-byte`
`Bit52`	`8 bytes`	`4-byte`
`Boolean`	`1 bit`	`Bit`
`Char`	`1 byte`	`Byte in array, bit in record`
`Enumeration`	See "Enumerations and Subranges"	
`File`	`See "Files"`	`8-byte`
`Function`	`8 bytes`	`4-byte`
`Globalanyptr`	`8 bytes`	`4-byte`
`Integer`	`4 bytes`	`4-byte`
`Localanyptr`	`4 bytes`	`4-byte`
`Longint`	`8 bytes`	`4-byte`
`Longreal`	`8 bytes`	`8-byte`
`Pointer`	`4 bytes`	`4-byte`
`Procedure`	`8 bytes`	`4-byte`
`Real`	`4 bytes`	`4-byte`

Table 5-2.
Allocation and Alignment of Packed Variables
(HP Pascal Packing Algorithm) (continued)

Variable Type	Allocation	Alignment
`Record`	See "Records" for information on entire record and "Packed Records" for information on fields.	
`Set`	See "Sets"	
`Shortint`	`2 bytes`	`2-byte`
`String`	See "Strings"	`4-byte`
`Subrange`	See "Enumerations and Subranges".	

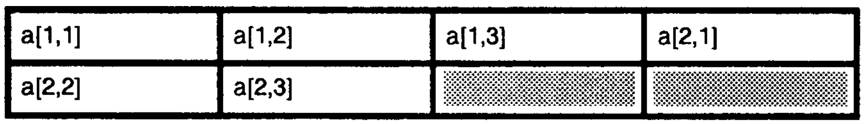
Arrays

Arrays are stored in *row-major order*. This means that an array is stored a row at a time, rather than a column at a time (*column-major order*).

Example

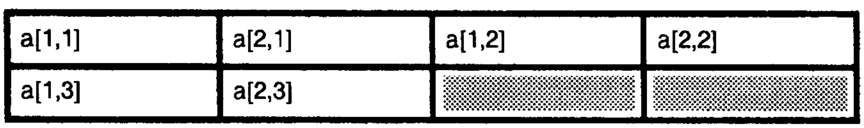
```
VAR
   a : ARRAY [1..2,1..3] OF char;
```

Row-major order:



LG200173_004

Column-major order:



LG200173_004

The HP Pascal packing algorithm uses this formula to allocate an array:

number_of_elements * *space_for_one_element*

The *space_for_one_element* depends upon the array element type and whether the array is unpacked, packed, or crunched. The same factors determine element alignment. See the tables indicated below:

If the array is:	**See:**	**In the section:**
Unpacked	Table 5-1	"Unpacked Variables"
Packed	Table 5-3	"Packed Arrays"
Crunched	Table 5-5	"Crunched Arrays and Records"

Records

A record allocation is the sum of the allocations of the fields in the fixed part and (if the record has them) the allocations of the tag field and the largest field in the variant part, plus trailing bits.

Field allocation depends on field type and whether the record is unpacked, packed, or crunched. The same factors determine field alignment. See the tables indicated below:

If the array is:	**See:**	**In the section:**
Unpacked	Table 5-1	"Unpacked Variables"
Packed	Table 5-4	"Packed Records"
Crunched	Table 5-5	"Crunched Arrays and Records"

The HP Pascal packing algorithm uses these two rules to align a record:

- The entire record is aligned on the same boundary as its most restricted field.
- The variant part of a record is aligned on the same boundary as the most restricted first field of all variants.

Example

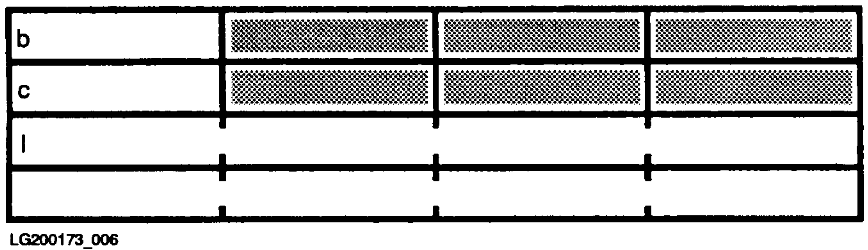
```
TYPE
   Rec = RECORD
            CASE b : Boolean OF
               TRUE  : (c : char;      {1 byte,  1-byte-aligned}
                        l : longreal;  {8 bytes, 8-byte-aligned}
                       );
               FALSE : (i : integer;   {4 bytes, 4-byte-aligned}
                       );
         END;
```

A record of the type `Rec` is 8-byte-aligned because its most restricted field, `l`, must be 8-byte-aligned.

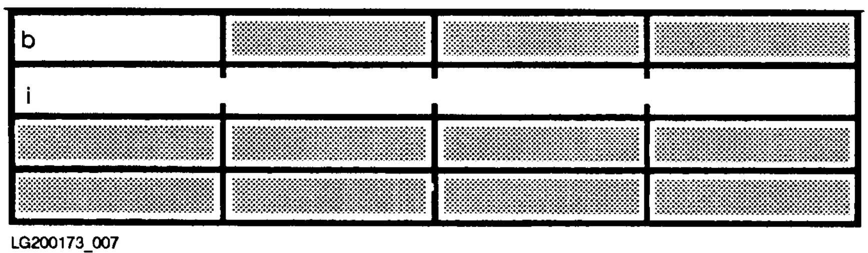
The variant part of a record of type `Rec` is 4-byte-aligned, because the most restricted first field of the two variants, `i`, must be 4-byte-aligned.

A variable of type `Rec` is allocated 16 bytes. The TRUE and FALSE variants are aligned like this:

TRUE Variant



FALSE Variant



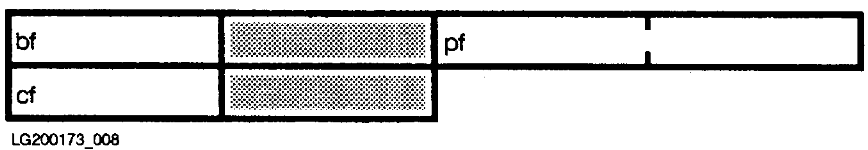
Sometimes you can reduce the space that a record takes by declaring its fields in different order.

Example

```
VAR
   upr1 : RECORD
             bf : Boolean;
             pf : 0..32767;
             cf : char;
          END;

   upr2 : RECORD
             bf : Boolean;
             cf : char;
             pf : 0..32767;
          END;
```

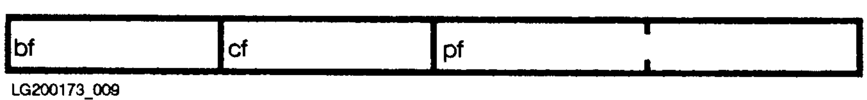
The only difference between the variables `upr1` and `upr2` is the order of their fields.

The variable `upr1` takes six bytes:



Because `pf` must be 2-byte-aligned, it cannot start in the second byte. The extra byte after `cf` is allocated because the most restricted element, `pf`, is 2-byte-aligned.

The variable `upr2` takes four bytes:



Sometimes you cannot reduce the space that a record takes by declaring its fields in different order.

Example

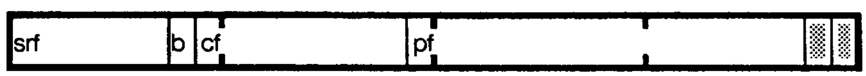
```
VAR
   pr1 : PACKED RECORD
            srf : 0..32;
            b : Boolean;
            pf : 0..32767;
            cf : char;
         END;

   pr2 : PACKED RECORD
            srf : 0..32;
            b : Boolean;
            cf : char;
            pf : 0..32767;
         END;
```

The only difference between the variables `pr1` and `pr2` is the order of their fields.

The variable `pr1` takes four bytes:

LG200173_010

The variable `pr2` also takes four bytes:



LG200173_011

Packed Arrays

Table 5-3 shows how the HP Pascal packing algorithm allocates and aligns the elements of a packed array. The element types are in alphabetical order.

Table 5-3. Allocation and Alignment of Packed Array Elements (HP Pascal Packing Algorithm)

Element Type	Allocation	Alignment
Anyptr	8 bytes	4-bytes
Array, crunched	Same as crunched array that is not part of an array or record (see Table 5-9); then padded to the nearest byte.	Byte
Array, packed	Same as packed array that is not part of an array or record (find element type in this table and use formula in section "Arrays"); then padded to alignment boundary.	Same as element, or byte, whichever is larger.
Array, unpacked	Same as unpacked array that is not part of an array or record (find element type in this table and use formula in section "Arrays").	Same as element.
Bit16	2 bytes	2-byte
Bit32	4 bytes	4-byte
Bit52	8 bytes	4-byte
Boolean	1 bit	1 bit
Char	1 byte	1-byte
Enumeration	See "Enumerations and Subranges".	
File	See "Files".	8-byte
Function	8 bytes	4-byte
Integer	4 bytes	4-byte
Globalanyptr	8 bytes	4-byte
Localanyptr	4 bytes	4-byte
Longint	8 bytes	4-byte
Longreal	8 bytes	8-byte
Pointer	4 bytes	4-byte
Procedure	8 bytes	4-byte
Real	4 bytes	4-byte
Record, crunched	Fields are allocated by type, and record is padded to byte boundary.	Byte

Table 5-3.
Allocation and Alignment of Packed Array Elements
(HP Pascal Packing Algorithm) (continued)

Element Type	Allocation	Alignment
Record, packed	Fields are allocated by type, and record is padded to the alignment boundary.	Largest alignment boundary of any field, or byte, whichever is larger.
Record, unpacked	Fields are allocated by type, and record is padded to the alignment boundary.	Largest alignment boundary of any field.
Set	See "Sets".	
Shortint	2 bytes	2-byte
Strings	See "Strings".	4-byte
Subrange	See "Enumerations and Subranges".	

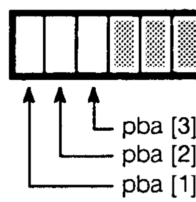
Example

```
VAR
   uba : ARRAY [1..3] OF Boolean;
   pba : PACKED ARRAY [1..3] OF Boolean;
```

The array `uba` takes three bytes:

LG200173_012

The array `pba` takes three bits:



LG200173_013

If an array is not within a crunched structure, the compiler aligns the entire array on the same boundary as its first element, or on a byte boundary.

Declaring an array PACKED has no effect on its elements if the elements are unpacked structures.

Packed Records

Table 5-4 shows how the HP Pascal packing algorithm allocates and aligns the fields of a packed record. The field types are in alphabetical order.

Table 5-4. Allocation and Alignment of Packed Record Fields (HP Pascal Packing Algorithm)

Field Type	Allocation	Field Alignment
Anyptr	8 bytes	4-byte
Array, crunched	Minimum number of bits required to represent any value of the element type.	Bit
Array, packed	Use formula in "Arrays" section and then pad to alignment boundary.	Same as element or byte, whichever is larger.
Array, unpacked	Use formula in "Arrays" section and then pad to alignment boundary.	Same as element.
Bit16	2 bytes	Bit
Bit32	4 bytes	4-byte
Bit52	8 bytes	4-byte
Boolean	1 bit	Bit
Char	1 byte	Bit
Enumeration	See "Enumerations and Subranges".	
File	See "Files".	8-byte
Function	8 bytes	4-byte
Integer	4 bytes	4-byte
Globalanyptr	8 bytes	4-byte
Localanyptr	4 bytes	4-byte
Longint	8 bytes	4-byte
Longreal	8 bytes	8-byte
Pointer	4 bytes	4-byte
Procedure	8 bytes	4-byte
Real	4 bytes	4-byte

Table 5-4.
Allocation and Alignment of Packed Record Fields
(HP Pascal Packing Algorithm) (continued)

Field Type	Allocation	Field Alignment
Record, packed	Fields are allocated by type, and record is padded to the alignment boundary.	Largest alignment of any field or byte, whichever is larger.
Record, unpacked	Fields are allocated by type, and record is padded to the alignment boundary.	Largest alignment of any field.
Set	See "Sets".	
Shortint	2 bytes	2-byte
Strings	See "Strings".	4-byte
Subrange	See "Enumerations and Subranges".	

The field that is aligned on the largest boundary determines the alignment of the entire record. For example, if a record has three fields—one byte-aligned field, one 2-byte-aligned field, and one 4-byte-aligned field—the entire record is 4-byte-aligned.

Packing a record has no effect on fields that are unpacked structures.

Example

```
TYPE
   ua = ARRAY [1..4] OF Boolean;
   ur1 = RECORD
            i : integer;
            c : char;
         END;
VAR
   ur2 : RECORD
            c : char;
            a : ua;
            r : ur1;
         END;

   pr : PACKED RECORD
           c : char;
           a : ua;
           r : ur1;
        END;
```

The fields in `ur2` and `pr` are allocated and aligned identically.

Crunched Arrays and Records

Crunched packing, a systems programming extension, packs a record or array as tightly as possible: it bit-aligns every field or element.

Table 5-5 shows how the HP Pascal packing algorithm allocates elements of crunched arrays or fields of crunched records. If a type is not in Table 5-5, a crunched array or record cannot have elements or fields of that type.

Table 5-5. Allocation of Crunched Array Elements and Record Fields (HP Pascal Packing Algorithm)

Element or Field Type	Allocation
Bit16	2 bytes
Bit32	4 bytes
Bit52	52 bits
Boolean	1 bit
Char	1 byte
Integer[1]	4 bytes
Longint	8 bytes
Shortint	2 bytes
Crunched array[2]	* Minimum #
Crunched record[2]	* Minimum #
Crunched set[1]	* Minimum #
Subrange[1,3]	* Minimum #

(* Minimum number of bits required to represent value.)

1. The value representation has the most significant bit first and the least significant bit last (no byte swapping).
2. If a record or array contains a crunched structure, it must be crunched itself.
3. The value zero is always included in the subrange when calculating the minimum number of bits; for example, this record takes seven bits:

```
CRUNCHED RECORD
   f : 100..101;
END;
```

If any element can be negative, an extra bit is allocated for the sign; for example, this record takes three bits:

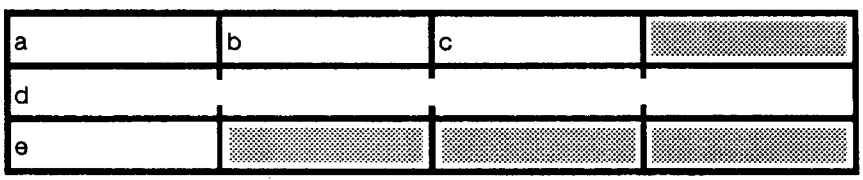
```
CRUNCHED RECORD
  f : -4..3;
END;
```

Example

A record that is defined:

```
TYPE
   u_rec = RECORD          {4-byte aligned}
      a,b : Boolean;
      c : char;
      d : minint..maxint;
      e : Boolean;
   END;
```

is allocated and aligned this way:

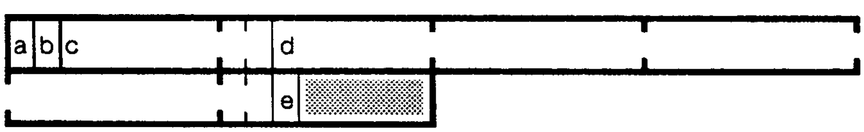


LG200173_014

A record that is defined:

```
TYPE
p_rec1 = PACKED RECORD  {Byte-aligned}
   a,b : Boolean;
   c : char;
   d : minint..maxint;
   e : Boolean;
END;
```

is allocated and aligned this way:

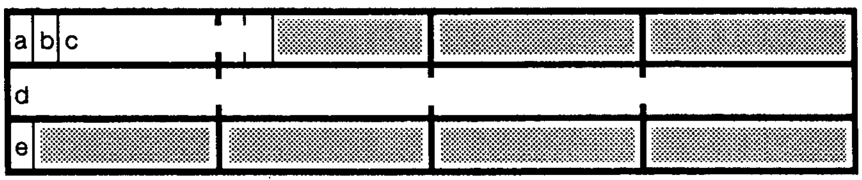


LG200173_015

A record that is defined:

```
p_rec2 = PACKED RECORD {4-byte-aligned}
   a,b : Boolean;
   c : char;
   d : integer;
   e : Boolean;
END;
```

is allocated and aligned this way:



LG200173_016

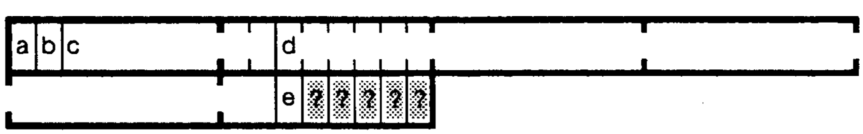
A record that is defined:

```
TYPE
   c_rec1 = CRUNCHED RECORD
      a,b : Boolean;
      c   : char;
      d   : minint..maxint;
      e   : Boolean
   END;
```

Or:

```
TYPE
   c_rec2 = CRUNCHED RECORD
      a,b : Boolean;
      c   : char;
      d   : integer;
      e   : Boolean
   END;
```

is allocated and aligned this way:



LG200173_017

The bits containing question marks are not allocated if the type is used inside another crunched structure.

Crunched Sets

Table 5-6 shows how the HP Pascal packing algorithm allocates and aligns a crunched set when it is the element of an array or the field of a record.

Table 5-6.
Allocation and Alignment of Crunched Sets in Arrays and Records
(HP Pascal Packing Algorithm)

Structure Containing Set	Allocation	Alignment
Unpacked array	* Minimum #	Byte
Unpacked record	* Minimum #	Byte
Packed array	* Minimum #	Byte
Packed record	* Minimum #	Bit

* Minimum number of bits required to represent every member of the set.

Enumerations and Subranges

HP Pascal allocates and aligns variables of enumeration and subrange types the same way. An enumeration of n elements and the subrange $0..n\text{-}1$ are equivalent. The allocation and alignment are based on the values of the subrange or the ordinal value of the enumeration.

Example

```
TYPE
   enum_type = (red,blue,yellow);  {enumeration of 3 elements}
   subr_type = 0..2;               {subrange 0..(3-1)}

VAR
   enum_var : enum_type;
   subr_var : subr_type;
```

The compiler allocates and aligns the variables `enum_var` and `subr_var` the same way.

The allocation and alignment of an enumeration or subrange variable depends on whether it is:

- Unpacked.
- An element of a packed array.
- A field of a packed record.
- In a crunched structure.

Unpacked Enumeration or Unsigned Subranges

Table 5-7 shows how the HP Pascal packing algorithm allocates and aligns unpacked enumeration or unsigned subrange variables.

Table 5-7.
Allocation and Alignment of Unpacked Enumeration or Unsigned Subrange Variables
(HP Pascal Packing Algorithm)

Values in Enumeration or Subrange	Allocation	Alignment
0..255	1 byte	byte
256..65535	2 bytes	2-byte
65536..maxint	4 bytes	4-byte

An unpacked, signed subrange is always allocated four bytes.

Example

The value zero is always included in the subrange when the minimum number of bits is calculated.

```
TYPE
   enum_type  = (red,blue,yellow);  {3 elements}
   subr_type1 = 1..300;             {Including zero, 2 bytes}
   subr_type2 = 1..66000;           {Including zero, 4 bytes}
   subr_type3 = 100000..100010;     {Including zero, 4 bytes}
   subr_type4 = -1..200;            {4 bytes}

VAR
   enum_var  : enum_type;   {Allocated 1 byte, byte-aligned}
   subr_var1 : subr_type1;  {Allocated 2 bytes, 2-byte-aligned}
   subr_var2 : subr_type2;  {Allocated 4 bytes, 4-byte-aligned}
   subr_var4 : subr_type4;  {Allocated 4 bytes, 4-byte-aligned}

   unpacked_array : ARRAY [1..3] OF enum_type;  {Each element is
                                                 allocated one byte
                                                 and is byte-aligned}

   unpacked_record : RECORD
                        f1 : subr_type1;  {Allocated 2 bytes,
                                           2-byte-aligned}
                        f2 : subr_type2;  {Allocated 4 bytes,
                                           4-byte-aligned}
                     END;
```

Packed Array Elements of Enumeration or Subrange Types

A packed enumeration or subrange variable *requires* the minimum number of bits needed to represent its values in a record. It is bit-aligned.

If the enumeration or subrange variable belongs to a packed array, the HP Pascal packing algorithm allocates it the smallest power of two bits that is greater than or equal to the number of bits it requires, and aligns it on that boundary.

Table 5-8 shows the relationship between the number of bits that a packed array element of an enumeration- or subrange-type array requires, the number of bits that the HP Pascal packing algorithm allocates to it, and its alignment.

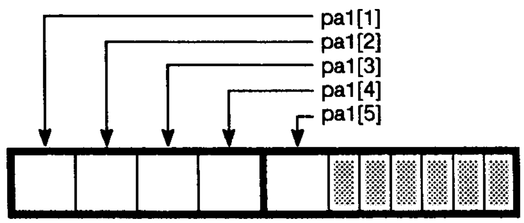
Table 5-8.
Allocation and Alignment of Packed Array Elements of Enumeration or Subrange Type
(HP Pascal Packing Algorithm)

Required Number of Bits Per Element	Number of Bits Allocated Per Element	Alignment
1	1	Bit
2	2	2-bit
3 or 4	4	4-bit
5 to 8	8 (1 byte)	Byte
9 to 16	16 (2 bytes)	2-byte
17 to 32	32 (4 bytes)	4-byte

Example

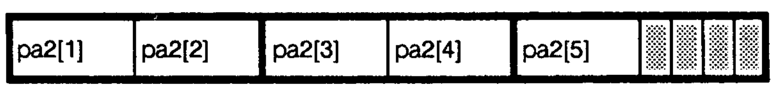
```
TYPE
   direction = (north,south,east,west);
   day = (sun,mon,tues,wed,thurs,fri,sat);

VAR
   pa1 = PACKED ARRAY [1..5] OF direction;
   pa2 = PACKED ARRAY [1..5] OF day;
```

Each element of the array *pa1* requires two bits. Two is a power of two, so each element is allocated two bits. The entire array occupies 10 bits. It is allocated two bytes:



LG200173_018

Each element of the array *pa2* requires three bits. The smallest power of two that is greater than or equal to three is four, so each element is allocated four bits. The entire array occupies 20 bits. It is allocated three bytes:



LG200173_019

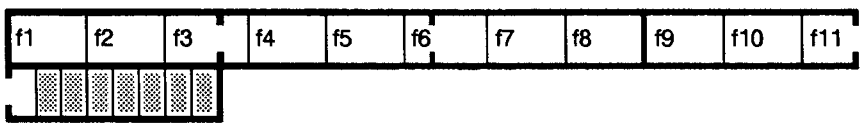
Packed Record Elements of Enumeration or Subrange Types

If the variable belongs to a packed record, the HP Pascal packing algorithm allocates it as many bits as it requires, and bit-aligns it.

Example

```
TYPE
   day = (sun,mon,tues,wed,thurs,fri,sat);

VAR
   r : PACKED RECORD
          f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11 : day;
       END;
```

Each field of the record *r* requires three bits. The entire record occupies 33 bits. It is allocated five bytes:



LG200173_020

Note Subranges can cross 4-byte boundaries, but code is less efficient when they do.

Packed records (such as those above) are byte-aligned. Code is more efficient when their alignment is specified with the ALIGNMENT compiler option.

Files

When your program declares a file, the compiler allocates space for the file control block and the file buffer variable. The amount of space allocated to each is fixed by the packing algorithm. The file is 8-byte-aligned.

Table 5-9 shows how the HP Pascal packing algorithm allocates file components for textfiles and nontextfiles.

Table 5-9. Allocation of File Components (HP Pascal Packing Algorithm)

File Component	Textfile	Nontextfile
`Control block`	`324 bytes`	`320 bytes`
`Buffer variable`	`254 bytes`	Size of component type

Sometimes you can reduce file buffer size or increase file operation speed by the way you declare a file. Compare the following file definitions, their buffer sizes, and how you can write 100 integers to them.

Declaration	Buffer Size	How to Write 100 Integers to the File

`VAR`
`f : FILE OF integer;`
	4 bytes	`FOR i:=1 TO 100 DO write(f,i);`
		
		`(100 calls to` *put*`)`

`VAR`
`f : FILE OF`
`ARRAY [1..100]`
`OF integer;`
	400 bytes	`FOR i:=1 TO 100 DO f^[i]:=i;`
		put`(f);`
		
		`(One call to` *put*`)`

Sets

The HP Pascal packing algorithm allocates sets in units called *set chunks*. Set chunk size depends on the number of bits required to represent the set and whether the set is unpacked, packed, or crunched.

The number of bits required to represent the set is determined by the formula:

```
bits_required_for_set = ord(largest_value_in_set) -
                        ord(smallest_value_in_set) + 1
```

Table 5-10 shows how the HP Pascal packing algorithm determines set chunk size.

Table 5-10. How Set Chunk Size Is Determined (HP Pascal Packing Algorithm)

Number of Bits Required To Represent Set	Set Chunk Size		
	Set is not PACKED	Set is PACKED	Set is CRUNCHED
1 to 8	32 bits	8 bits	1 bit
9 to 16	32 bits	16 bits	1 bit
17 or more	32 bits	32 bits	1 bit

The number of set chunks allocated to a set depends on its type. For the types Boolean, char, enumeration, and integer, the formula for the number of set chunks is:

number_of_set_chunks = *ceil(bits_required_for_set/set_chunk_size)*

(where *ceil(x)* means the integer closest to x that is greater than or equal to x).

Table 5-11 gives the values for *bits_required_for_set* and *number_of_set_chunks* for Boolean, char, and integer base types. For enumerated sets, *bits_required_for_set* is the number of elements in the set, and you must use the formula to determine *number_of_set_chunks*.

Table 5-11. Bit and Set Chunk Requirements for Boolean, Char, and Integer Types (HP Pascal Packing Algorithm)

Base Type	*bits_required_for_set*	*number_of_set_chunks*
Boolean	2	1
Char	256	8
Integer †	256 (by default) *	8

† Same for bit16, bit32, bit52, shortint, and longint.

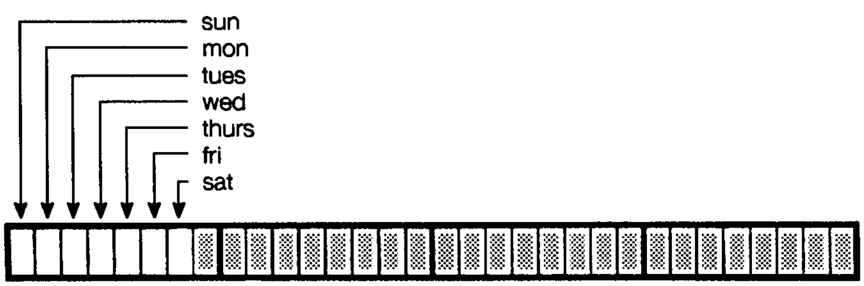
* Integers outside the range 0..255 cannot belong to the set.

Example 1

```
VAR
   days = SET OF (sun,mon,tues,wed,thurs,fri,sat);
   months = PACKED SET OF (ja,f,mr,ap,ma,jn,jl,au,s,o,n,d);

   set_33 = SET OF (e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,
                    e12,e13,e14,e15,e16,e17,e18,e19,e20,e21,e22,
                    e23,e24,e25,e26,e27,e28,e29,e30,e31,e32,e33);

   p_set_33 = PACKED SET OF (e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,
                    e12,e13,e14,e15,e16,e17,e18,e19,e20,e21,e22,
                    e23,e24,e25,e26,e27,e28,e29,e30,e31,e32,e33);
```

The set `days` has seven elements and requires seven bits. Its set chunk size is four bytes (32 bits), so `days` is allocated one set chunk.

Each element is represented by one bit, like this:



LG200173_021

The set `months` has 12 elements and requires 12 bits. Its set chunk size is two bytes, so `months` is allocated one set chunk (ceil(12/(2*8))=1). Each element is represented by one bit.

Each of the sets `set_33` and `p_set_33` has 33 elements and requires 33 bits. The set chunk size is four bytes, so `set_33` is allocated two set chunks (ceil(33/(4*8))=2). Each element is represented by one bit.

If the type is a subrange, the formula for the number of set chunks is:

number_of_set_chunks = (*upper_bound_set_chunk_number* - *lower_bound_set_chunk_number*) + 1

The upper bound of the subrange determines *upper_bound_set_chunk_number*, and the lower bound determines *lower_bound_set_chunk_number*. The formula is:

set_chunk_number = *floor(bound*/*set_chunk_size)*

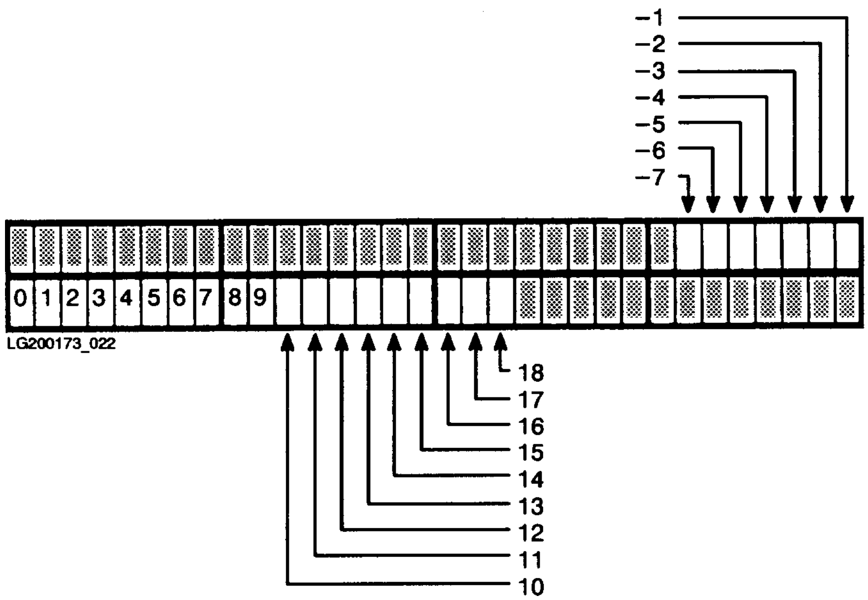
(where *floor(x)* means the integer closest to x that is less than or equal to x).

Example 2

```
VAR
   s : SET OF -7..18;
```

The set *s* is unpacked, so it has a 32-bit set chunk (see Table 5-10). The upper bound of the subrange is 18, so *upper_bound_set_chunk_number* is zero (floor(18/32=0)). The lower bound of the subrange is -7, so *lower_bound_set_chunk_number* is -1 (floor(-7/32)=-1). The set *s* is allocated two set chunks ((0-(-1))+1=2).

Each set element is represented by one bit, like this:



To minimize storage space, avoid base types that are small subranges that overlap set chunk boundaries.

Example 3

```
VAR
   s1 : SET OF 31..32;
   s2 : PACKED SET OF 15..16;
```

The set `s1` takes two 32-bit set chunks, using 64 bits to represent a set that requires only two bits. The arithmetic is: (floor(32/32) - floor(31/32)) + 1 = (1-0) + 1 = 2.

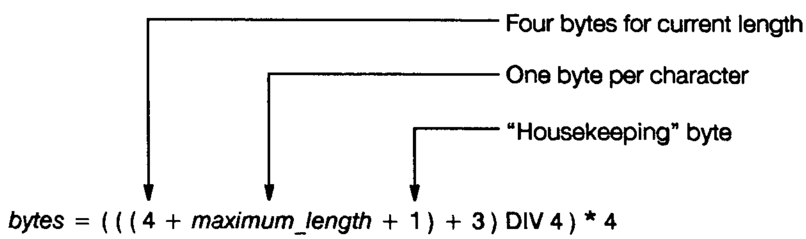
The PACKED set *s2* takes two 8-bit set chunks, using 16 bits to represent a set that requires only two bits. The arithmetic is: (floor(16/8) - floor(15/8)) + 1 = (2-1)+1 = 2.

Strings

A string is allocated four bytes for its current length (an integer), one byte per character, and one "housekeeping" byte. The number of characters is the string's declared maximum length. The "housekeeping" byte is only accessible to some of the standard string functions.

The HP Pascal packing algorithm aligns strings on 4-byte boundaries in all structures. Because the current length (an integer) is allocated four bytes, eight bytes is the smallest possible string allocation.

The formula for the number of bytes allocated to a string is:



LG200009_080

Example

```
VAR
   s1 : string[10];
   s2 : string[7];
```

The string `s1` takes 16 bytes:

```
(((4 + 10 + 1) + 3) DIV 4) * 4 =
                (18 DIV 4) * 4 =
                         4 * 4 = 16
```

The allocation is:

current length			
s1[1]	s1[2]	s1[3]	s1[4]
s1[5]	s1[6]	s1[7]	s1[8]
s1[9]	s1[10]	housekeeping	

LG200173_048

The string s2 takes 12 bytes:

```
(((4 + 7 + 1) + 3) DIV 4) * 4 =
               (15 DIV 4) * 4 =
                        3 * 4 = 12
```

The allocation is:

current length			
s2[1]	s2[2]	s2[3]	s2[4]
s2[5]	s2[6]	s2[7]	housekeeping

LG200173_024

6

Dynamic Variables

A *dynamic variable* is allocated during program execution. In contrast, a *global, module,* or *local* variable is allocated when the block containing its declaration is activated.

Table 6-1 shows the differences between dynamic and static variables.

Table 6-1. Dynamic versus Static Variables

	Dynamic Variable	**Global or Module Variable**	**Local Variable**
Declared?	No	Yes	Yes
Referenced by	Pointer (which is declared).	Name	Name
Allocated	During execution, with the function *new*.	Before compilation unit executes.	Upon entering procedure or function that declares it.
Stored on the	Heap	Static area	Stack
Deallocated	During execution, with the procedure *dispose* or *release*.	After program has executed.	After exiting the procedure or function that declares it.

This chapter explains:

- Pointer types peculiar to HP Pascal (*globalanyptr, anyptr*, and *localanyptr*).
- HP Pascal procedures *new* and *dispose,* which allocate and deallocate dynamic variables.
- HP Pascal procedures *mark* and *release,* which allow an HP Pascal program to deallocate a region of the heap that it no longer needs.
- Intrinsic procedures *p_getheap* and *p_rtnheap*, which allow a program written in any language that runs on the operating system to allocate and deallocate a region of the HP Pascal heap.

GLOBALANYPTR Variables

The pointer type *globalanyptr* is assignment compatible with every pointer type and the value *nil*. *Anyptr* is another name for the same type, provided for compatibility with older Pascals. This manual uses the term *globalanyptr* exclusively, but *anyptr* can be substituted wherever it appears.

A variable of type *globalanyptr* is not bound to a specific pointer type. You can assign it any pointer-type value, or compare it to any pointer-type value with the operator = or <>, but you cannot dereference it.

Because a *globalanyptr* variable can be assigned any pointer-type value, the compiler allocates it 64 bits. If your program does not use extended address pointers, you can save space by substituting *localanyptr* for *globalanyptr*.

Your program uses extended address pointers if it declares a type or variable with the EXTNADDR compiler option. Refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for detailed information on compiler options.

Example

This program works the same way and takes the same amount of space if you substitute *anyptr* for any or every occurrence of *globalanyptr*. This would be true even if the program had extended address pointers.

Since the program does not have extended address pointers, it works the same way if you substitute *localanyptr* for any or every occurrence of *globalanyptr*—but it takes less space. (Compare this program with the one in the section "LOCALANYPTR Variables".)

```
PROGRAM prog (input);
TYPE
   iptr = ^integer;
   rec  = RECORD
            f1, f2 : real;
          END;
   rptr = ^rec;
VAR
      v1, d1 : iptr;
      v2, d2 : rptr;
    anyv : globalanyptr;
       b : Boolean;
BEGIN
   {Initialize v1 and v2}
   new(v1);
   new(v2);
   v1^ := 0;
   WITH v2^ DO BEGIN
      f1 := 0;
      f2 := 0;
   END;

   {Set anyv to v1 or v2, depending on b}
   read(b);
   IF b THEN anyv := v1 ELSE anyv := v2;

   {You cannot dereference anyv, because it is a globalanyptr.
    This is how you can access its data:}
   IF anyv = v1 THEN BEGIN
      d1 := anyv;
      d1^ := d1^ + 1;
   END
   ELSE BEGIN
      d2 := anyv;
      WITH d2^ DO BEGIN
         f1 := 34.6;
         f2 := 91.2;
      END;
   END;
END.
```

LOCALANYPTR Variables

The pointer type *localanyptr* is similar to the type *globalanyptr* (or *anyptr*) in that it is assignment compatible with every pointer type and the value *nil*.

A *localanyptr* variable differs from a *globalanyptr* variable in that the compiler allocates it 32 bits instead of 64 bits. If your program does not use extended address pointers, you can save space by using *localanyptr* instead of *globalanyptr*.

Like a *globalanyptr* variable, a *localanyptr* variable is not bound to a specific pointer type. You can assign it any pointer-type value, but you can not assign it an extended address pointer that cannot be converted to a 32-bit value.

You can compare a *localanyptr* variable to any pointer-type value (even one that you cannot assign to it) with the operator = or <>.

You cannot dereference a *localanyptr.*

Example

This program is the same as the one in the section "GLOBALANYPTR Variables", except that *localanyptr* replaces every occurrence of *globalanyptr*. The two programs work the same way, but this one takes less space.

```
PROGRAM prog (input);

TYPE
   iptr = ^integer;
   rec = RECORD
           f1, f2 : real;
         END;
   rptr = ^rec;
VAR
      v1,
      d1 : iptr;
      v2,
      d2 : rptr;
    anyv : localanyptr;
       b : Boolean;

BEGIN
   {Initialize v1 and v2}
   new(v1);
   new(v2);
   v1^ := 0;
   WITH v2^ DO BEGIN
      f1 := 0;
      f2 := 0;
   END;

   {Set anyv to v1 or v2, depending on b}
   read(b);
   IF b THEN anyv := v1 ELSE anyv := v2;

   {You cannot dereference anyv, because it is a localanyptr.
    This is how you can access its data:}
   IF anyv = v1 THEN BEGIN
      d1 := anyv;
      d1^ := d1^ + 1;
   END
   ELSE BEGIN
      d2 := anyv;
      WITH d2^ DO BEGIN
         f1 := 34.6;
         f2 := 91.2;
      END;
   END;
END.
```

New Procedure

The predefined procedure *new* takes a pointer variable as a parameter, allocates a variable of the type that the pointer references, and "points" the pointer at the new variable (that is, *new* assigns the address of the new variable to the pointer). The program can then access the new variable by dereferencing the pointer.

Example 1

```
PROGRAM prog;

TYPE
   iptr = ^integer;
   cptr = ^char;
   rptr = ^real;

VAR
   ivar : iptr;  {pointer to a dynamic integer variable}
   cvar : cptr;  {pointer to a dynamic character variable}
   rvar : rptr;  {pointer to a dynamic real variable}

BEGIN
   new(ivar);  {allocate new integer variable on heap}
   new(cvar);  {allocate new character variable on heap}
   new(rvar);  {allocate new real variable on heap}

   ivar^ := 375;  {assign value to new integer variable}
   cvar^ := 'c';  {assign value to new character variable}
   rvar^ := 3.7;  {assign value to new real variable}
END.
```

The new variable is allocated space on the heap. A run-time error occurs if the heap cannot accommodate the variable.

If the new variable is a record with variant fields, you can specify the variant that you want with a tag. The tag only tells the *new* procedure how much space to allocate; it does not cause the *new* procedure to assign the value of the tag to the new variable's tag field.

Example 2

```
PROGRAM prog;

TYPE
   marital_status = (single, married);

   rec = RECORD
            lname,
            fname : string[30];
            kids : 1..20;
```

(*Example is continued on next page.*)

```
            CASE mstat : marital_status OF
               single : (divorced,
                         widowed,
                         engaged : Boolean);
               married : (how_many_times: 1..10;
                          how_long_this_time : 1..100);
         END;

      recptr = ^rec;

   VAR
      person1,
      person2,
      person3 : recptr;

   BEGIN
      new(person1,single);

      WITH person1^ DO BEGIN
         lname := 'Doe';
         fname := 'John';
         kids := 0;
         mstat := single;  {New does not make this assignment}
         divorced := FALSE;
         widowed := FALSE;
         engaged := FALSE;
      END;

      new(person2,married);

      WITH person2^ DO BEGIN
         lname := 'Smith';
         fname := 'Jane';
         kids := 3;
         mstat := married;  {New does not make this assignment}
         how_many_times := 1;
         how_long_this_time := 9;
      END;

      new(person3);
   END.
```

The new record variable `person1^` has space for the fixed fields `lname`, `fname`, `kids`, and `mstat`, and for the `single` variant fields `divorced`, `widowed`, and `engaged`.

The new record variable `person2^` has space for the same fixed fields, and for the `married` variant fields `how_many_times` and `how_long_this_time`.

If the new variable is a record with nested variant fields, you can specify a tag for each variant. If you do, you must specify them in the order that they are declared, and you cannot leave gaps in the sequence.

Example 3

In this program, the declaration order of the tag fields is obviously `t1, t2` or `t1, t3`.

```
PROGRAM prog;

TYPE

   r = RECORD
          f1 : integer;
          CASE t1 : (a,b) OF
             a : (arec : RECORD
                            i : integer;
                            CASE t2 : (c,d) OF
                               c : (j : integer);
                               d : (k : real);
                         END {arec}
                 );
             b : (brec : RECORD
                            CASE t3 : (e,f) OF
                               e : (l : real);
                               f : (m : char);
                         END {brec}
                 );
        END; {r}

   rptr = ^r;

VAR
   v : rptr;

BEGIN
   new(v);
   new(v,a);
   new(v,a,c);
   new(v,a,d);
   new(v,,d);   {illegal -- must specify a}
   new(v,d);    {illegal -- must specify a}
   new(v,b);
   new(v,b,e);
   new(v,e,b);  {illegal -- tags are not in order of declaration}
   new(v,b,f);
   new(v,a,f);  {illegal -- with variant a, variant f is impossible}
END.
```

Example 4

This program is semantically equivalent to the program in the immediately preceding example (Example 3), and the declaration order of the tag fields is the same.

```
PROGRAM prog;

TYPE

   arectype = RECORD
                 i : integer;
                 CASE t2 : (c,d) OF
                    c : (j : integer);
                    d : (k : real);
              END;

   brectype = RECORD
                 CASE t3 : (e,f) OF
                    e : (l : real);
                    f : (m : char);
              END;

   r = RECORD
          f1 : integer;
          CASE t1 : (a,b) OF
             a : (arec : arectype);
             b : (brec : brectype);
       END;

   rptr = ^r;

VAR
   v : rptr;

BEGIN
   new(v);
   new(v,a);
   new(v,a,c);
   new(v,a,d);
   new(v,,d);   {illegal -- must specify a}
   new(v,d);    {illegal -- must specify a}
   new(v,b);
   new(v,b,e);
   new(v,e,b);  {illegal -- tags are not in order of declaration}
   new(v,b,f);
   new(v,a,f);  {illegal -- with variant a, variant f is impossible}
END.
```

You do not have to specify tag fields. If you omit them, *new* allocates enough space for the largest possible variant, wherever there are variants. This allocation is the default allocation for variables of the particular record type.

If you use tags to specify smaller variants, *new* allocates less than the default allocation to the new variable. The advantage to using tags is that you save space. The disadvantage is that the new variable cannot appear in an assignment statement, or as an actual parameter. (Assignment statements and formal parameters use the default allocation.) It is legal for the fields of the new variable to appear as actual parameters, and to be used in a field by field assignment.

Example 5

```
PROGRAM prog;
TYPE
   rec = RECORD
            CASE t : (a,b) OF
               a : (a1,a2 : integer);
               b : (b1,b2,b3,b4,b5,b6 : integer);
         END;
   recptr = ^rec;
VAR
   small,
   small2,
   large,
   default : recptr;

PROCEDURE p (r : rec); EXTERNAL;
BEGIN
   new(small,a);  {allocates only enough space for smaller variant, a}
   new(small2,a); {allocates only enough space for smaller variant, a}
   new(large,b);  {allocates enough space for larger variant, b}
   new(default);  {allocates enough space for larger variant by default}

   WITH small^ DO BEGIN
      t := a;
      a1 := 350;
      a2 := 609;
   END;

   WITH large^ DO BEGIN
      t := b;
      b1 := 350;
      b2 := 609;
   END;
```

(*Example is continued on next page.*)

```
    default^.t := a;
    default^ := small^;  {illegal}
    default^.t := b;
    default^ := large^;  {illegal}
    small2^ := small^ {still illegal even though the spaces are allocated }
                      {using the same tag   }
    small2^.a1 := small^.a1  {legal}
    small2^.a2 := small^.a2  {legal}

    p(small^);  {illegal}
    p(large^);  {illegal}
    p(default^);  {legal}
 END.
```

The pointer parameter of *new* can belong to a PACKED structure.

Example 6

```
 PROGRAM prog;

 TYPE
    ptr = ^integer;
    pa = PACKED ARRAY [1..10] OF ptr;
    pr = PACKED RECORD
            f1,f2 : ptr;
         END;

 VAR
    v1 : pa;
    v2 : pr;

 BEGIN
    new(v1[5]);
    new(v2.f1);
 END.
```

A pointer created by *new* can be compared to another pointer for equality or inequality only. This is also true of a pointer created by *mark*. For more information on relational operators, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

Dispose Procedure

The predefined procedure *dispose* takes a pointer variable as a parameter and deallocates the dynamic variable that it references. When the variable is deallocated, it is inaccessible, and the pointer is undefined. Files in the deallocated space are closed.

The procedure *new* can only reallocate the space that *dispose* has deallocated if the program contains the compiler option HEAP_DISPOSE. For more information, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

It is an error to call *dispose* with a pointer that is:

- Undefined.
- Nil.
- The dynamic variable referenced by a pointer that is the actual parameter, passed by reference, of a currently executing routine.
- The dynamic variable referenced by a pointer that is in the record variable list of a currently executing WITH statement.

Example 1

```
PROGRAM prog;
TYPE
   rec = RECORD
            f1,f2,f3 : integer;
         END;

   recptr = ^rec;
VAR
   v1,v2,v3,v4,v5 : recptr;

PROCEDURE p (VAR x : rec);
BEGIN
   dispose(v4);  {illegal -- disposes x's actual parameter}

END;

PROCEDURE q;
BEGIN
   dispose(v4);  {illegal -- v4^ is in the record variable
                  list of an active WITH statement}
END;
```

(*Example is continued on next page.*)

```
PROCEDURE r (VAR z : recptr);
   PROCEDURE s;
   BEGIN
      dispose(v4);  {illegal -- v4^ is the actual parameter for z}
   END;
BEGIN
   s;
END;
BEGIN
   new(v1);
   WITH v1^ DO BEGIN
      f1 := 0;
      f2 := 0;
      f3 := 0;
   END;
   dispose(v1);
   dispose(v1);  {illegal -- v1 is undefined}

   new(v2);
   dispose(v2);

   new(v3);
   v3 := nil;
   dispose(v3);  {illegal -- v3 is nil}

   new(v4);
   p(v4^);

   new(v4);
   r(v4);    {s (within r) disposes r's actual parameter v4,
             which is illegal}
   new(v4);
   new(v5);
   WITH v4^,v5^ DO BEGIN
      f1 := 1;
      f2 := 2;
      f3 := 3;
      q;  {illegal -- q disposes v4 while the WITH statement
                      whose record variable list it is in
                      is active}

      dispose(v5);  {illegal -- v5 is in the record variable list
                                of an active WITH statement}
   END;
END.
```

If you specify tags when you allocate a variable with *new*, you must specify the same tags in the same order when you deallocate the variable with *dispose*.

Example 2

```
PROGRAM prog;

TYPE
   rec = RECORD
            CASE t1 : (a,b) OF
               a : (a1,a2 : integer);
               b : (b1 : RECORD
                           CASE t2 : (c,d) OF
                              c : (c1 : char);
                              d : (d1,d2 : real);
                        END
                   );
         END;

   recptr = ^rec;

VAR
   v1,v2,v3,v4,v5 : recptr;

BEGIN
   new(v1);
   new(v2,a);
   new(v3,b);
   new(v4,b,c);
   new(v5,b,d);

   dispose(v1);
   dispose(v2,a);
   dispose(v3,b);
   dispose(v4,b,c);
   dispose(v5,b,d);

   new(v1);
   new(v2,a);
   new(v3,b);
   new(v4,b,c);
   new(v5,b,d);

   dispose(v1,a);    {illegal -- a not specified on new}
   dispose(v2,b);    {illegal -- b not specified on new}
   dispose(v3);      {illegal -- b specified on new, but not here}
   dispose(v4,b);    {illegal -- c specified on new, but not here}
   dispose(v5,d,b);  {illegal -- b and d are in the wrong order}
END.
```

Mark and Release Procedures

The predefined procedure *mark* takes a pointer variable *p* as a parameter, marks the state of the heap, and sets the value of *p* to specify that state.

The pointer variable *p* is called a *mark* (once a pointer variable becomes a mark, you cannot dereference it). You can allocate heap space beyond the mark, and then deallocate that space with the predefined procedure *release*.

The predefined procedure *release* takes a mark pointer variable as a parameter and deallocates the heap space that was dynamically allocated after the mark was set. Variables in that space become inaccessible. Files in that space are closed. After *release* executes, the mark pointer variable is undefined. The procedure *new* can reallocate the released space (even if the program does not contain the compiler option HEAP_DISPOSE).

Example 1

```
PROGRAM prog;

TYPE
   ftype = FILE OF integer;
   ptype1 = ^ftype;
   ptype2 = ^integer;

VAR
   fptr : ptype1;
   iptr1,
   iptr2,
   m,
   iptr3,
   iptr4: ptype2;

BEGIN
   new(iptr1);     {Allocate heap space to iptr1^}
   new(iptr2);     {Allocate heap space to iptr2^}

   iptr1^ := 0;
   iptr2^ := 0;

   mark(m);        {Mark the heap with m}

   new(iptr3);     {Allocate heap space to iptr3^}
   new(iptr4);     {Allocate heap space to iptr4^}
   new(fptr);      {Allocate heap space to fptr^, a file}

   iptr3^ := 0;
   iptr4^ := 0;
   reset(fptr^);   {Open fptr^}

   release(m);     {Close fptr^, deallocating heap after m}

   iptr1^ := 1;
   iptr2^ := 2;
   iptr3^ := 3;     {illegal -- iptr3^ was deallocated}
   iptr4^ := 4;     {illegal -- iptr4^ was deallocated}
   write(fptr^,5);  {illegal -- iptr5^ was deallocated}
   m^ := 0;         {illegal -- cannot assign value to mark pointer}
END.
```

The parameter of *mark* (the mark) can be any pointer variable.

The parameter of *release* must be a mark—a pointer variable whose current value was assigned by the *mark* procedure. It is an error to call *release* with a pointer whose current value was not assigned by the *mark* procedure.

Example 2

```
PROGRAM prog;

TYPE
   ptr1 = ^integer;
   ptr2 = ^real;
   ptr3 = ^char;
   ptr4 = ^ptr3;

VAR
   m1 : ptr1;
   m2 : ptr2;
   m3 : ptr3;
   m4 : ptr4;
   m6 : ptr1;

    r : RECORD
           i : integer;
           m5 : ptr1;
        END;

BEGIN
   mark(m1);
   mark(m2);
   mark(m3);

   new(m4);    {m4^ is of type ptr3}
   mark(m4^);

   mark(r.m5);

   new(m6);
   release(m6);  {illegal -- current value of m6 was assigned by new}
END.
```

If you set several marks, and release one of them, those set after it are also released.

Example 3

```
PROGRAM prog;

TYPE
   ptr = ^integer;

VAR
   m1, m2,
   i1, i2, i3,
   j1, j2, j3,
   k1, k2, k3 : ptr;

BEGIN
   new(i1);
   new(i2);
   new(i3);

   mark(m1);

   new(j1);
   new(j2);
   new(j3);

   mark(m2);

   new(k1);
   new(k2);
   new(k3);

   release(m1);  {deallocates j1,j2,j3,k1,k2,k3; releases m1 and m2}
   release(m2);  {illegal -- m2 is undefined because it was released
                            with m1}
END.
```

P_getheap and P_rtnheap Procedures

The procedures *p_getheap* and *p_rtnheap* are intrinsics in the Pascal run-time library. Any program that runs on the operating system can call them, regardless of the language in which it is written. (For more information on intrinsics, Chapter 10).

The procedure *p_getheap* tries to allocate a region of heap space of a specified size and alignment. If it succeeds, it "points" its VAR pointer parameter at the first element of the region and assigns its VAR Boolean parameter the value *true*. If it fails, it assigns its VAR Boolean parameter the value *false*.

Syntax

```
p_getheap (VAR regptr     : localanyptr;
               regsize    : integer;
               alignment  : integer;
           VAR ok         : Boolean);
```

Parameters

regptr	If *p_getheap* can allocate the region of heap space, it "points" *regptr* at the first element of the region (that is, *p_getheap* assigns the address of the first element of the region to *regptr*).
regsize	The size of the region of heap space, in bytes.
alignment	See table below.
ok	If *p_getheap* can allocate the region of heap space, it assigns *ok* the value *true*; if not, it assigns *ok* the value *false*.

alignment:

Integer:	Specifies the region of heap space to be:
1	Byte-aligned
2	Halfword-aligned
4	Word-aligned
8	Double-word-aligned
16	16-byte aligned
32	32-byte aligned
64	64-byte aligned
2048	Page-aligned

The procedure *p_rtnheap* tries to deallocate a region of heap space that the *p_getheap* procedure allocated. If it succeeds, it assigns its VAR Boolean parameter the value *true*. If it fails, it assigns its VAR Boolean parameter the value *false*. *P_rtnheap* does not close files residing in the region allocated by *p_getheap*.

Syntax

```
p_rtnheap (VAR regptr    : localanyptr;
               regsize   : integer;
               alignment : integer;
           VAR ok        : Boolean);
```

Parameters

regptr — A pointer whose current value was assigned to it by the procedure *p_getheap*.

regsize — The size in bytes of the region of heap space that *p_getheap* assigned to *regptr*.

alignment — The number that specified the alignment of the region of heap space that *p_getheap* assigned to *regptr*.

ok — If *p_rtnheap* can deallocate the region of heap space, it assigns *ok* the value *true*; if not, it assigns *ok* the value *false*.

Example 1

```
$STANDARD_LEVEL 'HP_MODCAL'$
PROGRAM prog;

TYPE
   intpointer = ^integer;

VAR
   b    : Boolean;
   i    : integer;
   ptr1,
   ptr2 : intpointer;

PROCEDURE p_getheap (VAR regptr    : intpointer;
                         regsize   : integer;
                         alignment : integer;
                     VAR ok        : Boolean); EXTERNAL;

PROCEDURE p_rtnheap (VAR regptr    : intpointer;
                         regsize   : integer;
                         alignment : integer;
                     VAR ok        : Boolean); EXTERNAL;

BEGIN
   p_getheap(ptr1,40,4,b);   {allocate a 40-byte region}

   ptr2 := ptr1;             {save ptr1 for later call to p_rtnheap}

   FOR i := 1 TO 10 DO BEGIN
      ptr2^ := i;
      ptr2 := addtopointer(ptr2,4);
   END;

   p_rtnheap(ptr1,40,4,b);   {deallocate the 40-byte region}

   p_getheap(ptr1,50,2,b);
   p_rtnheap(ptr1,20,2,b);   {illegal -- 20 must be 50}

   p_getheap(ptr1,16,8,b);
   p_rtnheap(ptr1,16,1,b);   {illegal -- 1 must be 8}
END.
```

The procedures *p_getheap* and *p_rtnheap* are independent from the procedures *mark, release, new,* and *dispose*.

Example 2

```
$STANDARD_LEVEL 'HP_MODCAL'$
PROGRAM prog;
VAR
   i    : integer;
   b    : Boolean;
   p1,p2,p3,
   ptr1, ptr2, ptr3 : ^integer;

PROCEDURE p_getheap; INTRINSIC;
PROCEDURE p_rtnheap; INTRINSIC;

BEGIN
   p_getheap(ptr1,28,4,b);   {allocate a 28-byte region}

   ptr3 := ptr1;             {assign values in the 28-byte region}
   FOR i := 1 TO 7 DO BEGIN
      ptr3^ := i;
      ptr3 := addtopointer(ptr3,4);
   END;
   ptr3 := ptr1;

   mark(ptr2);               {mark the heap}

   new(p1);                  {allocate p1, p2, and p3}
   new(p2);
   new(p3);

   p_rtnheap(ptr1,28,4,b);   {deallocate the 28-byte region}

   ptr3^ := 0;               {illegal -- p_rtnheap deallocated ptr3^}

   p1^ := 1;                 {p_rtnheap did not deallocate p1, p2, or p3;}
   p2^ := 2;                 {they are still accessible}
   p3^ := 3;

   p_getheap(ptr1,4,4,b);    {allocate a 4-byte region}
```

(*Example continued on next page.*)

```
   release(ptr2);

   ptr1^ := 0;                {The 4-byte region was not
                                deallocated, and the values
                                in it are still accessible}

   p1^ := p2^ + p3^;  {illegal -- p1, p2, and p3 were deallocated}
END.
```

Getheap and Rtnheap Procedures

The procedures *getheap* and *rtnheap* are intrinsics in the Pascal run-time library. They are provided only for compatibility with existing source code that was written for the MPE V operating system and only exists on MPE/iX. If you are writing a new program, use the predefined procedures *p_getheap* and *p_rtnheap* instead.

The procedure *getheap* allocates a region of heap space, and the procedure *rtnheap* deallocates the region.

Syntax

```
getheap (VAR regptr  : localanyptr;
         VAR regsize : shortint;
         VAR ok      : shortint);

rtnheap (VAR regptr  : localanyptr;
             regsize : shortint;
         VAR ok      : shortint);
```

7

Parameters

This chapter explains:

- The differences between value and reference parameters.
- ANYVAR and READONLY reference parameters (which are HP Pascal system programming extensions).
- Conformant array parameters.
- Routines (procedures and functions) as parameters.
- Congruent parameter lists.
- Hidden parameters (which affect debugging and interfacing with external non-Pascal routines).

Note This chapter is intended for system software developers who already understand the systems for which they are programming. Its purpose is to explain the HP Pascal features of which they must be aware. It does not attempt to teach systems programming.

Value versus Reference Parameters

The terms *value* and *reference* must be explained in terms of formal and actual parameters. A *formal parameter* is defined in a routine header. An *actual parameter* is passed in a call to a routine.

Example 1

```
PROGRAM prog;
VAR
   a : integer;

PROCEDURE p (f : integer);  {f is a formal parameter}
BEGIN
END;

BEGIN
  p(a);  {a is an actual parameter}
END;
```

A *value parameter* is passed by value; that is, the value of the actual parameter is passed to the routine and assigned to the formal parameter. If the routine changes the value of the formal parameter, it does not change the value of the actual parameter. An actual value parameter can be a constant, an expression, a variable, or a function result.

A *reference parameter* is passed by reference; that is, the address of the actual parameter is passed to the routine and associated with the formal parameter. If the routine changes the value of the formal parameter, it changes the value of the actual parameter. An actual reference parameter must be a *variable access* (a variable name or the name of a component of an unpacked structure).

HP Pascal without system programming extensions has one kind of reference parameter: VAR. For more information on VAR parameters, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

HP Pascal with system programming extensions has two additional kinds of reference parameters: ANYVAR and READONLY. An actual READONLY parameter can be a constant, an expression, or a function result.

Example 2

```
PROGRAM prog;
VAR
   a,b : integer;

PROCEDURE p (    x : integer;  {x is a value parameter}
             VAR y : integer); {y is a reference parameter}
BEGIN
   x := x+1;  {this does not change x's actual parameter}
   y := y+1;  {this does change y's actual parameter}

   writeln(x);  {this writes 41}
   writeln(y);  {this writes 61}
END;

BEGIN
   a := 40;
   b := 60;

   p(a,b);

   writeln(a);  {this writes 40}
   writeln(b);  {this writes 61}
END.
```

Table 7-1 compares the four kinds of formal parameters.

Table 7-1. Comparison of Kinds of Formal Parameters

Kind of Formal Parameters	STANDARD_LEVEL	Actual Parameter Can Be	Actual Parameter Is Passed By	Routine Can Modify	
				Parameter	Actual Parameter
Value	ANSI	Constant, expression variable, or function result	Value	Yes	No
Var	ANSI	Variable only	Reference	Yes	Yes
ANYVAR	HP_MODCAL	Variable only	Reference	Yes	Yes
READONLY	HP_MODCAL	Constant, expression, variable, or function result	Reference	No	No

ANYVAR Parameters

An ANYVAR parameter is similar to a VAR parameter in that its actual parameter is passed by reference and must be a variable access. If the routine changes the value of a formal ANYVAR parameter, it changes the value of the actual parameter.

An ANYVAR parameter differs from a VAR parameter in that its actual parameter can be of any type. HP Pascal treats the actual parameter as if it were of the data type of the formal ANYVAR parameter. This is implicit type coercion.

Example 1

```
$STANDARD_LEVEL 'HP_MODCAL'$
PROGRAM prog;

TYPE
   type1 = ARRAY [1..10] OF integer;
   type2 = ARRAY [1..20] OF integer;
   type3 = ARRAY [1..11] OF real;

VAR
   var1 : type1;
   var2 : type2;
   var3 : type3;

PROCEDURE p (   VAR parm1 : type1;
             ANYVAR parm2 : type2); EXTERNAL;
BEGIN
   p(var1,  {legal}
     var1); {legal}
   p(var2,  {illegal -- must be of type1}
     var2); {legal}
   p(var3,  {illegal -- must be of type1}
     var3); {legal}
END.
```

The formal VAR parameter `parm1` must have an actual parameter of type `type1`. The formal ANYVAR parameter `parm2` can have an actual parameter of any type.

The first call to procedure *p* passes the variable `var1` (a 10-element integer array) to `parm2` (a 20-element integer array). This is legal because `parm2` is an ANYVAR parameter; however, `parm2[11]` through `parm2[20]` are undefined. Accessing them causes unpredictable results.

The second call to `p` passes the variable `var2` to `parm2`. Both are 20-element integer arrays. The procedure `p` can access all 20 elements of `parm2`.

The third call to `p` passes the variable `var3` (an 11-element real array) to `parm2` (a 20-element integer array). Although this is legal, `p` must not try to access any of the nonexistent elements `parm2[12]` through `parm2[20]`. The procedure `p` treats the elements of `parm2` as if they were integers (although the elements of `var3` are real).

The implicit type coercion requires that the actual parameter be aligned on a boundary that is the same or larger than the boundary on which the formal parameter is aligned (for example, if the formal parameter is 2-byte-aligned, the actual parameter can be 2-byte-aligned or 4-byte-aligned, but it cannot be byte-aligned).

Example 2

```
PROGRAM prog;
VAR
   c : PACKED ARRAY [1..2] OF char;
   j : shortint;
   i : integer;

PROCEDURE show_anyvar_alignment
          (ANYVAR anyvar_parm : shortint);
          EXTERNAL;
BEGIN
   show_anyvar_alignment(c);  {illegal -- must be 2-byte-aligned}
   show_anyvar_alignment(j);  {legal}
   show_anyvar_alignment(i);  {legal -- references high-order 2 bytes}
END.
```

When HP Pascal passes an actual parameter to a formal ANYVAR parameter, it also passes a hidden parameter. The hidden parameter can be used to determine the size of the actual parameter. See "Hidden Parameters" for more information.

READONLY Parameters

A READONLY parameter is similar to a value parameter in that the routine cannot directly modify its actual parameter, which can be a constant, an expression, or a variable. READONLY differs from a value parameter in that the routine cannot modify the formal parameter: you cannot assign a value to the formal READONLY parameter, pass it to a VAR or ANYVAR parameter, or pass it to either of the predefined functions *addr*, *baddress*, or *waddress*.

A READONLY parameter is similar to a VAR or ANYVAR parameter in that its actual parameter is passed by reference. If the actual parameter is an expression or constant, a copy of its value is passed by reference.

Example

```
PROGRAM prog;
$STANDARD_LEVEL 'HP_MODCAL'$

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

CONST
   arrayconst = arraytype [10 OF 0];

VAR
   arrayvar : arraytype;

FUNCTION arrayfunc : arraytype; EXTERNAL;

PROCEDURE p (            valuep : arraytype;
                       VAR varp : arraytype;
             READONLY readonlyp : arraytype); EXTERNAL;

BEGIN
   p(arrayconst,   {value is passed}
     arrayconst,   {illegal -- must be a variable}
     arrayconst);  {address of copy of value is passed}

   p(arrayvar,     {value is passed}
     arrayvar,     {address is passed}
     arrayvar);    {address is passed}

   p(arrayfunc,    {value is passed}
     arrayfunc,    {illegal -- must be a variable}
     arrayfunc);   {address of copy of value is passed}
END.
```

The comments in the preceding program explain the differences in passing a constant (`arrayconst`), a variable (`arrayvar`), and an expression (a call to the function `arrayfunc`) to a value parameter (`valuep`), a VAR parameter (`varp`), and a READONLY parameter (`readonlyp`).

Conformant Array Parameters

A *conformant array parameter* is a formal array parameter defined by a *conformant array schema* (the syntax appears in the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*). Its actual parameter must be an array variable that conforms to the schema.

An array variable conforms to a conformant array schema if all of the following are true:

- The variable and the schema are both PACKED, or neither is PACKED.
- The index types of the variable and the schema are compatible (as defined in the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*).
- The bounds of the index type of the variable are within the bounds of the index type of the schema.
- The element types of the variable and the schema are the same, unless the element type of the schema is another schema. If the element type of the schema is another schema, the element type of the variable conforms to the other schema.

Example 1

```
TYPE
   itype = 0..20;
   jtype = 'a'..'z';
   ktype = 0..5;

VAR
   var1 : ARRAY [0..10] OF integer;


PROCEDURE p (yes  : ARRAY [lb1..ub1 : itype] OF integer;
             no1  : PACKED ARRAY [lb3..ub3 : itype] OF integer;
             no2  : ARRAY [lb4..ub4 : jtype] OF integer;
             no3  : ARRAY [lb5..ub5 : ktype] OF integer;
             no4  : ARRAY [lb6..ub6 : itype] OF real;
             no5  : ARRAY [lb7..ub7 : itype;
                           lb8..ub8 : itype] OF integer);
```

The array variable `var1` conforms to the schemas of the conformant array parameter `yes`. `Var1` and the schema of `yes` have the same element type, and 0..10 is within the bounds of `itype`.

The variable `var1` does not conform to the schemas of conformant array parameters `no1`, `no2`, `no3`, `no4`, and `no5`. The following table gives the reasons for nonconformance.

Parameter	Why *var1* Does Not Conform to Schema
`no1`	Schema is PACKED and *var1* is not PACKED.
`no2`	Index types of *var1* and schema are not compatible.
`no3`	Bounds of index type of *var1* are not within bounds of index type of schema.
`no4`	Element types of *var1* and schema are different.
`no5`	Schema specifies two dimensions, and *var* has only one dimension.

Like array declarations, schemas can specify dimensions in syntactically different but structurally equivalent ways.

Example 2

```
VAR
   var1 : ARRAY [3..5,1..10] OF integer;
   var2 : ARRAY [3..5] OF ARRAY [1..10] OF integer;

PROCEDURE p (yes1 : ARRAY [lb1..ub1 : itype] OF
                    ARRAY [lb2..ub2 : itype] OF integer;
             yes2 : ARRAY [lb3..lb3 : itype;
                           lb4..ub4 : itype] OF integer;
             no1  : ARRAY [lb5..ub5 : itype] OF integer;
             no2  : ARRAY [lb6..ub6 : itype;
                           lb7..ub7 : itype;
                           lb8..ub8 : itype] OF integer);
```

The declarations of the array variables `var1` and `var2` are structurally equivalent, as are the schemas of conformant array parameters `yes1` and `yes2`. Both `var1` and `var2` conform to the schemas of `yes1` and `yes2`. Neither `var1` nor `var2` conforms to the schema of `no1` or `no2`.

When a conformant array schema is a formal parameter, its bounds are also formal parameters. They are read-only parameters. The actual parameter for the formal conformant array schema is an array, and its bounds are the actual parameters of the formal bounds parameters.

Example 3

```
TYPE
   itype = 0..20;

VAR
   v : ARRAY [0..10] OF integer;

PROCEDURE p (x  : ARRAY [lb..ub : itype] OF integer);

BEGIN
   p(v);
END;
```

The conformant array schema `x` is a formal parameter, so its bounds, `lb` and `ub` are read-only formal parameters. The array `v` is the actual parameter for `x`. The lower bound of `v`, zero, is the actual parameter for `lb`. The upper bound of `v (10)` is the actual parameter for `ub`.

When HP Pascal passes an actual parameter to a formal conformant array parameter of more than one dimension, it also passes one hidden parameter for each inner dimension that is itself a conformant array. See "Hidden Parameters" for more information.

Routines as Parameters

A routine can be a parameter in two ways: it can be a routine parameter (a procedure or function parameter, as defined by ANSI Pascal), or it can be a routine that is passed as a parameter (as defined by the systems programming extensions of HP Pascal).

Table 7-2 differentiates between routine parameters and parameters of routine types.

Table 7-2. Routine Parameters versus Parameters of Routine Type

	Routine Parameter	**Parameter of Routine Type**
Availability	ANSI Pascal	System programming extensions.
Where Defined	Formal parameter list of routine.	Parameter is defined in formal parameter list of routine, but its type is defined first in a type declaration section.
Corresponding Actual Parameter	User-defined routine.	*addr* applied to user-defined routine, or variable of a routine type.
Referenced By	Name	*Fcall* or *call* routine.

Routine Parameters

Routine parameters (procedure or functions parameters) are parameters that are routines (procedures or functions, respectively). They are completely defined in the formal parameter lists of other routines, which reference them by name.

A formal function parameter is a function definition. Its actual parameter is the name of a user-defined function with a congruent parameter list and the same result type.

A formal procedure parameter is a procedure definition. Its actual parameter is the name of a user-defined procedure with a congruent parameter list.

Predefined routines cannot be passed to routine parameters.

Example

```
PROGRAM prog;
VAR
   s : char;
PROCEDURE p (PROCEDURE procparm1 (a,b : integer);
                                        {formal procedure parameter}
             FUNCTION funcparm1 (c : integer) : char);
VAR
   ch : char;
BEGIN
   procparm1(1,2);
   ch := funcparm1(3);
END;
FUNCTION f (PROCEDURE procparm2;           {formal procedure parameter}
            FUNCTION funcparm2 : integer); {formal function parameter}
VAR
   i : integer;
BEGIN
   procparm2;
   i := funcparm2;
END;
PROCEDURE actual_procparm1 (x,y : integer);  {user-defined procedure}
BEGIN
   .
   .
END;
FUNCTION actual_funcparm1 (z : integer) : char; {user-defined function}
BEGIN
   .
   .
END;
PROCEDURE actual_procparm2; {another user-defined procedure}
BEGIN
   .
   .
END;
FUNCTION actual_funcparm2 : integer; {another user-defined function}
BEGIN
   .
   .
END;
BEGIN {prog}
   p(actual_procparm1,  {actual parameter for procparm1}
     actual_funcparm1); {actual parameter for funcparm1}
   s := f(actual_procparm2,  {actual parameter for procparm2}
          actual_funcparm2); {actual parameter for funcparm2}
END. {prog}
```

Parameters of Routine Types

Parameters of routine types (procedure or function types) are like parameters of other user-defined types. They are defined in the formal parameter lists of other routines, but their types—routine types—are defined in type declaration sections. The types must be declared first (see the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for more information on declaring routine types).

The actual parameter for a formal parameter of function type is either:

- The result of the function *addr* when applied to the name of a user-defined function.
- The name of a variable of function type (in which case the value of the variable must be a user-defined function).

In either case, the user-defined function and the formal parameter must have congruent parameter lists and the same result type.

The actual parameter for a formal parameter of procedure type is either:

- The result of the function *addr* when applied to the name of a user-defined procedure.
- The name of a variable of procedure type (in which case the value of the variable must be a user-defined procedure).

In either case, the user-defined procedure and the formal parameter must have congruent parameter lists.

Predefined routines cannot be actual parameters for formal parameters of routine types. For information on variables of routine types, see "Variables of Routine Types."

Example

The procedure `p` has a parameter of procedure type, `procparm1`, and a parameter of function type, `funcparm1`. The function `f` has a parameter of procedure type, `procparm2`, and a parameter of function type, `funcparm2`. Compare this example to the example in "Routine Parameters". See "Congruent Parameter Lists" for examples of congruent parameter lists. See "Fcall Function" and "Call Procedure" for information on the *fcall* function and *call* procedure.

```
$STANDARD_LEVEL 'HP_MODCAL'$
PROGRAM prog;

TYPE
   proctype1 = PROCEDURE (a,b : integer);
   functype1 = FUNCTION (c : integer) : char;
   proctype2 = PROCEDURE;
   functype2 = FUNCTION : integer;

VAR
   s : char;


PROCEDURE p (procparm1 : proctype1;
             funcparm1 : functype1);
VAR
   ch : char;
BEGIN
   call(procparm1,1,2);
   ch := fcall(funcparm1,3);
END;


FUNCTION f (procparm2 : proctype2;
            funcparm2 : functype2);
VAR
   i : integer;
BEGIN
   call(procparm2);
   i := fcall(funcparm2);
END;


PROCEDURE actual_procparm1 (x,y : integer);
BEGIN
   .
   .
   .
END;
FUNCTION actual_funcparm1 (z : integer) : char;
BEGIN
   .
   .
   .
END;
```

(*Example is continued on next page.*)

```
PROCEDURE actual_procparm2;
BEGIN
   .
   .
   .
END;


FUNCTION actual_funcparm2 : integer;
BEGIN
   .
   .
   .
END;


BEGIN {prog}
   p(addr(actual_procparm1), addr(actual_funcparm1));
   s := f(addr(actual_procparm2), addr(actual_funcparm2));
END. {prog}
```

Variables of Routine Types

Variables of routine types (procedure and function types) can be actual parameters for formal parameters of routine types (function and procedure types, respectively). See "Parameters of Routine Types".

The values that you can assign to a function variable are:

- The value *nil*.
- The value returned by the predefined function *addr* when you call it with the name of an appropriate function (*appropriate* is defined below).
- The value returned by any function whose return type is the same function type as that of the variable.
- Another function variable of the same type.

The values that you can assign to a procedure variable are:

- The value *nil*.
- The value returned by the predefined function *addr* when you call it with the name of an appropriate procedure (*appropriate* is defined below).
- The value returned by any function whose return type is the same procedure type as that of the variable.
- Another procedure variable.

A routine is an *appropriate* parameter for *addr* under these conditions:

- The routine and the variable have congruent parameter lists.
- In the case of a function and a function variable, if they have the same result type.
- The routine is declared at the same or a higher level than the variable.
- The routine is not predefined.

Routine variables are system programming extensions. To use them, specify `$STANDARD_LEVEL 'HP_MODCAL'$`. Refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for more information on compiler options.

Example 1

This program uses the predefined function *addr* to assign appropriate functions to a variable of function type and appropriate procedures to a variable of procedure type.

```
$STANDARD_LEVEL 'HP_MODCAL'$
PROGRAM proc (input);

TYPE
   proctype = PROCEDURE (x,y : integer);
   functype = FUNCTION (x,y : integer) : integer;

VAR
   procvar : proctype;
   funcvar : functype;
         b : Boolean;
         i : integer;

PROCEDURE p1 (a,b : integer); EXTERNAL;
PROCEDURE p2 (a,b : integer); EXTERNAL;

FUNCTION f1 (a,b : integer) : integer; EXTERNAL;
FUNCTION f2 (a,b : integer) : integer; EXTERNAL;

BEGIN
   read(b);

   IF b THEN BEGIN
      procvar := addr(p1);
      funcvar := addr(f1);
   END
   ELSE BEGIN
      procvar := addr(p2);
      funcvar := addr(f2);
   END;

   call(procvar,10,20);
   i := fcall(funcvar,10,20);
END.
```

Example 2

This program declares procedures and procedure variables at different levels and assigns each procedure visible to each variable. The comments tell you which assignments are illegal and why.

```
$STANDARD_LEVEL 'HP_MODCAL'$
PROGRAM prog;

TYPE
   proctype = PROCEDURE (x,y : integer);

VAR
   procvar : proctype;

   PROCEDURE p1 (a,b : integer);
   VAR
      pvar1 : proctype;

      PROCEDURE p2 (c,d : integer);
      VAR
         pvar2 : proctype;

         PROCEDURE p3 (e,f : integer);
         VAR
            pvar3 : proctype;
         BEGIN {p3}
            pvar3 := addr(p1);
            pvar3 := addr(p2);
            pvar3 := addr(p3);
         END; {p3}

      BEGIN {p2}
         pvar2 := addr(p1);
         pvar2 := addr(p2);
         pvar2 := addr(p3);  {illegal -- p3 is at a lower level than pvar2}
      END; {p2}

   BEGIN {p1}
      pvar1 := addr(p1);
      pvar1 := addr(p2);  {illegal -- p2 is at a lower level than pvar1}
   END; {p1}

BEGIN {prog}
   procvar := addr(p1);
END. {prog}
```

Example 3

This program uses functions whose return types are function and procedure types to assign values to routine variables. The comments tell you which assignments are illegal and why.

```
$STANDARD_LEVEL 'HP_MODCAL'$
PROGRAM proc;

TYPE
   proctype1 = PROCEDURE (x : integer);
   proctype2 = PROCEDURE (x,y : integer);
   functype1 = FUNCTION (y : real) : integer;
   functype2 = FUNCTION (y : real) : real;

VAR
   procvar : proctype1;
   funcvar : functype1;

FUNCTION returnproc1 (z : integer) : proctype1; EXTERNAL;
FUNCTION returnproc2 (z : integer) : proctype2; EXTERNAL;

FUNCTION returnfunc1 : functype1; EXTERNAL;
FUNCTION returnfunc2 : functype2; EXTERNAL;

BEGIN
   procvar := returnproc1(1);
   procvar := returnproc2(2);  {illegal -- function returns wrong type}

   funcvar := returnfunc1;
   funcvar := returnfunc2;  {illegal -- function returns wrong type}
END.
```

Example 4

Undefined routine variables are undetectable, and cause unpredictable results. The following program avoids problems caused by such undefined variables by assigning the value *nil* to those variables.

```
$STANDARD_LEVEL 'EXT_MODCAL'$
PROGRAM prog (input,output);

VAR
   i,j : integer;
   procvar1 : PROCEDURE (a,b : integer);
   procvar2 : PROCEDURE (VAR c,d : integer);

PROCEDURE alpha (x,y: integer); EXTERNAL;
PROCEDURE beta (x,y: integer); EXTERNAL;

PROCEDURE gamma (VAR x,y: integer); EXTERNAL;
PROCEDURE delta (VAR x,y: integer); EXTERNAL;

BEGIN
   read(i,j);

   {initialize variables of procedure type}

   procvar1 := nil;
   procvar2 := nil;


   {If -100 <= i <= -1, procvar1 is alpha;
    if 0 <= i <= 100, procvar1 is beta}

   IF (i IN [-100..-1] THEN procvar1 := addr(alpha)
   ELSE IF i IN [0..100] THEN procvar1 := addr(beta);


   {If -10 <= j <= -1, procvar2 is gamma;
    if 0 <= j <= 10, procvar2 is delta}

   IF j IN [-10..-1] THEN procvar2 := addr(gamma)
   ELSE IF j IN [0..10] THEN procvar2 := addr(delta);


   {Call procvar1 and procvar2, unless they are nil}

   IF procvar1 = nil THEN writeln('i is out of range')
   ELSE call(procvar1,i,j);

   IF procvar2 = nil THEN writeln('j is out of range')
   ELSE call(procvar2,i,j);
END.
```

Call Procedure

The predefined procedure *call* executes a call to the procedure specified by a procedure variable. Its parameters are a procedure variable and the actual parameters with which the procedure is to be called. Just as a pointer is dereferenced with ^, a procedure variable is dereferenced with *call*.

Example

```
$STANDARD_LEVEL 'EXT_MODCAL'$
PROGRAM prog;

TYPE
   proctype = PROCEDURE (x,y : integer);

VAR
   procvar : proctype;

PROCEDURE p (a,b : integer);
BEGIN
   .
   .
   .
END;

BEGIN
   procvar := addr(p);
   call(procvar,1000,3500);

   p(1000,3500);
END.
```

The calls to the procedures *call* and *p* are semantically equivalent.

The first parameter to *call* (procedure variable) cannot have the value *nil* or be undefined.

Fcall Function

The predefined function *fcall* executes a call to the function specified by a function variable. Its parameters are a function variable (which specifies the function to be called) and the actual parameters with which the function is to be called. Just as a pointer is dereferenced with ^, a function variable is dereferenced with *fcall*.

Example

```
$STANDARD_LEVEL 'EXT_MODCAL'$
PROGRAM prog;

TYPE
   functype = FUNCTION (x,y : integer) : integer;

VAR
   funcvar : functype;
        v1 : ^integer;

FUNCTION f (a,b : integer) : integer;
BEGIN
   f := (a+b)*(a-b);
END;

BEGIN
   new(v1);

   funcvar := addr(f);
   v1^ := fcall(funcvar,27,94);

   v1^ := f(27,94);
END.
```

The calls to the functions *fcall* and `f` are semantically equivalent.

The first parameter to *fcall* (the function variable) cannot have the value *nil* or be undefined.

Congruent Parameter Lists

Two parameter lists are *congruent* if they have the same number of parameters, and if parameters in the same positions are equivalent.

Two parameters are *equivalent* if any one of the following is true:

- They are value parameters of identical type.
- They are VAR parameters of identical type.
- They are parameters of procedure type with congruent parameter lists.
- They are parameters of function type with congruent parameter lists and identical result types.
- They are value conformant array parameters with equivalent schemas.
- They are VAR conformant array parameters with equivalent schemas.

Two conformant array schemas are *equivalent* if all of the following are true:

- Both are PACKED, or neither is PACKED.
- Corresponding index type specifications specify the same type.
- They have the same element type. If they have schemas for element types, then those schemas are equivalent.

Example 1

This program uses procedure parameters whose own parameter lists do not include conformant array parameters, function parameters, or other procedure parameters.

```
PROGRAM prog;

VAR
   r : real;

PROCEDURE proc (PROCEDURE procvar (x : integer; VAR y : char));
BEGIN
   .
   .
END;

FUNCTION func (PROCEDURE pvar (x : integer)) : real;
BEGIN
   .
   .
END;

PROCEDURE p1 (a : integer; VAR b : char); EXTERNAL;
PROCEDURE p2 (a : integer; VAR b : real); EXTERNAL;
PROCEDURE p3 (VAR a : integer; b : char); EXTERNAL;
PROCEDURE p4 (a,b : integer); EXTERNAL;
PROCEDURE p5 (a : integer); EXTERNAL;

BEGIN
   proc(p1);
   proc(p2);  {illegal}
   proc(p3);  {illegal}
   proc(p4);  {illegal}
   proc(p5);  {illegal}

   r := func(p5);
   r := func(p4);  {illegal}
   r := func(p3);  {illegal}
   r := func(p2);  {illegal}
   r := func(p1);  {illegal}
END.
```

The procedure `proc` has a procedure parameter, `procvar`. The parameter list of `procvar` is congruent with the parameter list of the procedure `p1`, but not with those of `p2`, `p3`, `p4`, or `p5`. Therefore, `p1` can be an actual parameter for `procvar`, but `p2`, `p3`, `p4`, and `p5` cannot.

The function `func` has a procedure parameter, `pvar`. The parameter list of `pvar` is congruent with the parameter list of the procedure `p5`, but not with those of `p1`, `p2`, `p3`, or `p4`. Therefore, `p5` can be an actual parameter for `pvar`, but `p1`, `p2`, `p3`, and `p4` cannot.

Example 2

This program uses function parameters whose own parameter lists do not include conformant array parameters, procedure parameters, or other function parameters.

```
PROGRAM prog;

VAR
   r : real;

PROCEDURE proc (FUNCTION funcvar : (a,b,c : char) : Boolean);
BEGIN
   .
   .
   .
END;

FUNCTION func (FUNCTION fvar : (a,b,c : char) : real) : real;
BEGIN
   .
   .
   .
END;

FUNCTION f1 (x,y,z : char) : Boolean; EXTERNAL;
FUNCTION f2 (x,y,z : char) : real; EXTERNAL;

BEGIN
   proc(f1);
   proc(f2);  {illegal}

   r := func(f2);
   r := func(f1);  {illegal}
END.
```

The procedure `proc` has a function parameter, `funcvar`. The parameter list of `funcvar` is congruent with the parameter list of the function `f1`, but not with that of `f2`. Therefore, `f1` can be an actual parameter for `funcvar`, but `f2` cannot.

The function `func` has a function parameter, `fvar`. The parameter list of `fvar` is congruent with the parameter list of the function `f2`, but not with that of `f1`. Therefore, `f2` can be an actual parameter for `fvar` but `f1` cannot.

Example 3

This program uses a procedure parameter, `procvar`. The parameter list of `procvar` includes conformant array parameters, `w` and `x`, another procedure parameter, `p1`, and another function parameter, `f1`.

```
PROGRAM prog;

TYPE
   itype = 1..10;

VAR
   a : ARRAY [1..6] OF integer;
   b : PACKED ARRAY [3..7] OF integer;

PROCEDURE alpha (m : integer); EXTERNAL;

FUNCTION beta (n : real) : integer; EXTERNAL;

PROCEDURE p (VAR cvar1 : ARRAY [a..b : itype] OF integer;
                 cvar2 : PACKED ARRAY [c..d : itype] OF integer;
                 PROCEDURE pvar (e : integer);
                 FUNCTION fvar (f : real) : integer;
            ); EXTERNAL;

PROCEDURE proc (PROCEDURE procvar
                   (VAR w : ARRAY [g..h : itype] OF integer;
                        x : PACKED ARRAY [i..j : itype] OF integer;
                        PROCEDURE p1 (x1 : integer);
                        FUNCTION f1 (x2 : real) : integer
                   )
               );
BEGIN
   procvar(a,b,alpha,beta);
END;

BEGIN
   proc(p);
END.
```

The parameter lists of the formal procedure parameter `procvar` and the procedure `p` are congruent: `cvar1` and `w` are reference conformant array parameters, `cvar2` and `x` are value conformant array parameters, `pvar` and function `p1` are procedure parameters with congruent parameter lists, and `fvar` and function `f1` are function parameters with congruent parameter lists.

Passing a routine as an actual parameter does not change its scope. If it has access to a nonlocal entity before being passed as an actual parameter, then it has access to that entity after being passed—even if the entity is outside the scope of the routine to which the routine is passed.

Example 4

```
PROGRAM prog (output);

PROCEDURE outer2 (PROCEDURE procvar (v : integer));
BEGIN {outer2}
   procvar(7);
END; {outer2}

PROCEDURE outer1 (p : integer);
VAR
   x : integer;
   PROCEDURE inner (i : integer);
   BEGIN {inner}
      writeln(x,i,x+i,p);
   END; {inner}
BEGIN {outer1}
   x := 5;
   outer2(inner);
END; {outer1}

BEGIN {prog}
   outer1(2);
END. {prog}
```

The preceding program prints:

```
5  7  12  2
```

Because the procedure `inner` has access to the nonlocal variables `x` and `p` before being passed to `outer2`, it has access to `x` and `p` after being passed to `outer2` (even though `x` and `p` are outside the scope of `outer2`).

Hidden Parameters

Hidden parameters do not appear in formal or actual parameter lists, but are nevertheless passed to routines. They are always integers.

You must know about hidden parameters in order to debug your program at the assembly language level, and you must include them in the parameter lists of external routines that are not written in Pascal. (For information, see Chapter 9.)

Table 7-3 shows which routines receive hidden parameters, how many hidden parameters they receive, where the hidden parameters are in the physical parameter order, and the values of the hidden parameters.

Table 7-3. Hidden Parameters

Routine With	Receives	Location of Hidden Parameters in Physical Order	Value of [Each] Hidden Parameter
ANYVAR parameters	One hidden parameter for each ANYVAR parameter.	Each one follows its corresponding ANYVAR parameter.	Size in bytes of the actual parameter.
Generic string parameters (not (PACs)	One hidden parameter for each generic string parameter.	Each one follows its corresponding generic string parameter.	Maximum length of string.
Extensible parameter list	One hidden parameter.	First parameter.	Number of actual parameters passed (excluding hidden parameters).
Multi-dimensional conformant array parameters	One hidden parameter for each nested conformant array.	Each one follows bounds values of corresponding nested conformant array.	Element size, in units meaningful to the code that indexes the array.
Routine parameters	One hidden parameter for each routine parameter.	Last parameters.	Static link for containing routine.
External SPL variable	One hidden parameter	First parameter	Same as SPL

ANYVAR Parameters

If a routine has ANYVAR parameters, its physical parameter order contains one hidden parameter for each. In the physical parameter order, each hidden parameter follows its corresponding ANYVAR parameter. The value of each hidden parameter is the size of the corresponding ANYVAR parameter (in bytes).

If the routine specifies the UNCHECKABLE_ANYVAR option, no hidden parameters are passed for ANYVAR parameters.

The UNCHECKABLE_ANYVAR option is used when calling routines that were not written in Pascal.

Example 1

```
$STANDARD_LEVEL 'HP_MODCAL'$
PROGRAM prog;

VAR
   x,y,z : integer;

PROCEDURE p (          a : integer;
             ANYVAR b, c : integer;
                       d : integer;
             ANYVAR e    : integer);
BEGIN {p}
   .
   .
   .
END; {p}

BEGIN {prog}
   x := 2;
   y := 3;
   z := 5;
   p(1,x,y,4,z);
END. {prog}
```

Including hidden parameters (`highlighted`), the parameter list that appears as `p(1,x,y,4,z)` in the preceding program is:

Value 1
Address of x
`Size of x`
Address of y
`Size of y`
Value 4
Address of z
`Size of z`

You can access these hidden parameters with the predefined functions *bitsizeof* and *sizeof*. If the UNCHECKABLE_ANYVAR procedure option is specified, *bitsizeof* and *sizeof* return the size of the formal parameter (for more information on UNCHECKABLE_ANYVAR, see Chapter 8).

Example 2

```
$STANDARD_LEVEL 'EXT_MODCAL'$
PROGRAM prog (output);

TYPE
   t1 = ARRAY [1..20] OF integer;
   t2 = ARRAY [1..11] OF integer;

VAR
   v : t1;

PROCEDURE p1 (ANYVAR parm : t2);
BEGIN {p1}
   writeln('Size of actual parameter = ', sizeof(parm):1);
   writeln('Bit size of actual parameter = ', bitsizeof(parm):1);
END; {p2}

PROCEDURE p2 (ANYVAR parm : t2);
              OPTION UNCHECKABLE_ANYVAR;
BEGIN {p2}
   writeln('Size of formal parameter = ', sizeof(parm):1);
   writeln('Bit size of formal parameter = ', bitsizeof(parm):1);
END; {p2}

BEGIN {prog}
   p1(v);
   p2(v);
END. {prog}
```

The preceding program prints:

```
Size of actual parameter = 80
Bit size of actual parameter = 640
Size of formal parameter = 44
Bit size of formal parameter = 352
```

The procedure `p1` does not specify the option UNCHECKABLE_ANYVAR, so it can access the hidden parameter associated with the actual parameter `v`. The functions `sizeof(parm)` and `bitsizeof(parm)` return the size of the actual parameter `v`.

The procedure `p2` specifies the option UNCHECKABLE_ANYVAR, so it cannot access the hidden parameter associated with the actual parameter `v`, because it is omitted from the physical parameter order. The functions `sizeof(parm)` and `bitsizeof(parm)` return the size of the formal parameter `parm` (that is, the sizes of the type `t2`).

Generic String Parameters

If a routine has generic string parameters, its physical parameter order contains one hidden parameter for each. In the physical parameter order, each hidden parameter follows its corresponding actual string parameter. The value of each hidden parameter is the maximum length of the corresponding actual string parameter.

Extensible Parameter List

If a routine has an extensible parameter list, its physical parameter order begins with a hidden parameter. The value of the hidden parameter is the number of actual parameters passed, excluding hidden parameters. This value is always greater than or equal to the number of nonextension parameters, because the routine must have a value for each of them.

Example

```
$STANDARD_LEVEL 'EXT_MODCAL'$
PROGRAM prog;


PROCEDURE p (x : integer;
             y : real);
          OPTION EXTENSIBLE 1
                 DEFAULT_PARMS (x := 0,
                                y := 1.0);
BEGIN
   .
   .
END;
BEGIN
   p;           {value of hidden parameter is one}
   p(9);        {value of hidden parameter is one}
   p(9, 2.7);   {value of hidden parameter is two}
   p(, 2.7);    {value of hidden parameter is two}
END.
```

The procedure `p` has one nonextension parameter, so the value of the hidden parameter for any call to `p` is at least one.

In the first call above, `p` receives one value from DEFAULT_PARMS; the value of the hidden parameter is one.

In the second call, `p` receives one value from the actual parameter list; the value of the hidden parameter is one.

In the third call, `p` receives two values from the actual parameter list; the value of the hidden parameter is two.

In the fourth call, `p` receives one value from DEFAULT_PARMS and one from the actual parameter list; the value of the hidden parameter is two. For more information on OPTION EXTENSIBLE and OPTION DEFAULT_PARMS, see Chapter 8.

Multidimensional Conformant Array Parameters

If a routine has multidimensional conformant array parameters, its physical parameter order contains one hidden parameter for each nested conformant array element. In the physical parameter order, each hidden parameter follows the actual parameters for the bounds of its corresponding dimension. The value of each hidden parameter is the size of its corresponding dimension. These hidden parameters are not accessible to the programmer. The program uses them to calculate values of the *sizeof* function.

Example

```
PROGRAM prog;

TYPE
   t = 1..10;

VAR
   a : ARRAY [1..3,1..8,1..4] OF integer;

PROCEDURE p (b : ARRAY [lb1..ub1 : t;
                        lb2..ub2 : t;
                        lb3..ub3 : t] OF integer; EXTERNAL;

BEGIN
   p(a);
END.
```

The call `p(a)` passes two hidden parameters to `p`, one for each nested conformant array dimension. Including hidden parameters (`highlighted`), the parameter list that appears in the preceding program as `p(a)` is:

Address of *a*
Value 1 (lb1)
Value 3 (ub1)
Value 1 (lb2)
Value 8 (ub2)
`(UB2-LB2+1) *(UB3)`
Value 1 (lb3)
Value 4 (ub3)
`(UB3)`

Routine Parameters

If a routine has routine parameters, its physical parameter order contains one hidden parameter for each routine parameter. (This is not true of parameters that are routine variables.) These hidden parameters are at the end of the physical parameter order, in the same order as their corresponding routine parameters. The value of a hidden parameter for a specific routine parameter is the static link. This static link allows access to the variables and parameters of the enclosing routines.

Note Level one routines do not require static links. Therefore, they are the only type of routine parameters that can be passed to extensible parameters.

Example

```
PROGRAM prog (input,output);

PROCEDURE p (PROCEDURE param1 (x : integer);
             PROCEDURE param2 (y : integer);
             FUNCTION  param3 (z : integer) : integer;
             v : integer);
VAR
   i : integer;
BEGIN {p}
   param1(v);
   param2(v);
   i := param3(v);
END; {p}

PROCEDURE actual1 (a : integer);
   PROCEDURE actual2 (b : integer);
      FUNCTION actual3 (c : integer) : integer;
      BEGIN {actual3}
         p(actual1,actual2,actual3,100);
      END; {actual3}
   BEGIN {actual2}
      .
      .
   END; {actual2}
BEGIN {actual1}
   .
   .
END; {actual1}
BEGIN
   .
   .
   .
END.
```

Including hidden parameters (`highlighted`), the physical parameter order that appears in the preceding program as `p(actual1,actual2,actual3,100)` is:

Procedure label for procedure *actual1*
Procedure label for procedure *actual2*
Function label for function *actual3*
Value 100
`Static link for procedure actual1 (nil)`
`Static link for procedure actual2 (actual1's locals)`
`Static link for function actual3 (actual2's locals)`

EXTERNAL SPL VARIABLE

The EXTERNAL SPL VARIABLE directive causes the compiler to pass a hidden parameter that specifies the presence of parameters. The hidden parameter is a 32 bit integer with the mask right justified as required by SPL/V.

Example

```
program prog1;
var count : integer;

procedure ext_spl(p1, p2, p3 : integer);
    external spl variable;

begin
ext_spl(1,,count);
ext_spl(1);
end.
```

Including hidden parameters (highlighted), the physical parameter order that appears in the preceding program as `ext_spl(1,,i)` is:

`Value 5`
Value 1
Value 0 (space holder)
Value of `count`

8

Procedure Options

Procedure options, which immediately follow a routine head, can specify:

- That the routine has an extensible parameter list—that is, one or more optional parameters (EXTENSIBLE option).
- Default values for formal parameters, allowing their actual parameters to be left out of actual parameter lists (DEFAULT_PARMS option).
- That formal ANYVAR parameters do not have the usual hidden parameters that specify their sizes (UNCHECKABLE_ANYVAR option).
- That the loader does not resolve the routine until run time (UNRESOLVED option).
- That the routine is duplicated in-line wherever the program calls it (INLINE option).

A routine heading can specify any combination of procedure options.

Example

```
PROCEDURE alpha (a,b,c : integer)
                OPTION EXTENSIBLE 2;

FUNCTION beta (x : integer; y : real) : boolean
              OPTION DEFAULT_PARMS (x:=0, y:=0);

FUNCTION delta (i,j,k : integer) : integer
               OPTION EXTENSIBLE 1
                      DEFAULT_PARMS (i:=0, j:=1, k:=1)
                      UNRESOLVED;

PROCEDURE gamma (ANYVAR r,s : char)
                OPTION UNCHECKABLE_ANYVAR;

PROCEDURE epsilon (ANYVAR t : real)
                  OPTION UNRESOLVED
                         UNCHECKABLE_ANYVAR;

FUNCTION zeta (ANYVAR u : real) : integer
              OPTION UNCHECKABLE_ANYVAR
                     DEFAULT_PARMS (u:=nil)
                     UNRESOLVED;
```

EXTENSIBLE

The EXTENSIBLE routine option identifies a procedure that has an extensible parameter list.

An *extensible parameter list* has a fixed number of nonextension parameters and a variable number of extension parameters. The integer n after the keyword EXTENSIBLE specifies that the first n parameters in the formal parameter list are nonextension parameters (n can be zero). Any other parameters are extension parameters.

A *nonextension parameter* is required. Every call to the routine must provide an actual parameter for it.

An *extension parameter* is optional. A call to the routine can omit its actual parameter from the actual parameter list. However, if the actual parameter list contains an actual parameter for the xth extension parameter, it must contain actual parameters for those before it.

Note You can pass only level 1 procedures to EXTENSIBLE.

You cannot pass large (greater than 8 bytes) value parameters to an extension parameter.

Example

```
PROGRAM prog;
$STANDARD_LEVEL 'EXT_MODCAL'$
VAR
   b : boolean;

FUNCTION f (i,j : integer) : boolean
         OPTION EXTENSIBLE 2;  {both parameters are required}
BEGIN
   .
   .
END;

PROCEDURE p (x,y : integer)
          OPTION EXTENSIBLE 0;  {no parameters are required}
BEGIN
   .
   .
END;

PROCEDURE q (a : integer;
             b : real;
             c : char;
             d : integer)
          OPTION EXTENSIBLE 2;  {first two parameters are required}
BEGIN
   .
   .
END;
```

(*Example is continued on the next page.*)

```
BEGIN
   b := f(36,45);   {legal}
   b := f(20);      {illegal}
   b := f(,66);     {illegal}
   b := f;          {illegal}

   p;               {legal}
   p();             {legal}
   p(100);          {legal}
   p(250,13);       {legal}
   p(,60);          {illegal}

   q(5,9.4);        {legal}
   q(4,3.0,'z');    {legal}
   q(7,8.8,'w',55); {legal}
   q(2,1.1,,93);    {illegal}
   q(,);            {illegal}
   q(,,45);         {illegal}
   q(400,,22);      {illegal}
END.
```

Both parameters of the function `f` are nonextension parameters. Every call to `f` must specify actual parameters for them.

Both parameters of the procedure `p` are extension parameters. A call to `p` can specify or omit actual parameters for them. If the second actual parameter is specified, the first must also be specified.

The first two parameters of the procedure `q` are nonextension parameters; the last two are extension parameters. A call to `q` must specify actual parameters for the first two parameters, but it can specify or omit actual parameters for the last two parameters. If the fourth actual parameter is specified, the third must also be specified.

The number of extension parameters in an extensible parameter list is flexible: you can add new ones later, and you need not recompile programs that call the routine. The updated version of the routine can use the predefined function *haveextension* to determine whether it was passed values for specific extension parameters.

Without the DEFAULT_PARMS procedure option, the predefined function *haveextension* returns *true* and *false* under these conditions:

Function	**Returns** *true*	**Returns** *false*
haveextension(x) where *x* is a formal parameter of the routine that called *haveextension*.	If the routine was passed an actual parameter for *x*.	If the routine was not passed an actual parameter for *x*.

Note A parameter cannot be referenced when *haveextension* would return false.

Example

The procedure `p` has two nonextension parameters:

```
PROCEDURE p (n1,n2 : integer)
          OPTION EXTENSIBLE 2;
BEGIN {p}
   .
   .
END; {p}
```

The program `oldprog` calls the procedure `p`:

```
PROGRAM oldprog;

PROCEDURE p (n1,n2 : integer)
          OPTION EXTENSIBLE 2;
          EXTERNAL;
BEGIN
   p(1,2);
END.
```

The procedure `p` is updated and two new parameters are added. It uses the predefined function *haveextension* to determine whether its two new extension parameters were passed to it.

```
PROCEDURE p (n1,n2,e1,e2 : integer)
          OPTION EXTENSIBLE 2;
BEGIN {p}
   IF haveextension(e1) AND haveextension(e2) THEN BEGIN
      .
      .
   END;
END; {p}
```

The procedure `p` must be recompiled, but the program `oldprog` need not be. Its call to `p` is still legal, as is the call to `p` from the program `newprog`:

```
PROGRAM newprog;
PROCEDURE p (n1,n2,e1,e2 : integer)
          OPTION EXTENSIBLE 2;
          EXTERNAL;
BEGIN
   p(1,2,3,4);
END.
```

A call to a routine with an extensible parameter list contains a hidden parameter. See Chapter 7 for details.

Note

A routine with n extensible parameters is not the same as a procedure with n parameters that does not have EXTENSIBLE, even if the two procedures are otherwise identical. For example, these procedures are not the same:

```
PROCEDURE proc (a,b : char)   PROCEDURE proc (a,b : char);
       OPTION EXTENSIBLE 2;
BEGIN                         BEGIN
END;                          END;
```

DEFAULT_PARMS

The DEFAULT_PARMS procedure option specifies default values to be assigned to formal parameters when actual parameters are not passed to them. If a nonextension parameter has a default value, its actual parameter can be left out of the actual parameter list, and its default value will be assigned to the formal parameter.

A default value must be a constant expression that is assignment compatible with its parameter. The value *nil* is the only legal default for a VAR, ANYVAR, function or procedure parameter.

Example

```
PROGRAM prog;

PROCEDURE p (a,b,c : integer)
          OPTION DEFAULT_PARMS (b:=2,c:=3);  {two have default values}
BEGIN
   .
   .
   .
END;

BEGIN
   p(10);      {a:=10, b:=2 (default), c:=3 (default)}
   p(10,20);   {a:=10, b:=20, c:=3 (default)}
   p(10,,30);  {a:=10, b:=2 (default), c:=30}
   p();        {illegal}
   p(,20);     {illegal}
END.
```

If an extension parameter has a default value, its actual parameter can be left out of the middle or off the end of the actual parameter list. If it is left out of the middle, its default value is assigned to the formal parameter. If it is left off the end, no value is assigned to the formal parameter.

Example

```
PROGRAM prog;

PROCEDURE p (a,b,c : integer)
          OPTION EXTENSIBLE 0  {all parameters are extensible}
                 DEFAULT_PARMS (a:=1,b:=2,c:=3);  {all have default values}
BEGIN
   .
   .
   .
END;

BEGIN
   p(9,,5);   {a:=9, b:=2 (default), c:=5}
   p(6,7);    {a:=6, b:=7, no value assigned to c}
   p(8);      {a:=8, no value assigned to b or c}
   p(,4,5);   {a;=1 (default), b:=4, c:=5}
END.
```

Table 8-1 tells the value that is passed to a formal parameter, x, when x is:

- Nonextension or extension.
- Its actual parameter is specified or not specified.
- It is before, the same as, or after the parameter n, where n is the last parameter for which an actual parameter is specified.

Table 8-1. Values Passed to Formal Parameter x

Type of Parameter	Actual Parameter for x is Specified	Position of x informal parameter list *p(..,n,..)* where n is the last actual parameter specified in the actual parameter list *p(..,n)*		
		x is before n: *p(.x.,n,..)*	x is n: *p(..,x,..)*	x is after n: *p(..,n,.x.)*
Nonextension Parameter	Yes	Actual value	Actual value	Impossible because x > n
Nonextension Parameter	No	Default value if specified; error otherwise	Impossible because x=n	Illegal unless defaulted, then defaulted value
Extension Parameter	Yes	Actual value	Actual value	Impossible because x > n
Extension Parameter	No	Default value if specified; error otherwise	Impossible because x=n	No value

Haveoptvarparm Function

A routine can use the predefined function *haveoptvarparm* to determine whether the value that it received for a formal reference parameter was passed as an actual parameter or defaulted.

The predefined function *haveoptvarparm* returns *true* and *false* under these conditions:

Function	Returns *true*	Returns *false*
haveoptvarparm(x) where *x* is a formal reference parameter of the routine that called *haveoptvarparm*	If the routine was passed an actual parameter for *x*	If the routine was not passed an actual parameter for *x* (in which case, *x* assumes its default value, *nil*)

Example

```
PROGRAM prog;
$STANDARD_LEVEL 'EXT_MODCAL'$

VAR
   i : integer;

PROCEDURE p (VAR x,y : integer)
          OPTION DEFAULT_PARMS (x := nil, y := nil);
VAR
   b : boolean;

BEGIN
   b := haveoptvarparm(x);  {b := true for p(i)}
   b := haveoptvarparm(y);  {b := false for p(i)}
END;

BEGIN
   p(i);  {x=i, y=nil (default)}
END.
```

Table 8-2 tells the value of *haveoptvarparm(x)* when the formal parameter x meets the following conditions:

- Nonextension or extension.
- Its actual parameter is specified or not specified.
- It is before, the same as, or after the parameter n, where n is the last parameter for which an actual parameter is specified.

Table 8-2. Values Returned by Haveoptvarparm(x)

Type of Parameter	Actual Parameter for x is Specified	Position of x in formal parameter list *p(..,n,..)* where n is the last actual parameter specified in the actual parameter list *p(..,n)*		
		x is before n: *p(.x.,n,..)*	x is n: *p(..,x,..)*	x is after n: *p(..,n,.x.)*
Nonextension Parameter	Yes	true	true	Impossible x > n
Nonextension Parameter	No	false	Impossible, because x=n	false
Extension Parameter	Yes	true	true	Impossible, x > n
Extension Parameter	No	false	Impossible, because x=n	false

Haveextension Function

With the DEFAULT_PARMS procedure option, the predefined function *haveextension* returns *true* and *false* under these conditions:

Function	Returns *true*	Returns *false*
haveextension(x) where x is a formal parameter of the routine that called *haveextension*.	If the routine was passed an actual parameter for x, or if DEFAULT_PARMS specified a default for x.	If the routine was not passed an actual parameter for x, and no default was specified for x with DEFAULT_PARMS.

Example

```
PROGRAM prog;
$STANDARD_LEVEL 'EXT_MODCAL'$

PROCEDURE p (a,b,c : integer)
          OPTION EXTENSIBLE 2
                 DEFAULT_PARMS (b:=2);

BEGIN
END;

BEGIN            {haveextension(b)} {haveextension(c)}
   p(10,20);     {true}             {false}
   p(10,20,30);  {true}             {true}
   p(10);        {true}             {false}
END.
```

Table 8-3 tells the value of *haveextension(x)* when the formal parameter x is:

- Nonextension or extension.
- Its actual parameter is specified or not specified.
- It is before, the same as, or after the parameter n, where n is the last parameter for which an actual parameter is specified.

Table 8-3. Values Returned by Haveextension(x)

Type of Parameter	Actual Parameter for x is Specified	Position of x informal parameter list $p(..,n,..)$ where n is the last actual parameter specified in the actual parameter list $p(..,n)$		
		x is before n: $p(.x.,n,..)$	x is n: $p(..,x,..)$	x is after n: $p(..,n,.x.)$
Nonextension Parameter	Yes No	Calling *haveextension(x)* causes a compile-time error.		
Extension Parameter	Yes	true	true	Impossible
	No	true	Impossible, because x=n	false

UNCHECKABLE ANYVAR

The UNCHECKABLE_ANYVAR procedure option specifies that ANYVAR hidden parameters will not be created for a routine. This allows its parameter list to be compatible with the parameter list of a routine written in a language other than HP Pascal. (See Chapter 7 for an explanation of ANYVAR parameters.)

Example

```
PROCEDURE cproc (ANYVAR ip1,ip2 : integer)
          OPTION UNCHECKABLE_ANYVAR;
          EXTERNAL C;
```

The disadvantage of UNCHECKABLE_ANYVAR is that it causes the predefined functions *sizeof* and *bitsizeof* to return the sizes of the types of the formal ANYVAR parameters, instead of the sizes of the actual parameters.

Example

```
PROGRAM prog;
TYPE
   t1 : PACKED ARRAY [1..50] OF char;
   t2 : PACKED ARRAY [1..100] OF char;
VAR
   y : t1;

PROCEDURE p1 (ANYVAR a : t2)
          OPTION UNCHECKABLE_ANYVAR;
VAR
   b : t1;
   i : 1..100;
BEGIN {p1}
   x := sizeof(a);  {x is always 100}
END; {p1}

BEGIN {prog}
END. {prog}
```

The UNCHECKABLE_ANYVAR option is illegal with a routine that has no ANYVAR parameters.

UNRESOLVED

The UNRESOLVED procedure option prevents the compiler/linker/loader from resolving a routine until the program calls it. The routine must be at level one.

To *resolve* a routine is to associate it with its system name. Calling an OPTION UNRESOLVED routine implicitly resolves it at run-time, before it is called. The routine must be resolvable.

Alternatively, an OPTION UNRESOLVED routine can be explicitly resolved by calling the predefined function *addr* with the routine name as its parameter. Then *addr* returns a routine reference that can be assigned to a routine variable and called with the predefined procedure *call* or *fcall*. If the routine cannot be resolved, *addr* returns *nil*.

Example

```
PROGRAM p (output);

VAR
   pv : PROCEDURE;

PROCEDURE p
          OPTION UNRESOLVED;
          EXTERNAL;

BEGIN {p}
   p;              {This ...}

   call(addr(p));  {is equivalent to this ...}

   pv := addr(p);  {and this}
   call(pv);
END. {p}
```

Note On the HP-UX operating system, the UNRESOLVED option causes the *addr* function to return *nil* whether or not the specified routine is resolved.

INLINE

The INLINE procedure option duplicates a routine wherever the program calls it. It makes your program bigger, but faster. It is worthwhile for short routines and when speed is more important than size.

Example

The program:

```
$STANDARD_LEVEL 'EXT_MODCAL'$
PROGRAM prog;
VAR
   i,j,k : integer;

   PROCEDURE max (l1,l2: integer;
                  VAR l3 : integer)
              OPTION INLINE;
   BEGIN
      IF l1 > l2 THEN
         l3 := l1
      ELSE
         l3 := l2 ;
   END;
BEGIN
   max(10,20,i);
   max(i,j,k);
END.
```

is equivalent to the program:

```
PROGRAM prog;
VAR
   i,j,k : integer;

BEGIN

   {max(10,20,i)}

   IF 10 > 20 THEN
      i := 10
   ELSE
      i := 20;

   {max(i,j,k)}

   IF i > j THEN
      k := i
   ELSE
      k := j;
END.
```

The INLINE procedure option requires STANDARD_LEVEL 'EXT_MODCAL'. The equivalent INLINE compiler option does not. Refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for more information on the INLINE compiler option.

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 or after the inlined code.

9

External Routines

An *external routine* is a routine that is not in the compilation unit that calls it. Its source language can be the same as that of the calling compilation unit or it can be different. This chapter explains:

- The EXTERNAL directive, which allows an HP Pascal compilation unit to access an external routine.
- How an HP Pascal program accesses external routines written in C, COBOL II, FORTRAN 77, FORTRAN 66/V, and SPL.
- How a switch stub allows a Native Mode HP Pascal program to access an external routine in a Compatibility Mode SL.
- How a program written in C, COBOL II, FORTRAN 66/V, FORTRAN 77, or SPL accesses an external HP Pascal routine.

EXTERNAL Directive

The EXTERNAL directive allows an HP Pascal compilation unit to access an external routine (a routine in another compilation unit). The source code of the external routine can be any one of the following languages:

- HP Pascal
- HP Pascal/V
- HP C
- HP COBOL II
- FORTRAN 66/V
- HP FORTRAN 77
- SPL

Syntax

```
         [ C            ]
         [ COBOL        ]
EXTERNAL [ FORTRAN      ]
         [ FTN77        ]
         [ SPL          ]
         [ SPL VARIABLE ]
```

Parameters

None	The source code of the external routine is HP Pascal or Pascal/V.
`C`	The source code of the external routine is C. See Table 9-1 for corresponding HP Pascal and C types.
`COBOL`	The source code of the external routine is COBOL II. See Table 9-2 for corresponding HP Pascal and COBOL II types.
`FORTRAN`	The source code of the external routine is FORTRAN 66/V. The compilation unit that makes the call must also contain the compiler option HP3000_16 (see compiler options in the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*). See Table 9-3 for corresponding HP Pascal and FORTRAN 66/V types.
`FTN77`	The source code of the external routine is FORTRAN 77. See Table 9-3 for corresponding HP Pascal and FORTRAN 77 types.
`SPL`	The source code of the external routine is SPL, without option variable parameters. The compilation unit that makes the call must also contain the compiler option HP3000_16 (see compiler options in the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*). See Table 9-4 for corresponding HP Pascal and SPL types.
`SPL VARIABLE`	The source code of the external routine is SPL, with optional variable parameters. You must specify `SPL VARIABLE` (rather than `SPL`) if the external routine has option parameters, even if you do not omit parameters when you call the routine. The compilation unit that makes the call must also contain the compiler option HP3000_16 (see compiler options in the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*). See Table 9-4 for corresponding HP Pascal and SPL types.

The programmer is responsible for matching the formal parameters and result type of the routine containing the EXTERNAL directive with the formal parameters and result type of the external routine. The matching rules are:

- Corresponding formal parameter lists must have the same number of parameters in the same order.
- Corresponding formal parameters must be of corresponding types. (Correspondence depends upon the source language of the external routine. See the parameter descriptions, below.)
- Corresponding formal parameters can have different names.

The INTRINSIC directive is more flexible about matching. See Chapter 10 for details.

The EXTERNAL directive replaces the block in a routine declaration (see the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual* for details). The declaration containing the EXTERNAL directive can be at any level, but the external routine itself must be at level one in its own compilation unit.

Example 1

The Pascal program `Pascal_Pascal` calls the external Pascal procedure `psubproc`. This is the program:

```
$GLOBAL$
PROGRAM Pascal_Pascal(output);
CONST
   looplimit = 10;
TYPE
   loopbound = 1..looplimit;
VAR
   loop : loopbound;
   global,
   dynamic,
   static : integer;
PROCEDURE psubproc (    parm1 : integer;
                    VAR parm2 : integer); EXTERNAL;
BEGIN {pascal_pascal}
   dynamic := 0;
   FOR loop := 1 to looplimit DO BEGIN
      IF loop <= 5 THEN
         static := 10
      ELSE
         static := 20;
      global := loop;
      psubproc(static,dynamic);
      write('Cycle = ', loop, 'Total = ', dynamic);
   END;
   write('Finish processing');
END. {pascal_pascal}
```

This is the external Pascal procedure:

```
$EXTERNAL$
PROGRAM PASCALSUB;
VAR
   global : integer;
PROCEDURE psubproc (    adder : integer;
                    VAR total : integer);
VAR
   localconstant : integer;
BEGIN {psubproc}
   IF (global MOD 2) = 0 THEN
      localconstant := adder * 2
   ELSE
      localconstant := adder;
   total := total + localconstant;
END; {psubproc}
BEGIN
END.
```

You can use the EXTERNAL directive with procedure declarations in the implement part of a module. In such a procedure declaration, repeating the formal parameters is optional. If you do repeat them, they must be identical to those in the export section.

Example 2

```
MODULE m;

EXPORT
   PROCEDURE proc1 (VAR parm1 : integer;
                    VAR parm2 : char);

   PROCEDURE proc2 (VAR parm1 : integer);

IMPLEMENT
   PROCEDURE proc1;  {formal parameters omitted}
      EXTERNAL;

   PROCEDURE proc2 (VAR parm1 : integer);  {formal parameter repeated}
      EXTERNAL;
END;
```

Use the EXTERNAL directive in exported procedures to link routines written in other languages into your program. You are responsible for ensuring that the formal parameters of the exported procedure correspond to those of the actual external procedure.

Note Do not confuse the EXTERNAL directive with the EXTERNAL compiler option. Refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for information on the EXTERNAL compiler option.

Calling HP C from HP Pascal

The table and example in this section assume that the HP Pascal program and the C routine that it calls are both compiled in Native Mode. If the C routine is in a Compatibility Mode SL instead, you must write a switch stub to access it from your HP Pascal program (see "Switch Stubs").

For more information on C types, please refer to the *HP C Programmer's Guide*.

Table 9-1 matches corresponding HP Pascal and C types. It contains only the types that are acceptable for formal intrinsic parameters. The variable *n* is an integer.

Table 9-1. Corresponding HP Pascal and HP C Types

HP Pascal Type	Corresponding HP C Types
`Array:   Not PACKED`	Array of corresponding type[1]
`Array:      PACKED`	Array of corresponding type[1]
Bit16	`unsigned short`
Bit32	`unsigned int`
Bit52	`struct` with two `unsigned ints`
Boolean (false = 0, true = 1)	Character or integer (false = 0, true <> 0)[2]
Char	`unsigned char`
`Enumerated   256 or fewer elements`	`unsigned char`[3]
`Enumerated   257 or more elements`	`unsigned short` or `int`[3]
File	Not available[8]
Function	Function
Function parameter or variable	Pass a pointer that references a C function[6]
Integer	`int` or `long`
Longint	`struct` with two `unsigned ints`
Longreal	`double` or `long float`
PAC of *n* characters	Array of char, index = 1..*n*-1
`Pointer:   Not EXTNADDR`	Pointer to corresponding type
`Pointer:      EXTNADDR`	Long ptr to corresponding type[7]
Procedure	`void function`
Procedure parameter or variable	Pass a pointer that references a C function[6]
Real	`float`[9]
Record	`struct` or `union`[4]

Table 9-1. Corresponding HP Pascal and HP C Types (continued)

HP Pascal Type	Corresponding HP C Types
Set	Not available
Shortint	`short`
String	`char` *[5]
String[*n*]	`char` *[5]
`VAR parameter:` `Not EXTNADDR`	Pointer to parameter
`VAR parameter:` `EXTNADDR`	Long pointer to parameter[7]
0..65535	`unsigned short`

Table 9-1 Notes

1. The lower bound of an HP Pascal array can be any integer, but the lower bound of a C array must be zero.
2. HP Pascal allocates one byte for a Boolean variable. It stores the value in the rightmost bit.
3. A C enumerated variable corresponds to an HP Pascal *integer*, but an HP Pascal enumerated variable corresponds to a C *unsigned char* if it is one byte, a C *unsigned short* if it is two bytes, and a C *unsigned int* if it is four bytes.
4. A C *union* type corresponds to the variant part of an HP Pascal record type. For example:

   The C type *union*

   ```
   typedef union
      {
      int           In ;
      real          Re ;
      unsigned char Ch ;
      } UnionType ;
   ```

   corresponds to the *untagged* HP Pascal record variant

   ```
   UnionType = RECORD CASE integer OF
      1 : (In : integer) ;
      2 : (Re : real) ;
      3 : (Ch : char) ;
      END ;
   ```

   while the *tagged* HP Pascal record variant

   ```
   Tagged_UnionType = RECORD CASE Tag : integer OF
      1 : (In : integer) ;
      2 : (Re : real) ;
      END ;
   ```

corresponds to the C *struct* type

```
typedef struct
   {
   Tag : int ;
   union
      {
      int : In ;
      float : Re ;
      }
   } Tagged_UnionType ;
```

5. The value of an HP C variable of type *(char *)* ends with a NULL. The HP Pascal type *string[n]*, where *n* is the maximum length, corresponds to the HP C type *(char *)*, but has a different layout.

   HP Pascal treats string parameters to external C routines differently. Just before the call to the C routine, HP Pascal puts a NULL character after the current length of the HP Pascal string parameter. The address sent to the C routine is that of the data part of the HP Pascal string parameter. When the C routine returns to the HP Pascal program, HP Pascal strips the NULL character from the HP Pascal string and updates its current length.

6. To pass an actual parameter of this type to a C routine, declare the formal parameter in the EXTERNAL declaration to be of type *integer* (in the Pascal compilation unit that makes the call). Before calling the C routine, call the predefined function *waddress* to get the integer address of the Pascal routine. Pass the integer address to the C routine. For example:

   A C function:

```
int Signal (Sig , Func)
   int Sig ;
   int (*Func) () ;  /* functional parameter */
   {
   ...
   }
```

   A portion of the HP Pascal program that calls the C function:

```
{ EXTERNAL declaration for C function Signal }
FUNCTION Signal (Sig : integer ; Func : integer) ;
   EXTERNAL C ;

{ Procedure whose address is passed to C function Signal }
PROCEDURE Signal_Handler (Sig : integer) ;
   BEGIN
   ...
   END ;

BEGIN { main program }
   { Actual call to C function Signal }
Dummy := Signal(3 , waddress(Signal_Handler) ) ;
END .
```

7. Declaring a *long pointer* in C is analogous to declaring an ordinary pointer in Pascal, except that the "*" is replaced by "^". For example,

```
int Func (Rec)
struct Stat ^Rec ;
```

declares `Rec` to be a `VAR $EXTNADDR$` of type `Stat`.

8. Limited compatibility exists if the callee is written in C to do raw I/O (using read(2) or write(2)) on a Pascal file. Such functions can be called from Pascal by passing the result of a call to *fnum(pascal_file)* to the C function.

9. If you are passing a real parameter to a C routine that expects a float you must compile the routine in ANSI mode or with the `+r` option to the C compiler. This insures that floats are not promoted to doubles. Otherwise, you should pass a longreal value. (For more information refer to the *HP C Programmer's Guide*.

Example 1

The Pascal program `Pascal_C` calls the external C routine `add`, passing a VAR parameter.

Pascal program:

```
PROGRAM Pascal_C (input,output);

VAR
   int1,
   int2,
   int3 : integer;

PROCEDURE add (    parm1 : integer;
                   parm2 : integer;
               VAR parm3 : integer); EXTERNAL C;

BEGIN
   int1 := 25000;
   int2 := 30000;
   add(int1,int2,int3);
   writeln(int3);
END.
```

C routine:

```
void add (a,b,c)
int  a,b;
int  *c;
{
     *c = a + b;
}
```

Example 2

The Pascal program `Pascal_C2` calls the external C routine `cread`. The Pascal program passes a string parameter to the C routine.

Pascal program:

```
PROGRAM Pascal_C2 (output);

VAR
   str : string[40];

FUNCTION c_read (VAR s : string) : Boolean; EXTERNAL C;

BEGIN
   setstrlen(str,0);
   IF c_read(str) THEN
      writeln('str = ', str)
   ELSE
      writeln('couldn''t read str');
END.
```

C routine:

```
#include <stdio.h>
int c_read(s)       /* no Boolean type in C */
char *s;
{
     return (fgets(stdin,s) >= 0);
}
```

Calling COBOL II from HP Pascal

The table and example in this section assume that the HP Pascal program and the COBOL II routine that it calls are both compiled in Native Mode. If the COBOL II routine is in a Compatibility Mode SL instead, you must write a switch stub to access it from your HP Pascal program (see "Switch Stubs").

Table 9-2 matches corresponding HP Pascal and COBOL II types. (It contains only the types that are acceptable for formal intrinsic parameters.) The variable *n* is an integer.

Table 9-2. Corresponding HP Pascal and Cobol II Types

HP Pascal Type	Corresponding Cobol II Types
`Array: Not PACKED`	Array of corresponding type. Specify SYNC.
`Array: PACKED`	Array of corresponding type. Do not specify SYNC.
Boolean (false = 0, true = 1)	Not available.
Char	PIC X (8 bits).
Enumeration	Not available.
File	Not available.
Function	Not available.
Function parameter or variable	Not available.
Integer	(1) PIC S9(5) to S9(9) (2) Level 01, 77, or SYNC without `$CONTROL SYNC 16` (3) COMP or BINARY
Longreal	Not available.
PAC of *n* characters	PIC X(*n*) (8 bits).
Pointer	Not available.
Procedure	Not available.
Procedure parameter or variable	Not available.
Real	Not available.
Record	Build equivalent record.
Set	Not available.
Shortint	Any one of the following: (1) PIC S9 to S9(4) (2) LEVEL 01, 77, or SYNC without `$CONTROL SYNC 16` (3) COMP or BINARY
String	Not available.
String[*n*]	Build equivalent record.
VAR parameter	Default.

Example

The Pascal program `Pascal_COBOL` calls the external COBOL II routine `subprog1`.

Pascal program:

```
PROGRAM Pascal_COBOL (input,output);

VAR
   int1,
   int2,
   int3 : integer;

PROCEDURE subprog1 (VAR parm1 : integer;
                    VAR parm2 : integer;
                    VAR parm3 : integer); EXTERNAL COBOL;

BEGIN
   int1 := 25000;
   int2 := 30000;
   subprog1(int1,int2,int3);
   writeln(int3);
END.
```

COBOL routine:

```
$CONTROL SUBPROGRAM
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SUBPROG1.
 AUTHOR. BP.
 DATA DIVISION.
 LINKAGE SECTION.
 77 IN1      PIC S9(07) COMP.
 77 IN2      PIC S9(07) COMP.
 77 OUT      PIC S9(07) COMP.
 PROCEDURE DIVISION USING IN1, IN2, OUT.
 PARA-1.
   ADD IN1, IN2, GIVING OUT.
   EXIT PROGRAM.
```

Calling FORTRAN 77 from HP Pascal

The table and example in this section assume that the HP Pascal program and the FORTRAN 77 routine that it calls are both compiled in Native Mode. If the FORTRAN 77 routine is in a Compatibility Mode SL instead, you must write a switch stub to access it from your HP Pascal program (see "Switch Stubs").

Table 9-3 matches corresponding HP Pascal and FORTRAN 77 or FORTRAN 66/V types. (It contains only the types that are acceptable for formal intrinsic parameters.) The variable *n* is an integer.

Table 9-3.
Corresponding HP Pascal and FORTRAN 77 or FORTRAN 66/V Types

HP Pascal Type	Corresponding FORTRAN 77 or FORTRAN 66/V Type
`Array: Not PACKED`	An array of a corresponding type. (Pascal arrays are stored in row-major order; FORTRAN arrays are stored in column-major order.)
`Array: PACKED`	Not available
Boolean (false = 0, true = 1)	LOGICAL*1 (false = 0, true = 1)
Char	CHARACTER
Enumeration	Not available
File	Not available
Function	Function[3]
Function parameter or variable	Not available
Integer	INTEGER*4
Longreal	REAL*8 or DOUBLE PRECISION
PAC of *n* characters	CHARACTER**x*, *x* in 1..*n* [1,2]
Pointer	Not available
Procedure	Subroutine[3]
Procedure parameter or variable	Not available
Real	REAL or REAL*4
Record	Build equivalent record
Set	Not available

Table 9-3.
Corresponding HP Pascal and FORTRAN 77 or FORTRAN 66/V Types
(continued)

HP Pascal Type	Corresponding FORTRAN 77 or FORTRAN 66/V Type
Shortint	INTEGER*2
String	CHARACTER*(*)[2]
String[*n*]	CHARACTER*(*)[2]
VAR parameter	Default parameter mechanism
RECORD real_part : real ; imaginary_part : real ; END ;	COMPLEX

Table 9-3 Notes

1. When you call a Pascal routine from a FORTRAN routine, use the FORTRAN directive `$ALIAS` in the FORTRAN compilation unit to specify a nonstandard calling sequence for the Pascal routine. Specify %REF for each character string parameter (the FORTRAN default for character strings is %DESCR). See the example in "How Non-Pascal Programs Call Pascal Routines".

2. For calling FORTRAN 77 from Pascal only. In the FORTRAN 77 compilation unit, declare the parameter as CHARACTER**n* or CHARACTER*(*). For a PAC type HP Pascal parameter, HP Pascal passes the address followed by the length. For either string type HP Pascal parameter, HP Pascal passes the address of the data part of the string followed by its current length. The current length is loaded from the length field. For example:

   A FORTRAN 77 routine:

   ```
   CHARACTER*40 FUNCTION F77_Func (Str1,Str2)
   CHARACTER*80 Str1
   CHARACTER*(*) Str2
   ...
   RETURN
   END
   ```

   An HP Pascal program that calls the FORTRAN 77 routine:

   ```
   TYPE
      Str40 = string[40] ;
      Pac80 = PACKED ARRAY [1..80] OF char ;

   FUNCTION F77_Func (VAR Str1 : Pac80 ;
                      VAR Str2 : Str40) : Str40 ;
                     EXTERNAL FTN77 ;

   VAR
      Vbl1, Vbl2 : Str40 ;
      Pac1 : Pac80 ;
   ```

```
BEGIN { main program }
...
Vbl2 := strrtrim(F77_Func(Vbl1,Pac1)) ;
...
END ;
```

3. This is not correctly implemented in FORTRAN 77.

Example

The Pascal program *Pascal_Fort* calls the external FORTRAN 77 routine *FORTPRC*.

Pascal program:

```
PROGRAM Pascal_Fort (input,output);

TYPE
   char_str = PACKED ARRAY [1..20] OF char;
VAR
   a_str : char_str;
   int1,
   int2,
   sum  : integer;

PROCEDURE fortprc (VAR cstr : char_str;
                   VAR inta : integer;
                   VAR intb : integer;
                   VAR total : integer); EXTERNAL FTN77;

BEGIN
   a_str := 'Add these 2 numbers:';
   int1 := 25;
   int2 := 15;
   writeln(a_str,int1,int2);
   fortprc(a_str,int1,int2,sum);
   writeln(a_str,sum);
END.
```

FORTRAN 77 routine:

```
SUBROUTINE FORTPRC(CSTR,INT1,INT2,SUM)
    INTEGER INT1, INT2, SUM
    CHARACTER CSTR*20

    SUM = INT1 + INT2
    CSTR = "SUM OF TWO NUMBERS: "
    RETURN
    END
```

Note that on HP-UX, you must compile this code with `f77 +800` command-line option or the FORTRAN `$HP9000_800` directive.

Calling FORTRAN 66/V from HP Pascal

FORTRAN 66/V is a Compatibility Mode language only. The FORTRAN 66/V routine that your HP Pascal program calls must reside in a Compatibility Mode SL, and you must write a switch stub to access it from your HP Pascal program (see "Switch Stubs").

The directive EXTERNAL FORTRAN passes parameters the same way in HP Pascal as it does in FORTRAN 66/V.

For corresponding HP Pascal and FORTRAN 66/V types, see Table 9-3 in "Calling FORTRAN 77 from HP Pascal".

Example

The Pascal program `Pass_heap_var` calls the external FORTRAN 66/V routine `FORT`.

Pascal program:

```
$HP3000_16$
PROGRAM Pass_heap_var (input,output);

TYPE
   ptr = ^arr;
   arr = PACKED ARRAY [1..80] OF char;

VAR
   aptr : ptr;

PROCEDURE fort (VAR arrptr : arr); EXTERNAL FORTRAN;

BEGIN
   new(aptr);
   aptr^ := 'I am a dynamic variable';
   fort (aptr^);
END.
```

FORTRAN 66/V routine:

```
SUBROUTINE FORT(PTRARR)
  CHARACTER PTRARR(80)
  DISPLAY PTRARR
  RETURN
  END
```

Calling SPL from HP Pascal

SPL is a Compatibility Mode language only. The SPL routine that your HP Pascal program calls must reside in a Compatibility Mode SL, and you must write a switch stub to access it from your HP Pascal program. The switch stub cannot be written in SPL. (See "Switch Stubs".)

The directive EXTERNAL SPL passes parameters the same way in HP Pascal as it does in Pascal/V.

Table 9-4 matches corresponding HP Pascal and SPL types. (It contains only the types that are acceptable for formal intrinsic parameters.) The variable *n* is an integer.

Table 9-4. Corresponding HP Pascal and SPL Types

HP Pascal Type	Corresponding SPL Type
`Array: Not PACKED`	Array of corresponding type.
`Array: PACKED`	Array of corresponding type.
Bit16	Logical.
Bit32	Array of logical
Bit52	Array of logical
Boolean (false = 0, true = 1)	Byte (odd is false, even is true).
Char	Byte.
`Enumeration 256 or fewer elements`	Byte.
`Enumeration 257 or more elements`	Logical.
File	Not available.
Function	Typed procedure.
Function parameter or variable	Not available.
Integer	Double.
Longint	Array of logical
Longreal (HP3000_16)	Longreal.
PAC of *n* characters	Byte array.
`Pointer Not EXTNADDR`	Not available.
`Pointer EXTNADDR`	Not available.
Procedure	Procedure.
Procedure parameter or variable	Not available.
Real (HP3000_16)	Real.

Table 9-4. Corresponding HP Pascal and SPL Types (continued)

HP Pascal Type	Corresponding SPL Type
Record	Not available, but you can lay out the equivalent.
Set	Not available.
Shortint	Integer.
String	Not available, but you can lay out the equivalent.
String[*n*] (by value only)	Not available, but you can lay out the equivalent.
`VAR parameter   Not EXTNADDR`	Address of parameter.
`VAR parameter       EXTNADDR`	Not available.
-32768..32767	Integer.
0..65535	Logical.

Example 1

The Pascal program `Pascal_SPL` calls the external SPL routine `splprc`.

Pascal program:

```
$HP3000_16$
PROGRAM Pascal_SPL (input,output);

TYPE
   char_str = PACKED ARRAY [1..20] OF char;
   small_int = -32768..32767;

VAR
   a_str : char_str;
   int1,
   int2,
   sum   : small_int;

PROCEDURE splprc (VAR cstr : char_str;
                      inta : small_int;
                      intb : small_int;
                  VAR total : small_int); EXTERNAL SPL;

BEGIN
   a_str := 'Add these 2 numbers:';
   int1 := 25;
   int2 := 15;
   writeln(a_str,int1,int2);
   splprc(a_str,int1,int2,sum);
   writeln(a_str,sum);
END.
```

SPL routine:

```
$CONTROL SUBPROGRAM
BEGIN
PROCEDURE splprc(cstr,int1,int2,sum);
  VALUE int1,int2;
  INTEGER int1,int2,sum;
  BYTE ARRAY cstr;
  BEGIN
    sum := int1 + int2;
    MOVE cstr := "Sum of two numbers: ";
  END;
END.
```

Example 2

The Pascal program `Pascal_SPL_V` calls `splprv`, an external SPL routine with variable parameters.

Pascal program:

```
$HP3000_16$
PROGRAM Pascal_SPL_V (input,output);

TYPE
   char_str = PACKED ARRAY [1..20] OF char;
   small_int = -32768..32767;

VAR
   a_str : char_str;
   int1,
   int2,
   sum  : small_int;

PROCEDURE splprv (VAR cstr : char_str;
                      inta : small_int;
                      intb : small_int;
                  VAR total : small_int);
                  EXTERNAL SPL VARIABLE;

BEGIN
   a_str := 'Add these 2 numbers:';
   int1 := 25;
   int2 := 15;
   writeln(a_str,int1,int2);
   splprv(a_str,int1,int2,sum);
   writeln(a_str,sum);
END.
```

SPL routine with variable parameters:

```
$CONTROL SUBPROGRAM
BEGIN
PROCEDURE splprv(cstr,int1,int2,sum); OPTION VARIABLE;
  VALUE int1,int2;
  INTEGER int1,int2,sum;
  BYTE ARRAY cstr;
  BEGIN
    sum := int1 + int2;
    MOVE cstr := "Sum of two numbers: ";
  END;
END.
```

Switch Stubs

A switch stub is a program that allows your HP Pascal program, which is compiled in Native Mode (the default on PA-RISC machines) to call a routine compiled in Compatibility Mode (the default on earlier HP 3000 machines). The routine must reside in a Compatibility Mode SL.

Figure 9-1 shows how a switch stub works. When the program calls the routine, what actually happens is that the program calls the switch stub (in Pascal) and the switch stub calls the routine in the Compatibility Mode SL. This is transparent to the program and routine (except for performance, which is slower). It is the responsibility of the switch stub to make whatever transformations are necessary to call the Compatibility Mode routine.

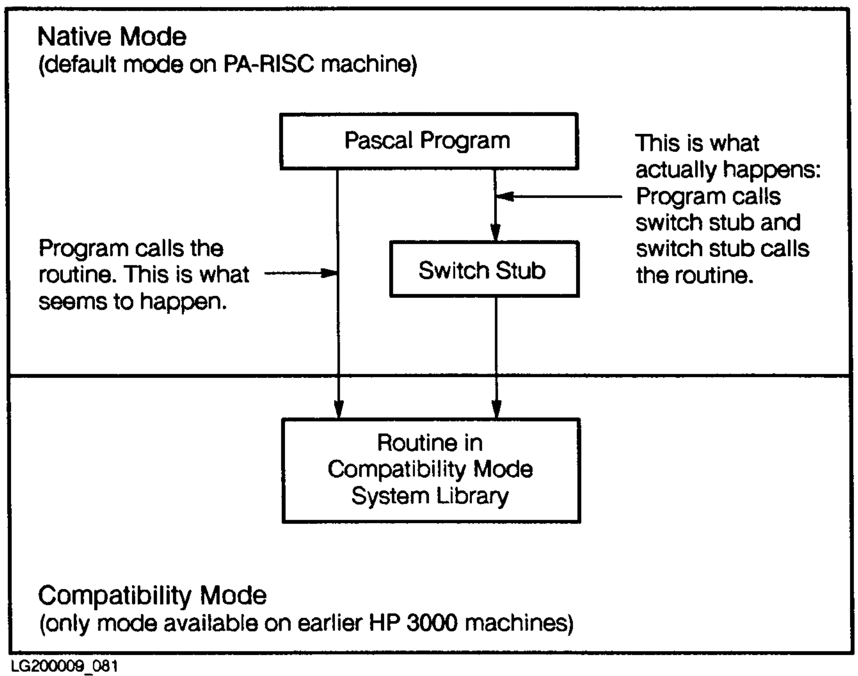


Figure 9-1. How a Switch Stub Works

You must write a switch stub for each Compatibility Mode routine that your program calls. The Switch Assist Tool (SWAT), an interactive utility, can help you write your switch stubs (see step 2 of the example in "Calling SPL from HP Pascal"). For more information, refer to the *Switch Programming Guide*.

How Non-Pascal Programs Call Pascal Routines

A program written in C, COBOL II, FORTRAN 66/V, FORTRAN 77, or SPL can call an external routine written in HP Pascal. You must match the formal parameters and result type of the HP Pascal routine with those that the calling program specifies.

The matching rules are:

- Corresponding formal parameter lists must have the same number of parameters in the same order. If the Pascal routine requires hidden parameters, the non-Pascal routine must have actual parameters that correspond to them (see Chapter 7 for details).
- Corresponding formal parameters must be of corresponding types. Correspondence depends upon the source language of the external routine. See the parameter descriptions in "EXTERNAL Directive".
- Corresponding formal parameters can have different names.

Example 1

This C program calls the external Pascal procedure `pas`:

```
main()
{ extern void pas();  /*This is non ANSI C  */

  char carr[21];
  short sint1, sint2;
  short sum;

  strcpy(carr, "Add these 2 numbers ");
  sint1 = 25;
  sint2 = 15;
  pas(carr, sint1, sint2, &sum);
}
```

This Pascal program contains the procedure `pas`:

```
$SUBPROGRAM$
PROGRAM Pas_Proc;
TYPE
   arr = PACKED ARRAY [1..21] OF char;

PROCEDURE pas (VAR carr  : arr;
                   sint1 : shortint;
                   sint2 : shortint;
               VAR sum   : shortint);
BEGIN
   carr := 'Sum of two numbers: '#0;
   sum := sint1 + sint2;
END;

BEGIN
END.
```

Example 2

The COBOL II program `COBOL-TO-PASCAL` calls the external Pascal procedure `pasprog`.

COBOL II program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBOL-TO-PASCAL.
AUTHOR. BP.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 ASTRING   PIC X(16) VALUE "A COBOL STRING!".
77 ANUM      PIC 9(04) USAGE COMP.
77 ANUM2     PIC 9(04) USAGE COMP.
77 RESULT    PIC -ZZZZ.
PROCEDURE DIVISION.
FIRST-PARA.
  MOVE 9999 TO ANUM.
  DISPLAY ASTRING.
  CALL "PASPROG" USING ASTRING, \ANUM\, ANUM2.
  MOVE ANUM2 TO RESULT.
  DISPLAY ASTRING, RESULT.
  STOP RUN.
```

Pascal procedure:

```
$SUBPROGRAM$
PROGRAM pas_proc;
TYPE
   charstr = PACKED ARRAY [1..16] OF char;

PROCEDURE pasprog(VAR astr : charstr;
                      num : short_int;
                  VAR num2 : short_int);
BEGIN
   astr := 'A PASCAL STRING!';
   num2 := num;
END;
BEGIN
END.
```

Example 3

The following FORTRAN 66/V program calls the external Pascal procedure `pas`:

```
INTEGER INT1, INT2, ISUM
CHARACTER CSTR*20

CSTR = "Add these 2 numbers"
INT1 = 25
INT2 = 15

DISPLAY CSTR, INT1, INT2
CALL PAS(CSTR,\INT1\,\INT2\,ISUM)
DISPLAY CSTR, ISUM

STOP
END
```

Pascal procedure:

```
$SUBPROGRAM$
PROGRAM example(input,output);
TYPE
   arr = PACKED ARRAY [1..20] OF char;
   small_int = -32768..32767;

PROCEDURE pas $CHECK_ACTUAL_PARM 0; CHECK_FORMAL_PARM 0$
   (VAR carr : arr;
        sint : small_int;
        sint2 : small_int;
    VAR sum : small_int);

BEGIN
    carr := 'Sum of two numbers: ';
    sum := sint1 + sint2;
END;

BEGIN
END.
```

Example 4

The following FORTRAN77 program calls the external Pascal procedure `pas`:

```
$ALIAS PAS(%REF,%VAL,%VAL,%REF)
       INTEGER INT1, INT2, ISUM
       CHARACTER CSTR*20

       CSTR = "Add these 2 numbers"
       INT1 = 25
       INT2 = 15

       PRINT *, CSTR, INT1, INT2
       CALL PAS(CSTR, INT1, INT2, ISUM)
       PRINT *, CSTR, ISUM

       STOP
       END
```

Pascal procedure:

```
$SUBPROGRAM$
PROGRAM example;
TYPE
   arr = PACKED ARRAY [1..20] OF char;
   small_int = -32768..32767;

PROCEDURE pas(VAR carr : arr;
                  sint : small_int;
                  sint2 : small_int;
              VAR sum : small_int);

BEGIN
   carr := 'Sum of two numbers: ';
   sum := sint1 + sint2;
END;

BEGIN
END.
```

Example 5

The following SPL program calls the external Pascal procedure `pas`:

```
BEGIN
  LOGICAL ARRAY chr(0:9) := "Add these 2 numbers:";
  BYTE ARRAY bchr(*) = chr;
  INTEGER sint:=15,sint2:=25,len;
  INTEGER int, int2, sum;
  BYTE ARRAY csum(0:1), cint(0:1), cint2(0:1);

  INTRINSIC PRINT,ASCII
  PROCEDURE pas(chr,sint,sint2,sum);
    VALUE sint,sint2;
    INTEGER sint,sint2,sum;
    BYTE ARRAY chr;
    OPTION EXTERNAL;

  PRINT(chr,10,0);
  len := ASCII(sint,-10,cint);
  len := ASCII(sint2,-10,cint2);
  PRINT(cint,-2,0);
  PRINT(cint2,-2,0);

  pas(chr,sint,sint2,sum);

  PRINT(chr,10,0);
  len := ASCII(sum,-10,csum);
  PRINT(csum,-2,0);
END.
```

Pascal procedure:

```
$HP3000_16$
$SUBPROGRAM$
PROGRAM example;
TYPE
   arr = PACKED ARRAY [1..20] OF char;
   small_int = -32768..32767;

PROCEDURE pas(VAR carr : arr;
                  sint : small_int;
                  sint2 : small_int;
              VAR sum : small_int);
BEGIN
   carr := 'Sum of two numbers: ';
   sum := sint1 + sint2;
END;
BEGIN
END.
```

How To Do Pascal I/O with a Non-Pascal Outer Block

Normally, the outer block of a Pascal program allocates space for the default text files `stdin`, `stdout`, and `stderr`. The outer block allocates space even if these files are referenced through Pascal modules (see Appendix A and Appendix B). The outer block also opens these standard files.

In addition, the outer block performs initialization for trap handling for TRY_RECOVER and for the standard Pascal module `arg`.

If the outer block is non-Pascal, the following routine can be used to allocate space, open the default files, and initialize trap handling and the module `arg`.

Example

To compile on MPE/iX, on the command line type:

```
pasxl initstuf,,$null;info="set 'hpux=false'"
```

To compile on HP-UX, on the command line type:

```
pc -c -Dhpux=true init_stuff.p
```

The file (`initstuf` on MPE/iX or `init_stuff.p` on HP-UX) is as follows:

```
{ how to have a non-pascal outer block and still do pascal i/o }
$if 'hpux'$
{ pascal doesn't buffer these files, uses hp-ux system calls }
{ also initialize the data for the module arg, and so that
  the names on the command line are used for file opens. }
$endif$

$global; subprogram$   { allocates text files }
$literal_alias on$
program dick(input,output
$if 'hpux'$  ,stderr $endif$ );
$if 'hpux'$
type argtype = packed array[1..32000] of char;
     argarray= array[0..32000] of ^argtype;
     argarrayptr = ^argarray;
var argc $alias '__argc_value'$ : integer;
    argv $alias '__argv_value'$ : argarrayptr;
    env  $alias '_environ'$ : argarrayptr;

procedure p_init_args $alias 'P_INIT_ARGS'$(c:integer;
                                           v,e:argarrayptr); external;
$endif$
procedure u_init_traps $alias 'U_INIT_TRAPS'$; external;
```

(*Example continued on next page.*)

```
procedure initialize_pascal_standard_files;
begin
$if 'hpux'$
p_init_args(argc,argv,env);          { initialize for module arg }
$endif$
u_init_traps;                        { initialize for trap handling }

{ now open standard files }
reset(input,'$stdin','shared');
rewrite(output,'$stdlist');
$if 'hpux'$
rewrite(stderr,'$stderr');
$endif$
end;
begin end.
```

10

Intrinsics

An *intrinsic* is an external routine that can be called by a program written in any language that the operating system supports. An intrinsic can be written in any supported language, but its formal parameters must be of types that have counterparts in the other supported languages.

An intrinsic definition resides in an intrinsic file (though its code resides in a library). You can use existing intrinsics as they are, modify them, or define new intrinsics. You can put new intrinsics in new or existing intrinsic files and libraries. Your program can access any intrinsic by declaring it and specifying the intrinsic file that defines it.

This chapter:

- Explains how your program can use intrinsics.
- Tells you how to define an intrinsic.
- Tells you how to build or change an intrinsic file.

Using Intrinsics

To use an intrinsic, your program must specify the intrinsic file in which its definition resides and declare the intrinsic with the INTRINSIC directive. How your program can declare the intrinsic as a routine—specifying all, part, or none of its formal parameters—depends upon its definition in the intrinsic file.

This section explains:

- How to specify intrinsic files.
- How to declare an intrinsic with the INTRINSIC directive.
- Actual and intrinsic parameter compatibility.
- How to declare formal function types for an intrinsic.
- How to declare formal parameters for an intrinsic to ensure stricter type checking for actual parameters.
- How to use an intrinsic function as a procedure.

Specifying Intrinsic Files

When compiling a program that references an intrinsic, the compiler reads the intrinsic definition from an intrinsic file. The intrinsic file can be the default intrinsic file for the system, or it can be one that you or another programmer built (see "How to Build or Change an Intrinsic File"). The program can specify different intrinsic files for different intrinsics.

The SYSINTR compiler option determines the intrinsic file. If the program does not contain a SYSINTR option, or if the SYSINTR option does not specify a file name, the compiler reads intrinsic definitions from the default intrinsic file. (The default intrinsic file is system-dependent. See Appendix A for the MPE/iX operating system; Appendix B for the HP-UX operating system.) Otherwise, the compiler reads intrinsic definitions from the file that the SYSINTR option specifies, until another SYSINTR option specifies another file. (See the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for more information on the SYSINTR compiler option.)

To list an intrinsic file, use the LISTINTR compiler option (refer to the *HP Pascal/iX Reference Manual* the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for more information on the LISTINTR compiler option).

Note The compiler options LITERAL_ALIAS and UPPERCASE apply to all external routine names, including intrinsic names. When either of these options is set, the compiler performs a case-sensitive search of the intrinsic file for the intrinsic names.

INTRINSIC Directive

The INTRINSIC directive allows a program to access an intrinsic routine. It follows the routine declaration.

Example

```
PROGRAM p;

VAR
   f,m : shortint;

PROCEDURE FSETMODE; INTRINSIC;

BEGIN
   FSETMODE(f,m);
END.
```

The program `p` can call the intrinsic procedure `FSETMODE` because it declares it with the INTRINSIC directive.

The *system name* of an intrinsic is the name by which the operating system recognizes it, the name that it has in the intrinsic file.

The system names of some intrinsics are illegal in HP Pascal. If you want to use such an intrinsic in your program, give it a legal name in your program and specify its system name with the ALIAS compiler option (refer to the *HP PascaliX Reference Manual* the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for more information on ALIAS).

Example

```
$SYSINTR 'myintr'$  {myintr contains the intrinsic P'F'INFO}
PROGRAM q (output);

PROCEDURE pfileinfo $ALIAS 'P''F''INFO'$; INTRINSIC;
BEGIN
   pfileinfo;
END.
```

The name `P'F'INFO` is illegal in HP Pascal because it contains single quotes. The program *q* can call the intrinsic procedure `P'F'INFO` by the name *pfileinfo* because it declares it with the INTRINSIC directive and specifies its system name with the ALIAS compiler option.

Actual and Intrinsic Parameter Compatibility

An intrinsic's *actual parameters* are those with which your program calls it. Its *intrinsic parameters* are those in its definition, in the intrinsic file. Its *formal parameters* are those that your program declares for it.

Formal parameters are optional. If you do not declare them, you can pass the intrinsic actual parameters of types that would otherwise be incompatible. Usually, programmers want this flexibility; therefore, they rarely declare formal parameters.

If you do not declare a formal parameter, its actual parameters are type-checked against their corresponding intrinsic parameters. Type checking depends upon whether the intrinsic parameter is a reference, value, or function or procedure parameter. The following subsections explain these three cases, using these terms:

alignment-compatible — An actual and intrinsic parameter are *alignment-compatible* if the actual parameter is aligned on the same or a larger boundary than the intrinsic parameter. For example, a 2- or 4-byte-aligned actual parameter is alignment-compatible with a 2-byte-aligned intrinsic parameter. A byte-aligned actual parameter is not alignment-compatible with a 2-byte-aligned intrinsic parameter.

size-compatible — An actual and intrinsic parameter are *size-compatible* if the actual parameter is allocated more or the same amount of space as the intrinsic parameter. For example, a 2- or 4-byte actual parameter is size-compatible with a 2-byte intrinsic parameter. A 1-byte actual parameter is not size-compatible with a 2-byte intrinsic parameter.

intrinsic-compatible — See Table 10-1 for reference parameters; Table 10-2 for value parameters.

Reference Parameter Compatibility

A *reference parameter* is a parameter that is passed by reference. VAR, ANYVAR, and READONLY parameters are reference parameters.

All actual reference parameters must be alignment-compatible with their corresponding intrinsic parameters. Actual VAR and READONLY parameters must also be size-compatible and intrinsic-compatible with their corresponding intrinsic parameters.

An intrinsic and an actual reference parameter are *intrinsic-compatible* if their types are in the same row of Table 10-1. The *intrinsic parameter type* is the type of the intrinsic parameter, as the intrinsic file declares it. The *actual parameter type* is the type of the actual parameter.

Table 10-1. Intrinsic-Compatible Intrinsic and Actual Reference Parameter Types

Intrinsic Parameter Type		Actual Parameter Type
Array		Any type
Boolean		Boolean
Char		Char
Integer		Integer
Integer		Integer subrange $m..n$ with either $m < 0$ or $m >= 0$ and $n > 65535$
Integer subrange $m..n$	$m < 0$, or $m >= 0$, $n > 65535$	Integer, or integer subrange $m..n$ with either $m < 0$ or $m >= 0$ and $n > 65535$
Integer		Bit32
Integer subrange $m..n$	$m >= 0$ and $n <= 65535$	Integer subrange $m..n$ with $m >= 0$ and $n <= 65535$
Integer subrange $m..n$	$m >= 0$ and $n <= 65535$	Bit16
Longreal		Longreal
Real		Real
Record		Any type
Set		Any type
Shortint		Bit16
Shortint		Shortint
Shortint		Integer
Shortint		Integer subrange $m..n$ (except where $m>=0$ and $n<=255$)

Value Parameter Compatibility

A *value parameter* is a parameter that is passed by value. All parameters except VAR, ANYVAR, READONLY, function, and procedure parameters are value parameters.

An actual value parameter of a structured type (array, record, or set) must be the same size as its corresponding intrinsic parameter. An actual value parameter of an unstructured type must be assignment-compatible with its corresponding intrinsic parameter.

Table 10-2 shows which intrinsic and actual value parameter types are intrinsic-compatible. It also shows, for each intrinsic parameter type, which of the compatible actual parameter types are converted to that intrinsic parameter type, and which are not. The *intrinsic parameter type* is the type of the parameter as the intrinsic file declares it. The *actual parameter type* is the type of the actual parameter.

Table 10-2. Intrinsic-Compatible Intrinsic and Actual Value Parameter Types

Intrinsic Parameter Type	Actual Parameter Type	
	Not Converted to Intrinsic Type	**Converted to Intrinsic Type**
Array	Any type	
Boolean	Boolean	
Char	Char	
Integer or Integer Subrange	Array Bit16 Bit32 Bit52 Integer Integer Subrange Longint Record Set Shortint	
Longreal	Longreal	Bit16 Bit32 Bit52 Integer Integer Subrange Longint Real Shortint

Table 10-2.
Intrinsic-Compatible Intrinsic and Actual Value Parameter Types (continued)

Intrinsic Parameter Type	Actual Parameter Type: Not Converted to Intrinsic Type	Actual Parameter Type: Converted to Intrinsic Type
Real	Real	Bit16 Bit32 Bit52 Integer Integer Subrange Longint Longreal Shortint
Record	Any Type	
Set	Any Type	
Shortint	Array Bit16 Bit32 Bit52 Integer Integer Subrange Longint Record Set Shortint	

Function and Procedure Parameter Compatibility

A *function* or *procedure parameter* is a parameter that is a routine. The compiler only checks that the actual parameter for a function or procedure parameter is a routine. You are responsible for making sure that the actual parameter is what the intrinsic expects.

Using Strings as Actual Parameters

If you use a string variable as an actual value parameter to an intrinsic routine, HP Pascal passes a copy of the *data portion* only of the string. The length portion is ignored.

If you use a string variable as an actual reference parameter to an intrinsic routine, HP Pascal passes the address of the *data portion* of the string, and not the string length. If the intrinsic returns data in the string variable, you must determine and update the length of the string when the intrinsic returns control to your program.

There are a number of ways to obtain and update the string length:

- If the intrinsic returns the correct length as a parameter or function return, use the `setstrlen` procedure with the returned value.
- If the length is defined in documentation of the intrinsic, use the `setstrlen` procedure with that value.
- If the intrinsic appends some end-of-string character (such as NUL), scan for the character and set the string length with the `setstrlen` procedure to one less than the character's position.
- If the intrinsic does not provide any length indication, you can use the `strrpt` function to fill the string with blanks to its full physical length, call the intrinsic, and then use the `strrtrim` function to get rid of the trailing blanks and update the string length.

Example

This example demonstrates the sequence of filling a string with blanks, calling an intrinsic that returns a value in the string, and updating the string length.

```
PROGRAM TestIntrin ;
VAR
   Str : string [80] ;
PROCEDURE Dateline ; INTRINSIC ;

BEGIN { main program }
...
Str := strrpt ( ' ' , 80 ) ;   { fill string with blanks }
Dateline ( Str ) ;             { call intrinsic }
Str := strrtrim ( Str ) ;      { remove trailing blanks }
...
END .
```

Formal and Intrinsic Function Type Compatibility

A function type must be specified when using the intrinsic directive with functions. A formal function type is compatible with an intrinsic function type as long as the size of the formal type matches the size of the intrinsic type.

Note In general, the formal type and the intrinsic type should match the function return type. If the types do not match, they are the same as a free union type coercion. This can cause problems for signed versus unsigned types.

Example

```
program m(output);
var a,b:shortint;
    buf:packed array[1..16] of char;
    i:integer;
function calendar:shortint; intrinsic;
function cal_16  $alias 'calendar'$:bit16; intrinsic;
function neg:shortint;
begin
neg:=-1;
end;
begin
a := calendar;
b := calendar;
writeln(a = calendar,' ',a = b);
end.
```

Assuming the date did not change, the output is unexpected:

```
FALSE TRUE
```

Function `cal_16` shows the correct definition; `a` and `b` should be declared as bit16.

User-Defined Formal Parameters

If you want stricter type checking for an intrinsic's actual parameters, you can declare formal parameters for some or all of its intrinsic parameters. Then, actual parameter types are compared to their corresponding formal parameter types, not to their corresponding intrinsic parameter types. This type checking is as strict as that for the parameter of a nonintrinsic routine: if the actual parameter is a reference parameter, it must be of the same type as the formal parameter; if the actual parameter is a value parameter, it must be assignment-compatible with the formal parameter.

If an intrinsic is defined without an extensible parameter list, you cannot declare it with one.

If an intrinsic is defined with an extensible parameter list, you can declare it with or without one. If you declare the intrinsic with an extensible parameter list, you must declare at least as many nonextensible (required) parameters as the definition does. If you declare the intrinsic without an extensible parameter list, you must declare all of its nonextensible (required) parameters.

Example 1

The intrinsic file defines the intrinsic Pascal procedure `intr` this way:

```
PROCEDURE intr (a, b, c, d, e : integer)
               OPTION EXTENSIBLE 2;
```

The program can declare `intr` in any of these ways:

```
PROCEDURE intr (a, b, c, d, e : integer);  {All parameters}
               INTRINSIC;

PROCEDURE intr (a, b : integer);  {Required parameters only}
               INTRINSIC;

PROCEDURE intr (a, b, c : integer);  {First extensible parameter}
               INTRINSIC;

PROCEDURE intr (a, b, c, d : integer);  {Extensible parameters}
               INTRINSIC;
```

The program cannot declare `intr` in any of these ways:

```
PROCEDURE intr (a : integer);  {Without second nonextensible parameter}
          INTRINSIC;

PROCEDURE intr (a, b, c, d : integer)  {Fewer required parameters than}
          OPTION EXTENSIBLE 1;         {in the intrinsic definition}
          INTRINSIC;
```

If you supply default values for the formal parameters that you declare, your default values override those supplied by the intrinsic definition.

Example 2

The intrinsic file defines the intrinsic Pascal procedure `intr` this way:

```
PROCEDURE intr (a, b : integer)
              OPTION EXTENSIBLE 2
              DEFAULT_PARMS (a := 10, b := 20);
```

If the program declares `intr` this way

```
PROCEDURE intr (a, b: integer)
              OPTION EXTENSIBLE 2
              DEFAULT_PARMS (a := 35, b := 60);
              INTRINSIC;
```

Then the default value of `a` is 35 (not 10) and the default value of `b` is 60 (not 20).

If you declare a formal parameter, you must give it a type that is compatible with the type of its corresponding intrinsic parameter. Compatibility rules are different for reference and value parameters.

Reference Parameter Compatibility

A formal reference parameter is compatible with its corresponding intrinsic parameter if any of the following is true:

- Their types (Boolean, integer, etc.) are intrinsic-compatible (see Table 10-3).
- They are alignment-compatible.
- Their types (VAR, ANYVAR, UNCHECKABLE_ANYVAR, READONLY) are compatible.
- If the intrinsic parameter is a VAR or READONLY array, record, or set, then:

  sizeof (*formal_parameter*) `<=` *sizeof* (*intrinsic_parameter*)

An intrinsic and a formal reference parameter are *intrinsic-compatible* if their types are in the same row of Table 10-3. The *intrinsic parameter type* is the type of the intrinsic parameter, as the intrinsic file declares it. The *formal parameter type* is the type of the formal parameter in your program.

Table 10-3. Intrinsic-Compatible Intrinsic and Formal Reference Parameter Types

Intrinsic Parameter Type		Formal Parameter Type
Array		Any type
Boolean		Boolean
Char		Char
Integer		Integer
Integer		Bit32
Integer		Integer subrange $m..n$ with either $m < 0$ or $m >= 0$ and $n > 65535$
Integer subrange $m..n$	$m < 0$, or $m >= 0$, $n > 65535$	Integer, or integer subrange $m..n$ with either $m < 0$ or $m >= 0$ and $n > 65535$
Integer subrange $m..n$	$m >= 0$ and $n <= 65535$	Integer subrange $m..n$ with $m >= 0$ and $n <= 65535$
Integer subrange $m..n$	$m >= 0$ and $n <= 65535$	Bit16
Longreal		Longreal
Real		Real
Record		Any type
Set		Any type
Shortint		Bit16
Shortint		Shortint
Shortint		Integer
Shortint		Integer subrange $m..n$ (except where $m>=0$ and $n<=255$)

Table 10-4 shows which intrinsic and formal reference parameter types are compatible. The *intrinsic parameter type* is the type that the intrinsic parameter has in the intrinsic file; the *formal parameter types* are the types that you can give the formal parameter when you declare it in your program.

Table 10-4. Compatible Intrinsic and Formal Reference Parameter Types

Intrinsic Parameter Type	Formal Parameter Type
VAR	VAR
ANYVAR	ANYVAR VAR
UNCHECKABLE_ANYVAR	UNCHECKABLE_ANYVAR VAR
READONLY	READONLY VAR

Value Parameter Compatibility

A formal value parameter is compatible with its corresponding intrinsic parameter if any of the following is true:

- They are intrinsic-compatible (see Table 10-5).
- If the intrinsic parameter is an array, record, or set, then:

  sizeof (*formal_parameter*) = *sizeof* (*intrinsic_parameter*)

An intrinsic and formal value parameter are *intrinsic-compatible* if their types are in the same row of Table 10-5. The *intrinsic parameter type* is the type of the intrinsic parameter, as the intrinsic file declares it. The *formal parameter type* is the type of the formal parameter.

Table 10-5. Intrinsic-Compatible Intrinsic and Formal Value Parameter Types

Intrinsic Parameter Type	Formal Parameter Type
Array	Array Record
Boolean	Boolean
Char	Char
Integer	Bit16 Bit32 Bit52 Integer Integer subrange Longint Shortint
Integer subrange	Bit16 Bit32 Bit52 Integer Integer subrange Longint Shortint
Longreal	Longreal
Real	Real
Record	Record Array
Set	Set
Shortint	Bit16 Bit32 Bit52 Integer Integer subrange Longint Shortint

Using Intrinsic Functions as Procedures

Your program must use an intrinsic procedure as a procedure, but it can use an intrinsic function as a function, a procedure, or both.

To use an intrinsic function as a function, declare it as a function in your program, including its result type in the declaration. To use an intrinsic function as a procedure, declare it as a procedure in your program, omitting the result type. To use an intrinsic function as both a function and a procedure, declare it both ways, giving the routine different names in your program. Use the ALIAS compiler option to associate the intrinsic's system name with the names you have given it.

If you declare an intrinsic function as a procedure only, you cannot call it as a function.

Example

The intrinsic file defines the intrinsic Pascal functions `f1` and `f2` this way:

```
FUNCTION f1 (i1 : integer) : integer;

FUNCTION f2 (i1,i2 : integer) : Boolean;
```

The Pascal program `prog` declares the function `f1` as a procedure. It cannot call it as a function. It declares the function `f2` as a function (which it calls `ffunc`) and as a procedure (which it calls `fproc`), using the compiler option ALIAS to associate them with the system name `f2`. The program cannot call `fproc` as a function.

```
PROGRAM prog;

VAR
   x : Boolean;
   y,z : integer;

PROCEDURE f1 (a : integer) INTRINSIC;

FUNCTION $ALIAS 'f2'$ ffunc : Boolean; INTRINSIC;

PROCEDURE $ALIAS 'f2'$ fproc; INTRINSIC;

BEGIN
   f1(y);
   x := ffunc(y,z);
   fproc(y,z);
   z := f1(y);      {illegal -- declared as a procedure}
END.
```

Defining Intrinsics

Syntactically, an intrinsic is defined in the same way as any other routine. (Refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual* for details.) Because an intrinsic can be called by a program written in any language that the operating system supports, its intrinsic parameters must be of types that have counterparts in the other supported languages.

These HP Pascal types are acceptable for intrinsic parameters and function returns:

Array
Boolean
Char
Function
Integer
Longreal
Procedure
Real
Record
Set
Shortint
Subrange m..n except where m>=0 and n<=255

These HP Pascal types are *not* acceptable for intrinsic parameters or function returns:

Anyptr
Bit16
Bit32
Bit52
Longint
Conformant array
Enumeration
File
Function type
Globalanyptr
Localanyptr
PAC, with the directive EXTERNAL FTN77 *
Pointer
Procedure type
String
Subrange m..n where m>=0 and n<=255

* An intrinsic parameter of type PAC is not an acceptable intrinsic parameter when used in an external procedure declaration with the directive EXTERNAL FTN77.

If you define your own intrinsics, restrict system programming extensions to:

- Compiler options ALIGNMENT and EXTNADDR (refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*).
- ANYVAR and READONLY intrinsic parameters (explained in Chapter 7).
- Procedure options EXTENSIBLE, UNCHECKABLE_ANYVAR, and DEFAULT_PARMS (explained in Chapter 8).

An intrinsic definition can specify default values for some or all of its parameters with the procedure option DEFAULT_PARMS. If programs that use the intrinsic do not provide actual parameters for these intrinsic parameters, the intrinsic parameters receive their default values.

An intrinsic definition can specify that a given number of its parameters are nonextensible (required) with the procedure option EXTENSIBLE. Programs that use the intrinsic need not provide actual parameters for extensible intrinsic parameters; they *must* provide actual parameters for nonextensible parameters—although the actual parameters can be empty if the DEFAULT_PARMS procedure option specifies default values for them. (See Chapter 8 for more information on the procedure options DEFAULT_PARMS and EXTENSIBLE.)

Compile your intrinsics and create an object file. This object file can be linked with other object files or used to build a library.

How to Build or Change an Intrinsic File

You can build an intrinsic file, or change an existing intrinsic file, with the BUILDINT compiler option and the EXTERNAL directive.

To build a new intrinsic file:

1. Put the BUILDINT option at the front of the compilation unit. Specify a new name for your intrinsic file—do not give it the name of an existing file. (Refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for more information on BUILDINT.)
2. Declare the constants, types, and variables that will appear in your intrinsic routines headings.
3. Declare your intrinsics as you would declare external routines (explained in Chapter 9), except:
   - Use only the acceptable intrinsic parameter types listed in "Defining Intrinsics".
   - Use only these forms of the EXTERNAL directive:

     ```
     EXTERNAL
     EXTERNAL C
     EXTERNAL COBOL
     EXTERNAL FTN77
     ```
4. Leave the outer block of the compilation unit empty.

Example 1

This program builds an intrinsic file.

```
$BUILDINT 'myintr'$
$STANDARD_LEVEL 'EXT_MODCAL'$

PROGRAM build_intrinsic_file;

TYPE
   t_integer_1 = $ALIGNMENT 1$ integer;  {allows byte-aligned integer}

   t_barray = PACKED ARRAY [1..1024] OF CHAR;

   t_status = RECORD
                 f1 : shortint;
                 f2 : shortint;
              END;

PROCEDURE proc1 (    i : integer;
                 VAR b : integer
                );
                EXTERNAL;

PROCEDURE proc2 (ANYVAR $EXTNADDR$ parm1 : t_barray;
                                   parm2 : shortint
                )
                OPTION DEFAULT_PARMS (parm1 := NIL,
                                      parm2 := 0
                                     )
                UNCHECKABLE_ANYVAR;
                EXTERNAL;

PROCEDURE proc3 (    parm1 : integer;
                 VAR parm2 : t_status
                )
                OPTION EXTENSIBLE 1;
                EXTERNAL;

PROCEDURE cob_proc (VAR i : t_integer_1); EXTERNAL COBOL;

BEGIN
{empty body}
END.
```

To change an existing intrinsic file:

1. Put the BUILDINT option at the front of the compilation unit. Specify the name of the intrinsic file that you want to change.
2. Declare any new constants, types, or variables that will appear in new or changed intrinsic routines headings.
3. Declare any new intrinsic routines (see the third instruction for building an intrinsic file). If a new routine has the same name as one that is already in the file, the new one replaces the old one; otherwise, the new one is added to the file.
4. Leave the outer block of the compilation unit empty.

Example 2

This program changes the intrinsic file that the preceding example built, replacing the procedure `proc1` and adding the function `func1`.

```
$BUILDINT 'myintr'$
$STANDARD_LEVEL 'EXT_MODCAL'$

PROGRAM change_intrinsic_file;

PROCEDURE proc1 (    i : shortint;
                 VAR b : shortint;
                 VAR c : integer;
                );
                EXTERNAL;

FUNCTION func1 (p : integer) : shortint; EXTERNAL;

BEGIN
{empty body}
END.
```

To list an intrinsic file that you have built, use the compiler option LISTINTR (for information on compiler options, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

11

Error Recovery and Debugging

There are three types of Pascal errors. They are:

- An *error*, which violates the definition of the HP Pascal language.
- A *compile-time error*, which occurs when you compile your program (as in the case of a syntax error).
- A *run-time* error, which occurs when you run your program (as in the case of a value out of range).

Errors are not to be confused with notes and warnings, both of which occur at compile time. A *note* gives you information that may help you make your program more efficient. A *warning* alerts you to a situation that could cause a run-time error (the compiler cannot tell if it will).

This chapter explains:

- How to write error recovery code for your program, so that it can handle run-time errors that would otherwise cause it to abort (error recovery code does not catch compile-time errors, warnings, or notes).
- How to use the MPE/iX traps that you can use with HP Pascal.
- How to compile your program for use with the HP TOOLSET/XL debugger, the HP Symbolic Debugger, or the system debuggers.

Error Recovery

The system programming extensions that support error recovery are the predefined procedure *escape,* the predefined function *escapecode,* and the TRY-RECOVER construct. They are interdependent. A typical TRY-RECOVER construct has the form:

```
TRY
   statement;
  {statement;}
        .
        .
        .
RECOVER
   BEGIN  {error-handling code}
      temp := escapecode;  {save escapecode value, which can change}

      CASE temp OF  {handle error}
         {handle expected values of temp here}
      OTHERWISE
         escape(temp);  {cannot handle this error here;
                          pass to any enclosing TRY-RECOVER construct}
      END;  {CASE}
   END;  {error-handling code}
```

Escape Procedure

The predefined procedure *escape* is called by your program, a library routine, or the operating system when a run-time error occurs. If a TRY-RECOVER construct is active when the system calls *escape*, the program executes the *statement* associated with the RECOVER part (see "TRY-RECOVER Construct"). If no TRY-RECOVER construct is active, the program aborts. A TRY-RECOVER construct is active if the TRY statement has been executed, but the RECOVER statement has not.

The procedure *escape* has one parameter, *error_code*, which is an integer expression. *Escape* sets *error_code*, whose value you can then access with the predefined function *escapecode*.

Example

```
PROGRAM p;
VAR
   x : integer;
   ecode : integer;
   .
   .
PROCEDURE PUTJCW; INTRINSIC;

PROCEDURE proc (n : integer);
BEGIN {proc}
   {Test for erroneous parameter}
   IF NOT (n IN [0..100]) THEN
      escape(-755);
      .
      .
   putjcw(jcwname,jcwvalue,error);  {system call}

   IF error > 0 THEN
      escape(error);  {system call failed}
      .
      .
END;  {proc}

BEGIN {main program}
   TRY
      proc(x);
   RECOVER
      ecode := escapecode;  {See note in "Escapecode Function"}

      IF ecode = -775 THEN
         {Report bad value of m}
      ELSE IF ecode = -3550 THEN
         {Report failure of system call}
      ELSE
         halt(ecode);
END. {main program}
```

Escapecode Function

The predefined function *escapecode* returns the integer value of *error_code*, the parameter of the predefined procedure *escape* (see “Escape Procedure”).

The result of *escapecode* is undefined if *escape* was never called, and after exit from the TRY-RECOVER construct by normal, sequential means (rather than exit by explicit *escape, exit,* or *goto*). If you call *escapecode* when its result is undefined, the result is indeterminate and meaningless. Access *escapecode* only in the RECOVER part of a TRY-RECOVER construct.

To see the symbolic names for the escape codes that the Pascal subsystem returns, list the file PASESC.PUB.SYS (on MPE/iX) or `/usr/include/pasesc.ph` (on HP-UX).

TRY-RECOVER Construct

The TRY-RECOVER construct defines a group of statements as error recovery code.

Syntax

`TRY` *statement* `[;` *statement*`]...` `RECOVER` *statement*

Parameter

statement Labeled or unlabeled statement.

If an error occurs when the program executes a *statement* (or any routines called by the statement in the TRY part):

1. The subsystem in which the error occurred (the program, a library, or the operating system) calls the predefined procedure *escape* with *error_code* as its parameter. The parameter *error_code* is an integer expression whose value represents the error.
2. The procedure *escape* sets *error_code* and saves it.
3. The program's run-time environment reverts to that of the program unit (main program, procedure, or function) that contains the TRY-RECOVER construct.
4. The program executes the *statement* of the RECOVER part (skipping any *statement*s between the *statement* where the error occurred and the RECOVER's *statement*).

If no *statement* causes an error, the program skips the RECOVER's *statement* and executes the statement that follows the TRY-RECOVER construct.

Example 1

```
PROGRAM prog (input,output);
$STANDARD_LEVEL 'HP_MODCAL'$

VAR
   i,j,k,l : integer;

PROCEDURE proc;
BEGIN
   i := 0;
   j := 0;
   k := 0;
END;

BEGIN
   TRY
      read(i);  {Error here transfers control to proc.}

      read(j);  {Executed only if no error occurs for read(i).
                 Error here transfers control to proc.}

      read(k);  {Executed only if no error occurs for read(i) or read(j).
                 Error here transfers control to proc.}

   RECOVER
      proc;  {Executed only if an error occurs
              for read(i), read(j), or read(k).}

   l := i+j+k;  {Always executed.}
END.
```

If the RECOVER's *statement* is empty, the person who is running the program will not know when the TRY-RECOVER construct has handled an error.

If an error occurs when the program executes the RECOVER's *statement*, the program aborts—unless the TRY-RECOVER construct is within another TRY-RECOVER construct. In that case, the program executes the RECOVER *statement* of the outer TRY-RECOVER construct.

Example 2

```
PROGRAM prog (input,output);
$STANDARD_LEVEL 'HP_MODCAL'$

VAR
   i,j : integer;
   iok : Boolean;

PROCEDURE newj;
BEGIN
   writeln('That value is illegal.');
   prompt('Please enter an integer for j:');
   read(j);
END;

PROCEDURE newij;
BEGIN
   IF NOT iok THEN i := 0 ELSE newj;
END;

BEGIN {prog}
   iok := FALSE;

   TRY
      prompt('Enter an integer for i:');
      read(i);      {An error here transfers control to newij}
      iok := TRUE;  {Not executed if read(i) causes an error}

      TRY
         read(j);   {An error here transfers control to newj}
      RECOVER
         newj;      {An error here transfers control to newij}

   RECOVER
         newij;     {An error here aborts the program}
END. {prog}
```

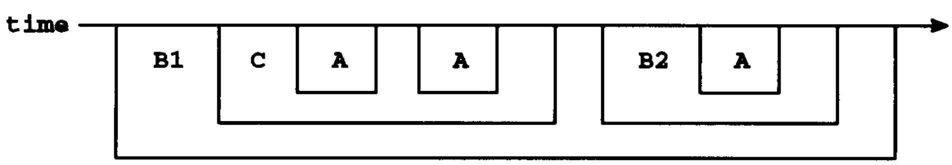
Example 3

The following example illustrates how nested TRY-RECOVER statements divide the responsibility of error recovery.

```
PROGRAM x;
$STANDARD_LEVEL 'HP_MODCAL'$
   PROCEDURE a;
   BEGIN {a}
      TRY
      RECOVER;    } A
   END; {a}

  PROCEDURE b;

      PROCEDURE c;
      BEGIN {c}
         TRY
            a;
            a;          } C
        RECOVER ;
      END; {c}

   BEGIN {b}
      TRY
         c;
         TRY
            a;    } B2     } B1
      RECOVER;
   RECOVER;
  END; {b}

BEGIN {x}
  b;
END. {x}
```

LG200009_109

The diagram below shows when, in time, the TRY-RECOVER statements labeled A, B1, B2, and C in the preceding program are active. When more than one TRY-RECOVER statement is active, the innermost one takes precedence.



LG200009_113

The RECOVER's *statement* can use the predefined function `escapecode` to determine the error that occurred and act accordingly.

Example 4

```
PROGRAM system;

IMPORT
   system_escapecodes;  {see note following example}

   PROCEDURE support;
   BEGIN
      IF error THEN escape(88);
   END;

   PROCEDURE userprogram;
   BEGIN
      support;
   END;

BEGIN {system}
   TRY userprogram
   RECOVER
      CASE escapecode OF
         minuser..maxuser : writeln('Software detected errors');
         range            : writeln('Value range error');
         stackoverflow    : writeln('Stack overflow');
         ioverflow        : writeln('Integer overflow');
         idivbyzero       : writeln('Integer divide by zero');
         roverflow        : writeln('Real overflow');
         runderflow       : writeln('Real underflow');
         rdivbyzero       : writeln('Real divide by zero');
         nilpointer       : writeln('Nil pointer reference');
         casebounds       : writeln('Case expression bounds error');
         stroverflow      : writeln('String overflow');
         filerror         : writeln('File I/O error');
      OTHERWISE
         writeln('Unrecognized error');
   END; {CASE}
END. {system}
```

Note This is only an example. The operating system on which HP Pascal runs does not use the constants that represent error codes in the example above (*ioverflow, roverflow,* and so on).

A program can access `error_code` only by calling the predefined function `escapecode`.

TRY-RECOVER and Optimization

If the OPTIMIZE compiler option is used with the TRY-RECOVER construct, the following information explains what will or will not work at different levels.

- If an ESCAPE is done in the TRY block, or in any procedure called from within the TRY block, all values on the left side of an assignment statement, appearing before an ESCAPE or a procedure call, are stored.
- If a trap occurs instead of an ESCAPE, the above statement is not true.

Example

The following example uses the local variable `flag` to indicate how far the program gets before an error. It is used to undo or unlock a resource.

```
$standard_level 'ext_modcal'$
$ovflcheck off$
program dick;
type iptr=^integer;
procedure lock; external;
procedure plock $alias 'lock'$; begin end;

procedure proc(j:integer;p:iptr);
var flag: {$VOLATILE$} boolean;
    i:integer;
begin
flag:=false;
try
   lock;
   flag:=true;
   i:=maxint;
   i:=i + j + p^;
   if j < 0 then escape(i);
recover
   begin
   if not flag then halt(1);       { should not halt }
   end;
end;
begin
proc(1,nil);
end.
```

This program does not work correctly with optimization because the store to the variable `flag` is done after the trap. To run the program correctly, use `$VOLATILE$` so that `flag` is stored before the trap occurs. See Chapter 12 for more information on the optimizer.

Assert Procedure

The predefined procedure *assert* allows your program to test assumptions, specify invariant conditions, and check data structure integrity.

Syntax

```
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)*.

Figure 11-1 illustrates how the predefined procedure *assert* works.

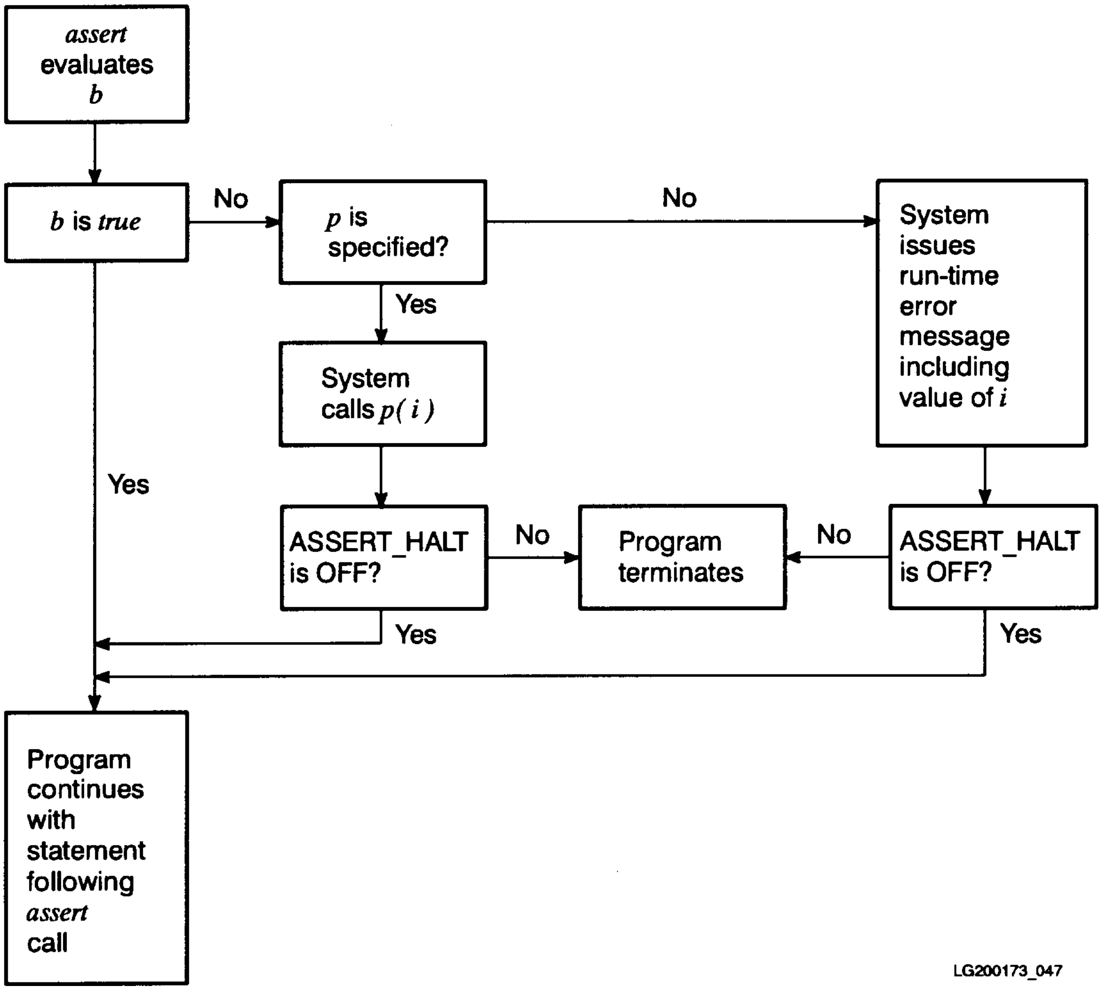


Figure 11-1. How the Predefined Procedure Assert Works

The default for the ASSERT_HALT compiler option is OFF (see the *HP Pascal/iX Reference Manual* or *HP Pascal/HP-UX Reference Manual* 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;
```

Traps

Your HP Pascal program can use these MPE/iX traps:

- MPE/iX intrinsic XLIBTRAP, which traps library errors.
- MPE/iX intrinsic XARITRAP, which traps arithmetic errors.
- MPE/iX intrinsics ARITRAP and HPENBLTRAP, which allow you to enable and disable trap conditions.
- MPE intrinsic XCONTRAP, which specifies a user-defined routine to handle the subsystem break (CONTROL Y).

The subsections of this section explain how to use these traps.

Note The user trap-handling routines whose addresses are passed to the traps in this section must be level-one routines.

Note The XLIBTRAP, XARITRAP, ARITRAP, and HPENBLTRAP routines can also be use on HP-UX.

ARITRAP and HPENBLTRAP Intrinsics

The MPE/iX intrinsics ARITRAP and HPENBLTRAP are supported by the Trap Subsystem. ARITRAP allows a user program to enable or disable traps collectively. HPENBLTRAP is a new MPE/iX intrinsic that allows a user program to enable selected trap conditions.

These terms apply to trap conditions:

Term	**Meaning**
enable	To allow a trap to be raised if the trap condition occurs.
arm	To specify that a particular trap handler is to be called if a certain trap is raised (the trap must be enabled to be raised).
disable	To prevent a trap from being raised, even if the trap condition occurs.

By default, all traps except IEEE floating-point traps are enabled. (This complies with the IEEE floating-point standard, which stipulates that IEEE traps are to remain disabled by default.)

Syntax

ARITRAP (*flag*);

HPENBLTRAP (*mask, oldmask*);

Parameters

flag
: 32-bit integer, passed by value. If *flag* is zero, all traps are disabled; otherwise, all traps are enabled.

mask
: 32-bit integer, passed by value, whose bits specify which trap conditions are enabled. The assignment of each position in the bit mask is described in "XARITRAP Intrinsic."

oldmask
: 32-bit integer, passed by reference, in which the old value of *mask* is returned.

On MPE/iX, declare ARITRAP and HPENBLTRAP as external procedures this way:

```
PROCEDURE ARITRAP; INTRINSIC;

PROCEDURE HPENBLTRAP; INTRINSIC;
```

On HP-UX, declare ARITRAP and HPENBLTRAP as external procedures this way:

```
$PUSH; UPPERCASE ON$
PROCEDURE ARITRAP (Flag : integer); EXTERNAL;

PROCEDURE HPENBLTRAP (    Mask    : integer;
                      VAR OldMask : integer
                     ); EXTERNAL;
$POP$
```

Example

```
ARITRAP (1);                         {enables all traps}

HPENBLTRAP (Hex('0007C000'), OldMask);  {enables IEEE floating-point traps}
```

XLIBTRAP Intrinsic

The MPE/iX intrinsic XLIBTRAP is supported by the HP Pascal run-time library. It enables a user program to arm a library trap handling procedure (Library Trap Handler). Subsequently, any Pascal library error causes this Library Trap Handler to be called, allowing the user to decide whether to abort or continue the program, or correct the error.

Syntax

`XLIBTRAP (`*plabel, oldplabel*`);`

Parameters

plabel — 32-bit integer, passed by value, which is the address of the Library Trap Handler.

oldplabel — 32-bit integer, passed by reference, in which the old value of *plabel* is returned.

On MPE/iX, declare XLIBTRAP as an external procedure this way:

```
PROCEDURE XLIBTRAP; INTRINSIC;
```

On HP-UX, declare XLIBTRAP as an external procedure this way:

```
$PUSH; UPPERCASE ON$
PROCEDURE XLIBTRAP (    PLabel    : INTEGER;
                    VAR OldPLabel : INTEGER
                   ); EXTERNAL;
$POP$
```

XLIBTRAP stores the address of the Library Trap Handler (*plabel*) so that the library routines can find the routine to call if an error occurs. The old value of `PLabel` is returned in the parameter `OldPLabel`.

The only ways to leave a trap handler is by a normal return or by an *escape*. Your library trap handler cannot execute a nonlocal *goto* (a *goto* whose destination is outside the procedure).

Note — This routine is available on the MPE/iX and HP-UX operating systems. On MPE/iX, it expects an MPE-style *plabel*; on HP-UX, it expects *plabel* to be the actual address of the Library Trap Handler. To make your program portable, use *baddress* (*Library_Trap_Handler_name*) as *plabel*.

Note — The result record will be different if the trap has been raised outside of the Pascal run time library.

The user's trap handler must be declared this way:

```
TYPE
   PStkMrk = RECORD  {Stack Marker}
               users_PCS : integer; {space id of users code space}
               users_PCO : integer; {program counter offset within the
                                     code space}
               users_SP  : integer; {stack pointer of the user's
                                     routine that called the library
                                     routine where the error occurred}
               users_DP  : integer; {data pointer for the above routine}

               {future implementations may have further fields to return
                more information to the user's trap handler.  If so, they
                will not affect existing code that uses the above fields.}
             END;

PROCEDURE My_Library_Trap_Handler (VAR StkRec    : PStkMrk;
                                   VAR ErrorCode : Integer;
                                   VAR AbortFlag : Integer
                                  );

BEGIN {My_Library_Trap_Handler}
      .
      .
      .
END;  {My_Library_Trap_Handler}
```

Where

`StkRec` A structure, as described above, passed by reference. Any changes to the fields of this structure are not reflected in the actual contents of the machine registers, when and if the program resumes normal execution.

`ErrorCode` 32-bit integer, passed by reference, which contains the error code. For a complete list of error codes generated by the Pascal run-time library, see the file `PASESC.PUB.SYS` (on MPE/iX) or `/usr/include/pasesc.ph` (on HP-UX). Either of these files can be directly included in a user program.

`AbortFlag` 32-bit integer, passed by reference. If `AbortFlag` is zero when the Library Trap Handler is exited, the program continues to execute. If `AbortFlag` is not zero, the Pascal run-time library prints an error message and aborts the program.

To trap all run-time library errors and have them invoke your Library Trap Handler, call `XLIBTRAP` this way:

`XLIBTRAP (`*baddress*`(`*My_Library_Trap_Handler*`),` *OldPLabel*`);`

To disable your Library Trap Handler, pass zero to XLIBTRAP as the first parameter.

Example

```
{the user declares the following Pascal record for the PStkMrk record}

TYPE
   PStkMrk = RECORD  {"Stack Marker"}
               users_PCS,
               users_PCO,
               users_SP,
               users_DP  : integer;
             END;

$INCLUDE '/usr/include/pasesc.ph'$  {this file lists all the Pascal
                                     run-time library error codes
                                     for the HP-UX operating system}

PROCEDURE My_Library_Trap_Handler (VAR StkRec    : PStkMrk;
                                   VAR ErrorCode : Integer;
                                   VAR AbortFlag : Integer
                                  );

BEGIN {My_Library_Trap_Handler}

   {ignore file close errors, abort on all others}

   IF (ErrorCode = PasErr_CloseError) THEN BEGIN
      writeln ('Oops! File close error, continue execution');
      AbortFlag := 0;   {no abort}
   END
   ELSE
      AbortFlag := 1;   {print message and abort}
END;  {My_Library_Trap_Handler}
```

XARITRAP Intrinsic

The MPE/iX intrinsic XARITRAP is supported by the Trap Subsystem. XARITRAP enables your program to arm an arithmetic trap handling procedure (Arithmetic Trap Handler). Subsequently, any arithmetic error causes this Arithmetic Trap Handler to be called, allowing the user to decide whether to abort or continue the program, or correct the error.

For more information on trap handling, see the *Trap Handling Programmer's Guide*.

Syntax

To arm your Arithmetic Trap Handler, call XARITRAP this way:

`XARITRAP (`*mask, plabel, oldmask, oldplabel*`);`

Parameters

mask 32-bit integer by value, whose bits specify which trap condition gets armed. The assignment of each position in the bit mask is as follows:

Bit	Error Trap
31	Compatibility Mode floating-point divide by zero
30	Integer divide by zero
29	Compatibility Mode floating-point underflow
28	Compatibility Mode floating-point overflow
27	Integer Overflow
26	Compatibility Mode double precision overflow
25	Compatibility Mode double precision underflow
24	Compatibility Mode double precision divide by zero
23	Decimal Overflow (COBOL)
22	Invalid ASCII digit (COBOL)
21	Invalid decimal digit (COBOL)
20-19	Reserved
18	Decimal divide by zero
17	IEEE floating-point inexact result
16	IEEE floating-point underflow
15	IEEE floating-point overflow
14	IEEE floating-point divide by zero
13	IEEE floating-point invalid operation
12	Range error (subrange violations, etc)
11	NIL pointer dereference
10	Result of pointer arithmetic is misaligned or error in conversion from long to short pointer
9	Unimplemented condition traps
8	Paragraph stack overflow (COBOL)
7-1	Reserved
0	Assertion Trap

plabel 32-bit integer, passed by value, which is the address of the Arithmetic Trap Handler.

oldmask 32-bit integer, passed by reference, in which the old value of *mask* is returned.

old plabel 32-bit integer, passed by reference, in which the old value of *plabel* is returned.

On MPE/iX, declare XARITRAP as an external procedure this way:

```
PROCEDURE XARITRAP; INTRINSIC;
```

On HP-UX, declare XARITRAP as an external procedure this way:

```
$PUSH; UPPERCASE ON$
PROCEDURE XARITRAP (    Mask,
                         plabel : integer;
                     VAR OldMask,
                         OldPlabel : integer
                    ); EXTERNAL;
$POP$
```

XARITRAP stores the address of the Arithmetic Trap Handler (*plabel*) so that the system trap handler can find the routine to call if an error occurs. The old value of *plabel* is returned in the parameter `OldPLabel`.

The only ways to leave a trap handler is by a normal return or by an *escape*. Your library trap handler cannot execute a nonlocal *goto* (a *goto* whose destination is outside the procedure).

Note This routine is available on both the MPE/iX and HP-UX operating systems. On MPE/iX, it expects an MPE-style *plabel*; on HP-UX, it expects *plabel* to be the actual address of your Library Trap Handler. To make your program portable, use *baddress*(*Arithmetic_Trap_Handler_name*) as *plabel*.

IEEE floating-point numbers are the default (native) real numbers in HP Precision Architecture. Compatibility Mode floating-point numbers have the format of reals on the MPE V system. The compiler options HP3000_32 and HP3000_16 specify native and compatibility Mode real numbers, respectively. For more information on HP3000_32 and HP3000_16, see the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

The user's trap handler must be declared this way:

```
TYPE
   TrapInfo= RECORD
               Instruction : integer; {the actual instruction word that
                                       caused the arithmetic trap}
               PC_Offset   : integer; {offset of the above instruction
                                       within the user's code space}
               PC_Space    : integer; {space id of user's code space}
               Error_Code  : integer; {Trap type. This word is formed
                                       by setting the bit corresponding
                                       to the trap condition in a 32-bit
                                       integer, with all other bits zero.
                                       More than 1 bit will be turned on
                                       if multiple traps occur together}

              {more fields are returned for certain of the trap conditions.
               See below for details}
             END;

PROCEDURE My_Arith_Trap_Handler (VAR Info : TrapInfo );
BEGIN {My_Arith_Trap_Handler}
   .
   .
   .
END; {My_Arith_Trap_Handler}
```

To enable (for example) all integer and IEEE floating-point traps (except inexact results), as well as all pointer traps, call XARITRAP this way:

```
XARITRAP (
               {bit  0         1         2         3 }
               {     01234567890123456789012345678901}
            Binary ('00000000000111111000000000010010'),
            BAddress (My_Arith_Trap_Handler),
            OldMask,
            OldPLabel
          );
```

Note In the preceding example, the IEEE inexact result trap is not enabled.

HP Precision Architecture has only three distinct hardware arithmetic trap conditions: *condition,* [integer] *overflow,* and *assist exception* (IEEE floating-point traps are in the last category). The system is able to categorize most integer and decimal traps (except integer overflow) because each category has its own unique trapping instructions. If a condition trap occurs, and the system cannot categorize it, *unimplemented condition trap* (bit 9) is raised.

The *IEEE inexact result trap* (bit 17), a trap required by the IEEE floating-point standard, indicates that a floating-point operation may have caused an inexact result (for example, the result of 10.0/3.0 is 3.333 ... regardless of the number of bits of precision you use). This trap is useful only for specialty number-crunching programs. Indiscriminate arming of this trap can severely degrade program performance, because almost any floating-point operation you perform will cause this trap to be raised.

To disable your Arithmetic Trap Handler, pass zero to XARITRAP as the second parameter.

For the following traps, the system trap handler passes your Arithmetic Trap Handler more fields than the four defined above in the *TrapInfo* record, and you must adjust *TrapInfo* accordingly.

```
Integer overflow trap
Decimal overflow trap
Invalid ASCII digit trap
Invalid decimal digit trap
IEEE floating-point traps
Compatibility Mode floating-point traps
```

The following sections describe the extra parameters.

Integer Overflow Trap

The *TrapInfo* record must have one extra field, *SubCode*. *SubCode* (word #5) contains one of the following codes, which tells what kind of integer overflow occurred.

SubCode Value	Type of Overflow
1	32/64-bit overflow
2	16-bit overflow
3	8-bit overflow
4	overflow on conversion from a compatibility-mode floating-point number
5	overflow on conversion from an IEEE floating-point number

Decimal Overflow Trap

The *TrapInfo* record must have one extra field, *SubCode*.

SubCode (word #5) contains one of the following codes, which tells what kind of decimal overflow occurred.

Subcode Value	Type of Overflow
1	overflow in decimal arithmetic operation
2	overflow in conversion from ASCII to decimal

Invalid ASCII Digit and Invalid Decimal Digit

The *TrapInfo* record has three extra fields:

1. *Subcode* (word #5) contains a code 0..3. Refer to the *Trap Handling Programmer's Guide* for more information.
2. *Address* (word #6) contains the address of the first digit of the number
3. *Count* (word #7) contains the digit count

IEEE Floating Point Traps

The *TrapInfo* record has six extra fields:

1. *Status* (word #5) contains the value in the status register of the IEEE floating-point coprocessor. Any change in this field is reflected in the value of the status register when the program resumes execution.
2. *Operation* (word #6) contains one of the following codes, which tells the type of floating-point operation that caused the trap.

Value	Type of Operation
3	ABS
4	SQRT
5	RND
8	CNVFF
9	CNVXF
10	CNVFX
16	CMP
24	ADD
25	SUB
26	MPY
27	DIV
28	REM

3. *Format* (word #7) contains the type of the operands (single, double, or quadruple). If the operation was CONVERT (CNV*xx*), then the following values are returned:

Value	Types of Operands
1	Source is single, result is double
3	Source is single, result is quadruple
4	Source is double, result is quadruple

   If the operation was NOT a CONVERT (CNV*xx*), then the following values are returned:

Value	Type of Operand
0	Single
1	Double
3	Quadruple

4. *source_op1_ptr* (word #8) contains the address of the first operand, which can be a single-, double- or quadruple-word floating-point number, depending on the operation and the format.
5. *source_op2_ptr* (word #9) contains the address of the second operand, which can be a single-, double-, or quadruple-word floating-point number, depending on the operation and the format.
6. *result_ptr* (word #10) contains the address of the result of the operation, which can be a single-, double-, or quadruple-word floating-point number depending on the operation and the format. You can examine and replace the contents of the area referenced by *result_ptr*, and the Trap Subsystem will ensure that the change is reflected in the appropriate place.

Compatibility Mode Floating-Point Traps

The *TrapInfo* record has one extra field, *Result_ptr*.

Result_ptr (word #5) contains the address of the result of the operation, which can be a single- or double-word floating-point number, depending on the type of trap. You can examine and replace the contents of the area referenced by *result_ptr*, and the Trap Subsystem will ensure that the change is reflected in the appropriate place.

Example

```
{user declares the following Pascal record for the TrapInfo record}

TYPE
   real_ptr = real; long_ptr = longreal;
   TrapInfo = RECORD
          { 1}  instruction,
          { 2}  pc_offset,
          { 3}  pc_space,
          { 4}  error_code,
          { 5}  status,
          { 6}  operation,
          { 7}  format       : integer;
          { 8}  source1_ptr,
          { 9}  source2_ptr,
          {10}  result_ptr   : localanyptr;
              END;
CONST
   IEEE_mask = hex('0007C000');
   fdiv_zero = hex('00002000'); {the error code for fl. pt. div. by 0}

{trap handler routine}

PROCEDURE IEEE_trap_handler (VAR Info : TrapInfo);
VAR
  long_res_ptr : long_ptr;
  real_res_ptr : real_ptr;
```

(*Example continued on next page.*)

```
CONST
  max_real = 3.402823E+38;
  max_longreal = 1.797693L+308;

BEGIN {IEEE_trap_handler}
  {handle only divide-by-zero, ignore others}

  WITH Info DO
     IF (Error_Code = fdiv_zero) THEN
        BEGIN {divide by zero}

        {change the value of the result}
        IF (format = 0) THEN
           BEGIN {real operation}
           real_res_ptr := result_ptr;
           real_res_ptr^ := maxreal;
           END   {real operation}
        ELSE IF (format = 1) THEN
           BEGIN {longreal operation}
           long_res_ptr := result_ptr;
           long_res_ptr^ := maxlongreal;
           END;  {longreal operation}
        END;  {divide by zero}
END;  {IEEE_trap_handler}

{user main program}
VAR
  l1, l2, l3 : longreal;
  oldmask,
  oldplabel  : integer;

BEGIN  {main program}
  ARITRAP (1);   {see "ARITRAP and HPENBLTRAP Intrinsics" for details}
  XARITRAP (IEEE_mask, BAddress (IEEE_trap_handler), oldmask, oldplabel);

  l1 := 233.0;
  l2 := 0.0;
  l3 := l1/l2;  {oops! divide by zero!}

  writeln (l3); {the trap handler should have fixed the result of the
                 previous operation to maxlongreal (1.79769e+308)}
END.  {main program}
```

XCONTRAP Intrinsic

The MPE intrinsic XCONTRAP specifies a user-defined routine (Subsystem Break Handler) that will be called when the user enters a subsystem break (CONTROL Y) on the keyboard. When XCONTRAP is enabled and the user enters CONTROL Y:

- Program control is transferred to the specified user-defined routine.
- The subsystem break function is temporarily disabled to reduce the chance of race conditions.

If normal program execution is to resume after the interrupt, the user-defined routine must re-enable the subsystem break by calling the intrinsic RESETCONTROL just before it ends. On MPE/iX, a normal exit from the user-defined routine is sufficient to return control to the point in the program where the subsystem break was trapped.

Syntax

To arm your Subsystem Break Handler, call XCONTRAP this way:

`XCONTRAP (`*plabel, oldplabel*`);`

```
Call RESETCONTROL this way:

RESETCONTROL;
```

Declare XCONTRAP and RESETCONTROL this way:

```
PROCEDURE XCONTRAP; INTRINSIC;

PROCEDURE RESETCONTROL; INTRINSIC;
```

Parameters

oldplabel — A 32-bit integer, passed by reference, in which the old value of *plabel* is returned. If the subsystem break handler is not armed, this value is zero.

plabel — A 32-bit integer, passed by value, which is the address of your Subsystem Break Handler.

Example

The main program is a loop. Whenever the user enters CONTROL Y on the keyboard, control transfers to the procedure `control_y_handler`, which writes the current loop counter value, then re-enables the subsystem break, and returns to the point in the loop where the interrupt occurred.

```
PROGRAM control_y_test (output);

VAR
       count : integer;
           i : integer;
   oldplabel : integer;

{Intrinsic Declarations}

PROCEDURE XCONTRAP; INTRINSIC;
PROCEDURE RESETCONTROL; INTRINSIC;

{User-defined Subsystem Break Handler}

PROCEDURE control_y_handler;
BEGIN
   writeln('<Control-Y>: Count = ', count:1);  {write counter value}
   RESETCONTROL;  {re-enable subsystem break}
END;

BEGIN
   {Arm the Subsystem Break Handler,
    specifying control_y_handler as the user-defined routine}

   XCONTRAP (BAddress (control_y_handler), oldplabel);

   {Loop}

   FOR i := 1 TO 30000000 DO
      count := i;
END.
```

If you compile, link, and run the preceding program on an MPE/iX system and press CONTROL Y several times while it is running, the program prints the value of `count` each time you press CONTROL Y. For example:

```
CONTROL Y: Count = 121765

CONTROL Y: Count = 2731435

CONTROL Y: Count = 5789345

CONTROL Y: Count = 10135467

CONTROL Y: Count = 23618560
```

HP TOOLSET/XL and HP Symbolic Debuggers

The HP TOOLSET/XL debugger is available on the MPE/iX operating system. The HP Symbolic Debugger is available on both the HP-UX and MPE/iX operating systems. The HP TOOLSET/XL debugger supports a subset of HP Pascal features. The HP Symbolic Debugger supports the HP Pascal language.

To debug your program with HP TOOLSET/XL or HP Symbolic Debugger, you must compile it with the compiler option SYMDEBUG. SYMDEBUG causes the compiler to generate the symbolic debug information that either debugger needs.

HP TOOLSET/XL and HP Symbolic Debugger need different information; if you compile part of your program for HP TOOLSET/XL and part of it for HP Symbolic Debugger, neither HP TOOLSET/XL nor HP Symbolic Debugger will work with it.

For more information on the SYMDEBUG compiler option, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation. For information on HP TOOLSET/XL, refer to the *HP TOOLSET/XL Reference Manual*. For more information on HP Symbolic Debugger, refer to the *MPE/iX Symbolic Debugger User's Guide*.

System Debuggers

The compiler listing of your program is an indispensable debugging aid. The following compiler options provide the listing with additional information, as noted.

The system debuggers are adb on HP-UX and NM Debug on MPE/iX.

Compiler Option	Effect
CODE_OFFSETS	For the main program and each routine, the CODE_OFFSETS option produces a table for every executable statement in which the value of the program counter for the first machine instruction that corresponds to the statement appears beside the statement number. The tables appear at the end of the compiler listing. Each program counter value is offset from the entry point of the procedure that contains the statement to which it corresponds. Program counter values are useful when debugging your program.
LIST_CODE	This option produces a mnemonic listing of the object code for each routine in the program. The mnemonic listing appears after the listing of the compilation unit.
TABLES	This option produces an identifier map for each routine and main program that the compiler parsed while the option was ON. An identifier map shows each identifier that the block declares and its class, type, address or constant value, size, alignment, and (if appropriate) field offset.

Note Program counter values are not exact when you use optimization.

See the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for more information on the compiler options CODE_OFFSETS, LIST_CODE, and TABLES.

You must debug your code before you compile it with optimization. CODE_OFFSETS and SYMDEBUG cannot be used in an optimized program, because the optimizer transforms the machine code so that the mapping of source code to machine code is not one-to-one.

12

The Optimizer

The optimizer is an optional part of the compiler that modifies your program so that it uses machine resources more efficiently, using less space and running faster.

This chapter explains:

- When and how to use the optimizer.
- Level one and level two optimization.
- Optimizer assumptions and what cannot be optimized.
- How to write code that is easily optimized.
- What to do if your optimized program fails.

In examining the examples in this chapter, please note:

- The optimizer operates on machine code rather than source code, but the examples use source code wherever possible because it is easier to understand. Transformations that cannot be shown at the source code level are shown in mnemonic assembly language.
- The optimizer's effectiveness depends strongly on interaction between transformations, but each example illustrates a single transformation for the sake of clarity.

When to Use the Optimizer

Compile your program with optimization only after you have debugged it. The optimizer can transform error-free programs only.

A warning indicates a possible source of run-time errors. If compiling your program produces warnings, do one of the following:

- Be sure that your program will never satisfy the conditions that the warnings specify (see the next example).
- Change your program so that compiling it does not produce warnings.

When you request optimization, the compiler issues warnings for local variables that are used before they are initialized.

Example

```
FUNCTION f (p : integer) : integer;

VAR
   v : integer;

BEGIN
   f := p;

   IF p < 0 THEN
      f := v;
END;
```

The preceding program causes the compiler to issue the warning `ACCESSED, BUT NOT INITIALIZED (535)`, which applies to the variable `v`. A run-time error occurs if `v` is accessed. If you are sure `p` will never have a value less than zero, you can ignore the warning, because the run-time error will never occur.

If you are sure that your program will never produce an out-of-range error, specify RANGE OFF before compiling it with optimization. When the compiler does not generate range-checking code, it compiles faster and can perform more optimizations. The compiled program runs faster without range-checking code, whether or not it is optimized. (See the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation, for information on the RANGE compiler option.)

Once you have compiled your program with optimization, you cannot use the symbolic debugger to debug it, because:

- Debug information will be missing from it (the compiler cannot generate debug information and perform optimizations at the same time).
- Level two optimization radically reorders the code, sometimes keeping in registers the values of variables that you may want to examine.

Invoking the Optimizer

You can invoke the optimizer in the source code or in the compiler command. To invoke the optimizer in the source code, use the OPTIMIZE compiler option as follows:

For	Use the Form
Level one optimization	`OPTIMIZE 'LEVEL1'`
Level two optimization	`OPTIMIZE 'LEVEL2'` or `OPTIMIZE ON`

(For more information on the OPTIMIZE compiler option, see the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.)

To invoke the optimizer in the compiler command on the MPE/iX operating system, include the OPTIMIZE compiler option in the INFO string (see Appendix A).

To invoke the optimizer in the compiler command on the HP-UX operating system, append one of the following options to the compiler command (see Appendix B).

For	Append the Option
Level one optimization	`+O1`
Level two optimization	`+O2` or `-O`

Basic Blocks

The compiler behaves differently on large procedures when you optimize 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 for that procedure:

```
Optdriver: <num> basic blocks; dropping to level 1 optimization for <proc>. (6059)
```

You can request Level 2 optimization and change the number of basic blocks at which the optimizer drops down to Level 1 optimization by specifying *num* in the compiler option:

`$OPTIMIZE 'BASIC_BLOCKS` *num*`'$`

You can change the default level (500) at which the optimizer drops down to Level 1 optimization (without requesting any level of optimization) by specifying *num* in the compiler option:

`$OPTIMIZE 'BASIC_BLOCK_FENCE` *num*`'$`

For more information on basic blocks, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation.

Level One Optimization

Level one optimization transforms small sections of code quickly, using little compile-time storage. Compile your program with level one optimization when you want to improve run-time performance with little increase in compile time.

The functions of the five level one optimizer transformations are:

- Branch optimization
- Dead code elimination
- Faster register allocation (including copy elimination)
- Instruction scheduling
- Peephole optimization

Branch Optimization

The branch optimization transformation makes branch instruction sequences more efficient wherever possible.

Table 12-1 gives examples of equivalent unoptimized and optimized branch instruction sequences.

Table 12-1. Unoptimized and Optimized Branch Instruction Sequences

Unoptimized Sequence	**Optimized Sequence**

Branch target is the default, as in:
`IF b THEN GOTO 100;`
`100 : writeln('Hi');`
	Branch is deleted:
	`100 : writeln('Hi');`

Branch target is an unconditional branch, as in:
`IF b1 THEN`
`   IF 2<5 THEN p(5);`
	Target of unconditional branch is target of conditional branch:
	`IF b1 THEN p(5);`

Target of an unconditional branch at the bottom of a loop is a conditional branch at the top of a loop, as in:
`100 : IF b THEN BEGIN`
`      .`
`      .`
`      .`
`      GOTO 100;`
`      END;`
	Conditional branch at the bottom of the loop:
	`IF b THEN BEGIN`
	` 100 :`
	`      BEGIN`
	`          .`
	`          .`
	`          .`
	`         IF b THEN GOTO 100;`
	`      END;`
	`END;`

Table 12-1.
Unoptimized and Optimized Branch Instruction Sequences (continued)

<table>
<tr><th>Unoptimized Sequence</th><th>Optimized Sequence</th></tr>
<tr><td>Target of unconditional branch is a routine exit, as in:
<pre>
PROCEDURE p;
BEGIN
   .
   .
   .
   GOTO 99;
   .
   .
   .
99: END;
</pre></td><td>Unconditional branch is an exit sequence, if possible. (This cannot be shown at the source code level.)</td></tr>
<tr><td>Branch over a single instruction, as in:
<pre>
IF b THEN GOTO 1;
i := 0; {single instruction}
1: j := j+1;
</pre>
Which has the machine code:
<pre>
LDB   "b",r1
BB,>=,N r1,31,false_if
B,N   label_1

false_if STW   r0,"i"

label_1 LDW   "j",r31
         ADDIO 1,r31,r19
         STW   r19,"j"
</pre></td><td>Conditional nullification of the instruction preceding the skipped instruction:
<pre>
IF NOT b THEN i:=0;
1: j := j+1;
</pre>
Which has the machine code:
<pre>
LDB   "b",r1
EXTRS,< r1,31,1,r0
STW   r0,"i"
LDW   "j",r31
ADDIO   1,r31,r19
STW   r19,"j"
</pre></td></tr>
<tr><td>Conditional branch over an unconditional branch, as in:
<pre>
IF b THEN GOTO 100;
GOTO 110;
100: writeln('Hi');
110: writeln('Bye');
</pre></td><td>The condition in the conditional branch is inverted, and the unconditional branch is deleted:
<pre>
IF NOT b THEN GOTO 110;
writeln('Hi');
110: writeln('Bye');
</pre></td></tr>
</table>

Dead Code Elimination

The dead code elimination transformation eliminates code that will never be executed. For example:

```
a := 2;
goto 1;
writeln('debug_patch_01',a);
1:
```

becomes:

```
a := 2;
```

Do not depend on dead code because the compiler can eliminate dead code even without optimization. The current compiler performs the following transformation without optimization:

The code:

```
IF 2>3 THEN a ELSE b;
WHILE 2>3 DO c;
FOR i := 7 TO 0 DO d;
CASE 1 OF
   1: e;
   2: f;
END;
REPEAT g UNTIL 3>2;
```

becomes:

```
b;
e;
g;
```

Faster Register Allocation

The faster register allocation transformation:

- Inserts entry and exit code.
- Generates code for operations (such as multiplication and division).
- Eliminates unnecessary copy instructions.
- Allocates actual registers to the dummy registers in instructions.

The faster register allocation transformation analyzes register use faster than the coloring register allocation transformation (at level two) does.

Instruction Scheduling

The instruction scheduling transformation:

- Reorders the instructions in a basic block to minimize waiting. (A basic block is an instruction sequence that can be entered only at the first instruction and exited only from the last instruction.)
- Follows a branch instruction with a useful instruction that can be executed as the branch occurs, where possible.
- Schedules floating-point instructions.

Example

The code:

```
LDW    -52(0,30),r1  }
ADDI   3,r1,r31      }  4 machine cycles
LDI    10,r19        }
```

LG200009_124

in which ADDI must wait an extra machine cycle for `r1` to be loaded with its new value, becomes:

```
LDW    -52(0,sp),r1  }
LDI    10,r19        }  3 machine cycles
ADDI   3,r1,r31      }
```

LG200009_125

which does not immediately use `r1` and, therefore, does not have to wait.

Peephole Optimization

The peephole optimization transformation makes one pass through the program, shortening instruction sequences in small windows of code by:

- Changing the addressing modes of instructions, so that they use shorter sequences.
- Substituting smaller, equivalent instruction sequences.

Example

The code:

```
LDI      32,r3
AND      r1,r3,r2
COMIB,=  0,r2,L1   {COMpare Immediate; Branch if equal}
```

becomes:

```
BB,>=    r1,26,L1  {Branch on Bit 26}
```

Real Expression Folding

The real expression folding transformation simplifies real expressions as follows:

Real Expressions of the Form:	Become:
r1 * 1	r1
1 * r1	
r1 + 0	
0 + r1	
r1 - 0	
r1 / 1	
r1 * 0	0
0 * r1	
0 - r1	-r1

Note Folding real expressions violates the IEEE Real Standard, which disallows operations for certain real values (for example, infinity and NAN). The real expression folding transformation assumes that `r1` does not have any of these values. The above expressions are not folded if optimization is turned off.

Level Two Optimization

Level two optimization transforms each routine as a unit, causing the compiler to use more compile-time storage and take longer than level one optimization or no optimization would. Compile your program with level two optimization when you want maximum run-time performance and compilation speed is not important.

The functions of the seven level two optimizer transformations are:

- Coloring register allocation.
- Induction variable elaboration and strength reduction.
- Common subexpression elimination.
- Constant folding.
- Loop invariant code motion.
- Store-copy optimization.
- Unused definition elimination.

Coloring Register Allocation

The coloring register allocation transformation:

- Inserts entry and exit code.
- Generates code for operations (such as multiplication and division).
- Eliminates unnecessary copy instructions.
- Allocates actual registers to the dummy registers in instructions.

The coloring register allocation transformation analyzes register use more slowly than the faster register allocation transformation (at level one), because it must handle the more extensive register usage caused by level two optimizations.

Example

The code:

```
LDI     2,r104
COPY    r104,r103
LDO     5(r103),r106
COPY    r106,r105
LDO     10(r105),r107
```

becomes:

```
LDI     2,r25
LDO     5(r25),r26
LDO     10(r26),r31
```

Induction Variable Elaboration and Strength Reduction

The induction variables and strength reduction transformation:

- Replaces loop counters with induction variables where appropriate (in the following example, the loop counter `i` is replaced by an offset into the array `r`).
- Substitutes addition for multiplication where possible.

Example

The code:

```
FOR i := 1 TO 10000 DO BEGIN
   r[i] := i * k;
END;
```

becomes:

```
t1 := k;
FOR i := 1 TO 10000 DO BEGIN
   r[i] := t1;
   t1 := t1 + k;
END;
```

Common Subexpression Elimination

The common subexpression elimination transformation identifies expressions that appear more than once and have the same result, computes the result, and substitutes it for each occurrence of the expression.

Example

The code:

```
a := x + y + z;
b := x + y + w;
```

becomes:

```
t1 := x + y;
a := t1 + z;
b := t1 + w;
```

Constant Folding

The constant folding transformation replaces constant expressions with their values within basic blocks.

Example

The code:

```
a := 1;
b := 2;
c := a + b;
```

becomes:

```
a := 1;
b := 2;
c := 3;
```

Loop-Invariant Code Motion

The loop-invariant code motion transformation moves loop-invariant code out of the loop (loop-invariant code is code whose effect is independent of the value of the loop counter).

Example

The code:

```
FOR i := 1 TO 100 DO BEGIN
   a[i] := (4 * x) + i;
END;
```

becomes:

```
t1 := 4 * x;
FOR i := 1 TO 100 BEGIN
   a[i] := t1 + i;
END;
```

Because optimization affects the machine code, but not the source code, error messages associated with loop-invariant source code inside the loop appear outside the loop after optimization.

Example

Unoptimized program:

```
i := 1;
REPEAT
   a[i] := i;
   b[i] := b[i] + x/y;
UNTIL i = 10;
```

Optimized program:

```
i := 1;
t := x/y;
REPEAT
   a[i] := i;
   b[i] := b[i] + t;
UNTIL i = 10;
```

If `y` is zero, the unoptimized program causes an error within the loop, at the assignment of `b[1]`. The optimized program causes an error before the loop is entered, before `b[1]` is assigned a value. This error occurs before the program enters the loop.

Store-Copy Optimization

The store-copy optimization transformation substitutes registers for memory locations where possible, by replacing store instructions with copy instructions and deleting load instructions.

Example

Source code:

```
t1 :=  x + y;
a  := t1 + z;
b  := t1 + w;
```

Unoptimized code:

```
LDW "x",r104
LDW "y",r105
ADD r104,r105,r106
STW r106,"t1"
LDW "t1",r106
LDW "z",r107
ADD r106,r107,r108
STW r108,"a"
LDW "t1",r106
LDW "w",r109
ADD r106,r107,r110
STW r110,"b"
```

Optimized code:

```
LDW "x",r104
LDW "y",r105
ADD r104,r105,r106


LDW "z",r107
ADD r106,r107,r108
STW r108,"a"

LDW "w",r109
ADD r106,r107,r110
STW r110,"b"
```

Unused Definition Elimination

The unused definition elimination transformation removes unused memory location and register definitions (which are often the by-products of other optimizations).

Example

The function:

```
FUNCTION f (x : integer) : integer;
VAR
   a,b,c : integer;
BEGIN
   a := 1;
   b := 2;
   c := 0;
   c := x * b;
   f := c;
END;
```

becomes:

```
FUNCTION f (x : integer) : integer;
VAR
   a,b,c : integer;
BEGIN
   b := 2;
   c := x * b;
   f := c;
END;
```

All the level two optimizations combined produce:

```
FUNCTION f (x : integer) : integer;
VAR
   a,b,c : integer;
BEGIN
   f := x * 2;
END;
```

Optimizer Assumptions

The optimizer makes the following assumption about variable use when it optimizes a program: inside a routine, the only variables that can be accessed indirectly (through a pointer or by another function) are:

- Global variables.
- Reference parameters.
- Local variables that are passed to other routines by reference.
- Local variables or value parameters that are passed to other routines by reference.
- Local variables or value parameters used by the predefined function *addr*. You will violate this assumption if you use *baddress* or *waddress*.

If your program violates this assumption, it will fail when optimized.

You can make the optimizer's job easier by telling it what other assumptions it can make. To do this, use the compiler option ASSUME (refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation).

You can derive the assumptions for your program without the overhead of level two optimization by following these directions:

1. Specify level one optimization. The compiler does not collect information that only level two requires, but it does collect sufficient information about the source program to issue notes that tell you which assumptions you can add to forward and external declarations to make your program easier to optimize.
2. If you only want to see warnings and not notes, use NOTES OFF (for more information on the NOTES compiler option, refer to the *HP Pascal/iX Reference Manual* or the *HP Pascal/HP-UX Reference Manual*, depending on your implementation). NOTES OFF does not suppress warnings, which the compiler issues if assumptions in routine headings are invalid. (Chapter 11 explains the difference between notes and warnings.)

Writing Programs That Are Easily Optimized

To maximize the opportunities for optimization and the speed with which your program is optimized, observe the following guidelines whenever possible:

- Specify all possible assumptions with the ASSUME compiler option.
- Do not use system programming extensions (specify ASSUME 'PASCAL_FEATURES').
- Use local variables and parameters instead of global variables (this helps the optimizer promote variables to registers).
- Don't use CRUNCHED, PACKED, or *shortint* variables.
- Don't put pointer fields in variant parts of records.
- Use unique types as often as possible (this helps the optimizer determine which variables are possible actual parameters for the formal parameters of routines).
- Make routines relatively small—approximately 50 executable lines maximum (this allows the optimizer to use registers more efficiently).
- Expand in-line any routine of fewer than five lines and any relatively small routine that your program calls only once (see "INLINE" in Chapter 8).
- Do not write a loop that contains only a routine call; put the loop in the routine instead, or use the INLINE compiler option.
- Design loops that terminate when their control values are zero or *nil*.
- Use WITH statements.
- Do not use the FORWARD and EXTERNAL directives without the ASSUME compiler option.
- Do not use the GOTO statement.

What to Do If the Optimized Program Fails

Occasionally, a program works differently after optimization. If your program fails after optimization:

1. Make sure that the program does not violate the optimizer assumptions. Check for program errors that may not occur without optimization, such as uninitialized variables and misaligned data.
2. If the program violates the optimizer assumptions, either correct and recompile it, or recompile it without optimization.
3. If the program does not violate the optimizer assumptions, isolate the problem code and recompile it with level one optimization. If the program still fails, recompile it without optimization.

On the HP-UX operating system, if the compiler runs out of space during optimization:

1. Split your program into smaller compilation units.
2. Run fewer concurrent processes when compiling with optimization.
3. Change the system configuration, increasing the amount of swap space on the disk.

A

MPE/iX Dependencies

This appendix explains how the HP Pascal compiler and HP Pascal programs work on the MPE/iX operating system. It explains:

- How MPE/iX affects system dependent HP Pascal features.
- MPE/iX extensions to HP Pascal.
- How to compile, link, and run your HP Pascal program on MPE/iX.

System-Dependent Features

System dependent features are available to all HP Pascal programs (regardless of the system on which the compiler is running), but the system affects their definitions and behavior. System dependent HP Pascal features fall into these categories:

- Compiler options.
- File names.
- Associating logical and physical files.
- Using file equations.
- Default file attributes.
- Standard modules.
- Miscellaneous.

Compiler Options

The following compiler options are available only to programs that are compiled by the HP Pascal compiler running on the MPE/iX operating system and contain the compiler option OS 'MPE/XL'.

```
FONT
HP3000_16
HP3000_32
```

The compiler option INCLUDE is available to programs compiled by the HP Pascal compiler running on either the MPE/iX or HP-UX operating system, but it works differently on the two systems.

Refer to the *HP Pascal/iX Reference Manual* for more information on the compiler options FONT, HP3000_16, HP3000_32, and INCLUDE.

File Names

An MPE/iX file name has the syntax

filename [/*lockword*][.*group*[.*account*]][:*nodename*]

where each of *filename*, *lockword*, *group*, *account* and *node* is a string of one to eight alphanumeric characters. The first character in the string is a letter, and each of *domain* and *organization* is a string of one to 16 alphanumeric characters, the first of which is a letter. The entire file name cannot have more than 86 characters. MPE/iX does not distinguish between uppercase and lowercase letters.

Example

```
myfile   }
Myfile   } Equivalent
MYFILE   }
myfile/secret
myfile/secret.mktg
myfile.mktg.acct1
myfile/secret.mktg.acct1:node4
myfile:node4.d10
myfile/secret.mktg.acct1:node4.d10.HP
myfile.mktg.acctt1:node4.d10.HP
myfile.mktg.acct1:node4.d10
myfile.mktg:node4
myfile.mktg.acct1
myfile.mktg
```

LG200009_123

For more information on MPE/iX file names, refer to the *MPE/iX Commands Reference Manual*.

Associating Logical and Physical Files

Your program does not affect its external environment unless its logical files are associated with physical files at run-time. If they are, file operations work concurrently on logical and physical files (see Chapter 3).

In HP Pascal on the MPE/iX operating system, a logical file is associated with a physical file under any one of the following conditions:

1. The name of the logical file is both a program parameter and the first parameter of a predefined file-opening procedure. The predefined file-opening procedure has no second parameter.

   The operating system associates the logical file name with a default physical file, whose name consists of the first eight characters of the logical file name. This name must be an acceptable MPE/iX file name (for example, it cannot contain an underscore character (_)). If the default physical file does not exist, HP Pascal creates a temporary physical file with that name.

Example

```
PROGRAM case_one (input,output,file1);

VAR
   file1 : FILE OF integer;

BEGIN
  reset(file1);
   .
   .
   .
END.
```

The operating system associates the logical file `file1` with the physical file `FILE1`. If `FILE1` does not exist, HP Pascal creates a temporary file named `FILE1`.

The standard files *input* and *output* are exceptions to this scheme. When they are program parameters, the operating system associates them with the physical files `$STDIN` and `$STDLIST`, respectively.

If a logical file name is not a program parameter, but is the first parameter of a file-opening procedure that has no second parameter, the operating system associates the logical file with a temporary, nameless physical file (assuming that the logical file is not already associated with a physical file). You cannot save the temporary file. When the program ends or the logical file is associated with another physical file, the temporary file is inaccessible.

2. The names of the logical and physical files are the first and second parameters, respectively, of a predefined file-opening procedure. It does not matter whether the logical file name is a program parameter or not.

Example

```
PROGRAM case_two (input,output);  {logical file name is not a
                                   program parameter}
VAR
   file1 : FILE OF integer;

BEGIN
   rewrite(file1,'numfile');
     .
     .
     .
END.
```

The operating system associates the logical file `file1` with the physical file `numfile`.

This association holds, even if the logical file name is a program parameter.

Example

```
PROGRAM case_three (input,output,file1);  {logical file name is a
                                           program parameter}
VAR
   file1 : FILE OF integer;
   fname : PACKED ARRAY [1..8] OF char;

BEGIN
   fname := 'numfile';
   rewrite(file1,fname);
      .
      .
      .
END.
```

The operating system still associates `file1` with `numfile`, not `FILE1`.

The second parameter of a file-opening procedure need not be a string literal. It can also be a PAC variable or string expression.

Using File Equations

The MPE/iX FILE command redirects the association of one physical file to another physical file and specifies additional file attributes, which are MPE/iX dependent.

Example

```
PROGRAM prog (outfile);

VAR
   i : integer;
   outfile : text;

BEGIN
   rewrite(outfile);
   FOR i := 1 TO 20000 DO
      writeln(outfile,i);
END.
```

If `PRG` is the program file for `prog` and you execute the MPE/iX command sequence

```
:FILE OUTFILE = FILE2
:RUN PRG
```

then output goes to `FILE2` instead of `OUTFILE`.

If you execute the MPE/iX command sequence

```
FILE OUTFILE; DISC=21000; REC=-20,,F,ASCII
RUN PROG
```

then a nondefault attribute file is created.

Default File Attributes

When HP Pascal creates a file, the physical file attributes depend on the file component type.

Table A-1 gives the default file attributes of files built by HP Pascal programs. After the program has executed, the MPE/iX command LISTF shows these values for the files that the program built (LISTF attribute names are in parentheses).

Table A-1. Default File Attributes

How Program Declares File	Default File Attribute			
	Record Size (SIZE)	**File Type (TYP)**	**Current File Size (EOF)**	**Maximum File Size (LIMIT)**
FILE OF type	Component size	Fixed length binary (FB)	Number of components written	1023
Text	256 bytes	Variable length ASCII with carriage control (VAC)	Number of lines written	1023

Standard Modules

Two standard modules are available on MPE/iX: *stdinput* and *stdoutput*.

If a module imports the *stdinput* module, it can use the predefined file *input* in I/O statements such as *read* and *readln*.

If a module imports the *stdoutput* module, it can use the predefined file *output* in I/O statements such as *write* and *writeln*.

Example

```
MODULE mymod;
IMPORT
   stdinput, stdoutput;

EXPORT
   FUNCTION myproc : integer;

IMPLEMENT
   FUNCTION myproc : integer;
   VAR
      i : integer;
   BEGIN
      prompt('enter number:');  {need not specify output file}
      readln(i);                {need not specify input file}
      myproc := i;
   END;
END.
```

Additional Features

The HP Pascal features in the left-hand column depend on the MPE/iX operating system in the ways explained in the right hand column.

Feature	MPE/iX Dependency
Close options	The optional third parameter of the predefined procedure *close* can be SAVE, LOCK, TEMP, NORMAL, CRUNCH, or PURGE, whose meanings are: SAVE LOCK — The file is saved as a permanent file after it is closed. TEMP NORMAL — The file is saved as a temporary file after it is closed. CRUNCH — Space after end-of-file marker is removed when the file is closed. PURGE — The file is purged after it is closed.
Halt	MPE/iX calls the intrinsic QUIT with an integer parameter.
Internal table size	The Job Control Word (JCW) PASXDATA is the number of pages to allocate to each internal table (there is one internal table for identifiers and another for structured constants). The default internal table size is 100 pages. To set the internal table size to *n* pages, use the command: `:SETJCW PASXDATA` *n*
Write	If the file being written is `$STDLIST` (the default output file), the output is unbuffered; therefore, a *write* to `$STDLIST` has the same behavior as *prompt*.
Input	The standard program parameter and textfile *input* is `$STDIN`.
Maxpos	The call *maxpos(f)* returns the position number of the last component of the file *f* that the program can access. It is an error if the file *f* is not open for direct access.

Open options	The third parameter of the predefined file-opening procedures *append*, *associate*, *open*, *read*, *reset*, *rewrite*, and *write*. They and their meanings are:

Option	Meaning
CCTL	The file has carriage control. (Ignored for *associate*.)
DIRECT	The file is open for read and write access (*associate* only).
NOCCTL	The file does not have carriage control. (Ignored for *associate*.)
READ	The file is open for read access only (*associate* and *open* only).
WRITE	The file is open for write access only (*associate* and *open* only).
SHARED	The file can be open to more than one program at a time. (Ignored for *associate*.)
EXCLUS	The file cannot be open to more than one program at a time. (Ignored for *associate*.)
LOCK	The file is locked. If the file is already locked, the program waits until it is unlocked. (Ignored for *associate*.)

At least one open option is required for *associate*; for all other file-opening procedures, open options are optional. You can specify more than one open option (separate them with commas).

If the physical file specified in the *associate* procedure has one or more of the characteristics specified by the open options, then the logical file assumes the same characteristics. If not, the *associate* procedure does not associate the new physical file with the logical file.

Output	The standard program parameter and textfile *output* is `$STDLIST`.
System intrinsic file	SYSINTR.PUB.SYS
System default module library	PASLIB.PUB.SYS

Restrictions on Using Executable Libraries (XLs)

Global variables cannot be referenced across load modules. This applies to globals declared through normal, global, external, and module subprogram compilation units. In particular, you cannot use the standard files *input* or *output*.

If a subprogram compilation unit is put in an XL, memory is overwritten. You cannot put an external compilation unit in an XL. Using MODULE or SUBPROGRAM with global compilation units will cause separate storage locations to be allocated.

A non-local GOTO from an XL cannot branch to a label in the outer block.

MPE/iX Extensions

MPE/iX extensions are available only to programs that are run on the MPE/iX operating system or contain the compiler option OS 'MPE/XL'. They are:

- Predefined function *ccode*
- Predefined function *fnum*
- Predefined function *get_alignment*
- Predefined function *statement_number*
- Predefined procedure *setconvert*
- Predefined procedure *strconvert*
- Pascal/V packing algorithm

ccode Function

The predefined function *ccode* returns an integer in the range 0..2, which represents the condition code set by the most recently executed intrinsic or external SPL routine.

The correspondence between possible return values and condition codes is:

Value	Condition Code
0	CCG
1	CCL
2	CCE

For the meanings of the condition codes, refer to the *MPE/iX Intrinsics Reference Manual*.

The value that *ccode* returns is valid between the time that the intrinsic or external SPL routine returns and any subsequent calls that can change the value of *ccode*, which are:

- Another intrinsic or external SPL routine.
- Any predefined routine.
- An HP Pascal error condition.

Note The scope rules for *ccode* are different in MPE/iX and MPE V.

Example

```
PROGRAM prog (output);
PROCEDURE intrin; INTRINSIC;
PROCEDURE extspl; EXTERNAL SPL;
PROCEDURE p;
BEGIN
   writeln(ccode);   {Garbage -- no intrinsic or external SPL
   intrin;
   writeln(ccode);   {Returns condition code that intrin set}
   extspl;
   writeln(ccode);   {Returns condition code that extspl set}
END;
BEGIN
   p;
END.
```

Fnum Function

The predefined function *fnum* returns the MPE/iX file number of the physical file currently associated with a given logical file. You can use this file number in calls to MPE/iX file system intrinsics.

Syntax

fnum (*filename*)

Parameter

filename The name of the logical file. This parameter is required, even if the logical file is the standard file *input* or *output*. The logical file must be associated with a physical file.

Example

```
PROGRAM aaa (output,f);

VAR
   f : text;
   file_number : integer;
   file_name : PACKED ARRAY [1..86] OF char;

PROCEDURE fgetinfo; INTRINSIC;

BEGIN
   reset(f);
   file_number := fnum(f);
   file_name := ' ';
   fgetinfo(file_number,file_name);
   writeln('File name of f is', file_name);
END.
```

Get_alignment Function

The predefined function *get_alignment* returns the alignment requirement of a given type or variable. For a type, *get_alignment* returns the minimum possible alignment. For a variable, it returns the actual alignment.

Syntax

```
                 { variable }
get_alignment (  {          } )
                 { type     }
```

Parameters

variable	Any variable. The function *get_alignment* returns its alignment requirement.
type	Any type identifier (the name of any type). The function *get_alignment* returns its alignment requirement.

Example

```
$OS 'MPE XL'$
PROGRAM prog;

TYPE
   Rec = $ALIGNMENT 8$
         RECORD
            f1 : integer;
            f2 : shortint;
            f3 : real;
         END;

   integer_ = $ALIGNMENT 2$ integer;

VAR
   ptr : ^integer_;

BEGIN
   i := get_alignment(rec);
   IF get_alignment(ptr^) <> 2 THEN ...
END.
```

Statement_number Function

The predefined function *statement_number* returns the statement number of the statement that calls it, as shown on the compiled listing. It is a useful debugging aid, especially when used with the predefined procedure *assert*.

Syntax

```
statement_number
```

Example

```
PROGRAM prog (output);

VAR
   i : integer;

BEGIN
   i := statement_number;
   writeln('Current Statement Number is ', i);
   assert(a > b, statement_number);
END.
```

Setconvert Procedure

The predefined procedure *setconvert* converts a set from HP Pascal packing algorithm (HP3000_32) format to Pascal/V packing algorithm (HP3000_16) format, or vice versa. It is enabled by the HP3000_16 compiler option.

Syntax

setconvert(*set1*,*set2*)

Parameters

set1 The name of the set variable to be converted.

set2 The name of the set variable into which the converted set is to be stored.

The sets *set1* and *set2* can vary only in packing algorithm format. Their packing (unpacked, packed, or crunched) and base types must be the same. Their packing algorithm formats cannot be the same.

Example

```
PROGRAM prog;
$HP3000_16$  {Enables setconvert procedure}

TYPE
   hp3000_16_set1 = SET OF char;

   hp3000_32_set1 = $HP3000_32$ SET OF char;
   hp3000_32_set2 = $HP3000_32$ PACKED SET OF char;
   hp3000_32_set3 = $HP3000_32$ SET OF integer;

VAR
   set16_1,
   set16_2 : hp3000_16_set1;

   set32_1 : hp3000_32_set1;
   set32_2 : hp3000_32_set2;
   set32_3 : hp3000_32_set3;

BEGIN
   setconvert(set16_1,set32_1);  {convert from Pascal/V to HP Pascal}
   setconvert(set32_1,set16_1);  {convert from HP Pascal to Pascal/V}

   setconvert(set16_1,set32_2);  {Illegal -- different packings}
   setconvert(set16_1,set32_3);  {Illegal -- different base types}
   setconvert(set16_1,set16_2);  {Illegal -- same packing algorithm format}
END.
```

Strconvert Procedure

The predefined procedure *strconvert* converts a string from Pascal/V packing algorithm (HP3000_16) format to HP Pascal packing algorithm (HP3000_32) format. It is enabled by the HP3000_16 compiler option.

Syntax

`strconvert(`*string1,string2*`)`

Parameters

string1 The name of the string variable to be converted. The string variable must be in Pascal/V packing algorithm (HP3000_16) format.

string2 The name of the string variable into which the converted string is to be stored. The string variable must be in HP Pascal packing algorithm (HP3000_32) format.

Example

```
PROGRAM prog;
$HP3000_16$  {Enables strconvert procedure}

TYPE
   str16_20=string[20];            {Pascal/V packing algorithm (HP3000_16)}
   str32_40=$HP3000_32$ string[40]; {HP Pascal packing algorithm (HP3000_32)}

VAR
   sv32_1,
   sv32_2 : str32_40;

   sv16_1,
   sv16_2 : str16_20;

BEGIN
   strconvert(sv16_1,sv32_1);
   strconvert(sv32_2,sv32_1);         {Illegal}
   strconvert(sv16_1,sv16_2);         {Illegal}
END.
```

Pascal/V Packing Algorithm

The Pascal/V packing algorithm is an alternative to the default HP Pascal packing algorithm that Chapter 5 explains. If you want the compiler to use the Pascal/V packing algorithm, include the compiler option HP3000_16 in your program (see the *HP Pascal/iX Reference Manual* for more information on the compiler option HP3000_16). HP3000_16 causes the compiler to use the Pascal/V packing algorithm, with these exceptions:

- Pointers are allocated four bytes each and are 4-byte-aligned.
- Files are aligned according to the HP Pascal packing algorithm. File control blocks are determined by the HP Pascal packing algorithm. Buffer size is determined by the Pascal/V packing algorithm.
- Variables of types that specify the HP3000_32 compiler option are allocated and aligned according to the HP Pascal packing algorithm.

Unpacked Variables

An *unpacked variable* is either not part of an array or record, or it is part of an unpacked array or record. In either case, it is allocated and aligned the same way.

Table A-2 shows how the Pascal/V packing algorithm allocates and aligns the elements of an unpacked array or the fields of an unpacked record. The element or field types are in alphabetical order. Subsections that Table A-2 references are in this section, “Pascal/V Packing Algorithm”.

Table A-2.
Allocation and Alignment of Unpacked Variables
(Pascal/V Packing Algorithm)

Variable Type	Allocation	Alignment
Array	Use formula in "Arrays"	Byte or 2-byte
Bit16	2 bytes	2-byte
Bit32	4 bytes	2-byte
Bit52	8 bytes	2-byte
Boolean	1 byte	Byte
Char	1 byte	
Enumeration	1-256 elements	
Enumeration	1 byte	Byte
Enumeration	257 or more elements	
Enumeration	2 bytes	2-byte
File	See "Files"	8-byte
Integer	4 bytes	2-byte
Longint	8 bytes	2-byte
Longreal	8 bytes	2-byte
Pointer	HP3000_16 does not affect pointers. See Table 5-1.	
Real	4 bytes	2-byte
Record	Each field is allocated by type and record is padded to nearest 2-byte boundary	2-byte
Set	See "Sets"	
String	See "Strings"	2-byte
Subrange of enumeration	Same as base type	Byte or 2-byte
Subrange of integer	Inside range -32768..32767	
Subrange of integer	2 bytes	2-byte
Subrange of integer	Outside range -32768..32767	
Subrange of integer	4 bytes	2-byte

Packed Variables

A *packed variable* is the element of a packed array or the field of a packed record. Packed elements and packed fields are allocated and aligned differently.

Table A-3 shows how the Pascal/V packing algorithm allocates and aligns the elements of a packed array. The element types are in alphabetical order. Subsections that Table A-3 references are in this section, "Pascal/V Packing Algorithm".

Table A-3. Allocation and Alignment of Packed Array Elements (Pascal/V Packing Algorithm)

Element Type	Allocation	Alignment
Array	Use formula in "Arrays"	Byte if element is allocated 8 bits; 2-byte otherwise
Bit16	2 bytes	2-byte
Bit32	4 bytes	2-byte
Bit52	8 bytes	2-byte
Boolean	1 bit	Bit
Char	1 byte	Byte
Enumeration	See "Packed Enumerations"	
File	See "Files"	8-byte
Integer	4 bytes	2-byte
Longint	8 bytes	2-byte
Longreal	8 bytes	2-byte
Pointer	HP3000_16 does not affect pointers.	
Real	4 bytes	2-byte
Record	Each field is allocated by type and record is padded to nearest 2-byte boundary	2-byte
Set	See "Sets"	
String	See "Strings"	
Subrange of enumeration	See "Packed Subranges of Enumerations"	
Subrange of integer	See "Packed Subranges of Integers"	

Table A-4 shows how the Pascal/V packing algorithm allocates and aligns the fields of a packed record. The field types are in alphabetical order. Subsections that Table A-4 references are in this section, "Pascal/V Packing Algorithm".

Table A-4.
Allocation and Alignment of Packed Record Fields
(Pascal/V Packing Algorithm)

Variable Type	Allocation	Alignment
Array	Use formula in "Arrays"	Byte if element is allocated 8 bits; 2-byte otherwise
Bit16	2 bytes	2-byte
Bit32	4 bytes	2-byte
Bit52	8 bytes	2-byte
Boolean	1 bit	Bit
Char	8 bits	Bit, but does not cross 2-byte boundary
Enumeration	See "Packed Enumerations"	
File	See "Files"	8-byte
Integer	4 bytes	2-byte
Longint	8 bytes	2-byte
Longreal	8 bytes	2-byte
Pointer	HP3000_16 does not affect pointers. See Table 5-1	
Real	4 bytes	2-byte
Record	Each field is allocated by type and record is padded to nearest 2-byte boundary	2-byte
Set	See "Sets"	
String	See "Strings"	
Subrange of enumeration	See "Packed Subranges of Enumerations"	
Subrange of integer	See "Packed Subranges of Integers"	

Arrays

This section applies to the allocation of unpacked and packed arrays. For alignment, see Table A-2 and Table A-3.

The Pascal/V packing algorithm stores arrays in row-major order (for a definition of row-major order, see Chapter 5).

The Pascal/V packing algorithm uses this formula to allocate an array:

(number_of_elements * space_for_one_element)
+
number_of_internal_unused_bits
+
number_of_trailing_pad bits

The space_for_one_element depends on the element type and whether the array is unpacked or packed. If the array is unpacked, find its type in Table A-2. If the array is packed, find its type in Table A-3.

If space_for_one_element is less than 16 bits, the number_of_internal_unused_bits is

16-((16 DIV space_for_one_element) * space_for_one_element)

otherwise, it is zero.

The number_of_trailing_pad_bits is the number of leftover bits in the last byte or word (whichever each element is allocated).

Example

```
TYPE
   day = (sun,mon,tues,wed,thurs,fri,sat);

VAR
   ua : ARRAY [1..8] OF day;
   pa : PACKED ARRAY [1..8] OF day;
```

Each element of ua takes one byte. The entire array takes eight bytes, with no internal unused bits and no trailing pad bits. The array ua is allocated and aligned like this:

ua[1]	ua[2]	ua[3]	ua[4]
ua[5]	ua[6]	ua[7]	ua[8]

LG200173_028

Each element of pa takes three bits. No element can cross a 2-byte boundary, so the bit following pa[5] is unused. The entire array takes four bytes, with one internal unused bit and seven trailing pad bits. It is allocated and aligned like this:

LG200173_029

Files

The HP Pascal compiler allocates space for an HP3000_16 file this way:

- The file control block is allocated according to the HP Pascal packing algorithm.
- The file buffer variable size is allocated according to the Pascal/V packing algorithm.
- The file is 8-byte-aligned.

Records

This section applies to unpacked and packed records unless otherwise noted.

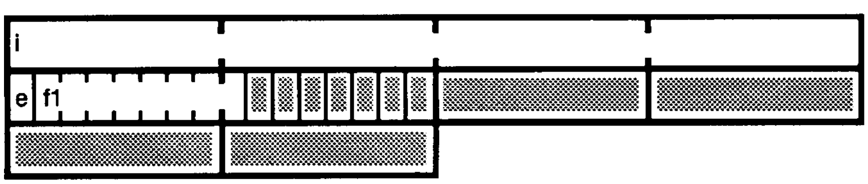
The Pascal/V packing algorithm does not always align variant parts of fields on the same boundary. Each variant part's boundary depends on its type.

Example

```
TYPE
   Rec = PACKED RECORD
         i : integer;
         CASE b : boolean OF
            TRUE : (f1 : char);
            FALSE : (f2 : ARRAY[1..2] OF -32768..32767;
         END;
```

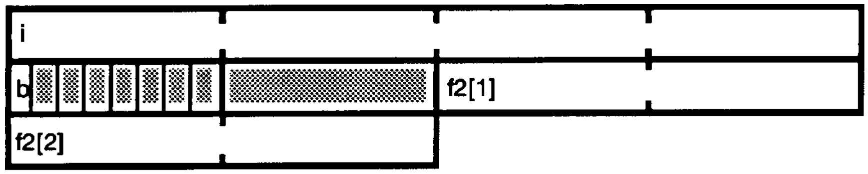
A variable of type Rec is allocated 10 bytes. The TRUE and FALSE variants are aligned like this:

TRUE Variant



LG200173_030

FALSE Variant



LG200173_031

The variants `f1` and `f2` do not start on the same boundary; therefore, `f1` cannot be overlaid with `f2`.

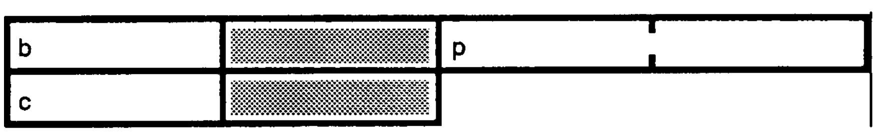
Sometimes you can reduce the space that a record takes by declaring its fields in different order.

Example

```
VAR
   upr1 : RECORD
             b : boolean;
             p : 0..32767;
             c : char;
          END;

   upr2 : RECORD
             b : boolean;
             c : char;
             p : 0..32767;
          END;
```

The only difference between the variables `upr1` and `upr2` above is the order of their fields.

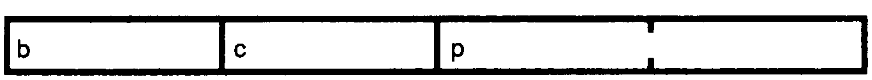
The variable `upr1` takes six bytes:



LG200173_032

Because `p` must be 2-byte-aligned, it cannot start in the second byte. The sixth byte is allocated to `upr1` also, because records are 2-byte-aligned.

The variable `upr2` takes four bytes:



LG200173_033

Sets

The Pascal/V packing algorithm allocates sets in byte pairs. The number of byte pairs allocated to a set depends on its type. For the types Boolean, char, enumeration, and integer, the formula for the number of byte pairs is:

number_of_byte_pairs = `ceil(`*bits_required_for_set*`/16)`

(where `ceil(`x`)` means the integer closest to x that is greater than or equal to x). Table A-5 gives the values for *bits_required_for_set* and *number_of_byte_pairs* for Boolean, char, and integer types.

Table A-5.
Bit and Byte Pair Requirements for Boolean, Char, and Integer Base Types (Pascal/V Packing Algorithm)

Base Type	*bits_required_for_set*	*number_of_byte_pairs*
`Boolean`	2	1
`Char`	256	16
`Integer` †	256 (by default) *	16

† Same as bit16, bit32, bit52, shortint, and longint.

* Integers outside the range 0..255 cannot belong to the set.

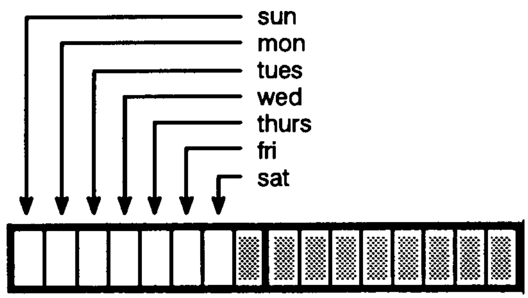
For enumerated sets, *bits_required_for_set* is the number of elements in the set, and you must use the formula to determine *number_of_byte_pairs*.

Example

```
VAR
   days = SET OF (sun,mon,tues,wed,thurs,fri,sat);
   months = SET OF (ja,f,mr,ap,ma,jn,jl,au,s,o,n,d);
   set_33 = SET OF (e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,
                    e12,e13,e14,e15,e16,e17,e18,e19,e20,e21,e22,
                    e23,e24,e25,e26,e27,e28,e29,e30,e31,e32,e33);
```

The set `days` has seven elements and requires seven bits. It is allocated one byte pair (`ceil(7/16) = 1`).

Each element is represented by one bit, like this:



The set `months` has 12 elements and requires 12 bits. It is allocated one byte pair (`ceil(12/16) = 1`). Each element is represented by one bit.

The set `set_33` has 33 elements and requires 33 bits. It is allocated three byte pairs (`ceil(33/16) = 3`). Each element is represented by one bit.

For integer subrange sets, the formula for the number of byte pairs is:

number_of_byte_pairs = (*upper_bound_byte_pair_number* -
lower_bound_byte_pair_number) + 1

The upper bound of the integer subrange determines *upper_bound_byte_pair_number*, and the lower bound determines *lower_bound_byte_pair_number*. The formula is:

byte_pair_number = `floor(`*bound* / 16`)`

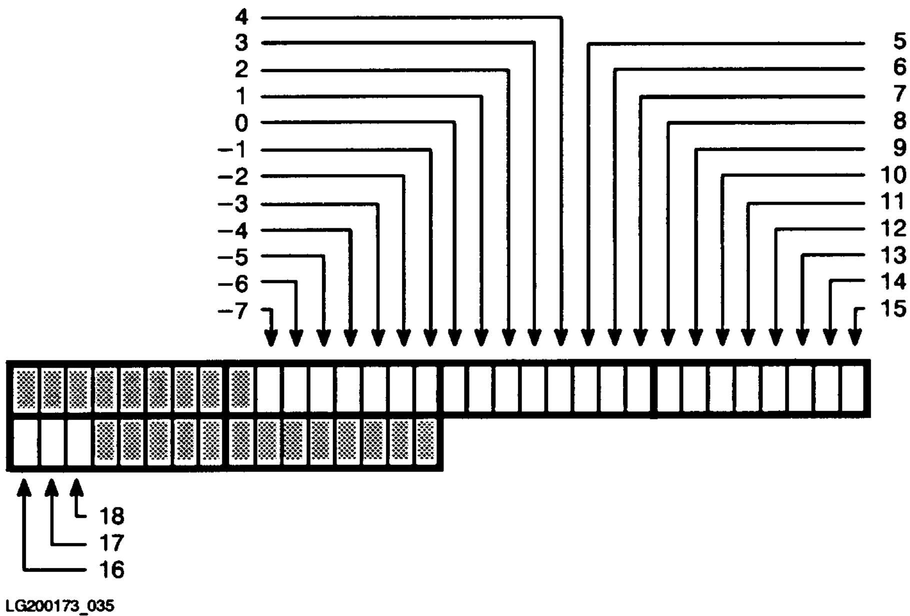
(where `floor(`x`)` means the integer closest to x that is less than or equal to x).

Example

```
VAR
   s : SET OF -7..18;
```

The upper bound of the subrange is 18, so *upper_bound_byte_pair_number* is `1 (floor(18/16)=1)`. The lower bound of the subrange is -7, so *lower_bound_byte_pair_number* is `-1 (floor(-7/16)=-1)`. The set `s` is allocated three byte pairs `((1-(-1))+1=3)`.

Each set element is represented by one bit, like this:



To minimize storage space, avoid base types that are small subranges that overlap byte pair boundaries.

Example

```
VAR
   s : SET OF 31..32;
```

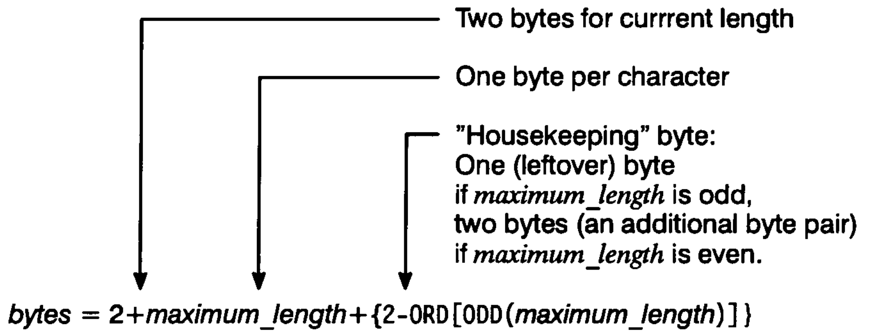
The set `s` takes two byte pairs, using 32 bits to represent a set that requires only two bits. The arithmetic is:

```
floor(32/16) - floor31/16) + 1 = (2-1)+1 = 2.
```

Strings

The Pascal/V packing algorithm aligns strings on 2-byte boundaries. Because the current length (0..32767) is allocated two bytes, four bytes is the smallest possible string allocation.

The formula for the number of bytes allocated to a string is:



LG200009_114

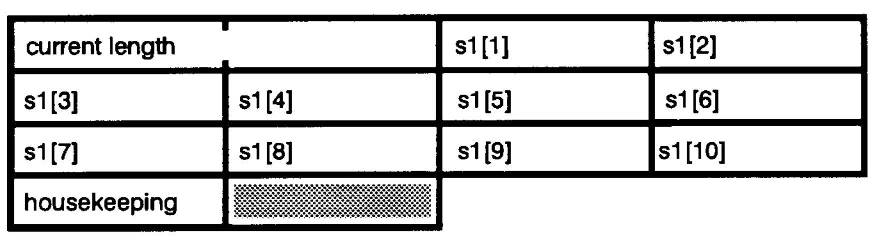
Example

```
VAR
   s1 : string[10];
   s2 : string[7];
```

The string `s1` takes 14 bytes:

```
2+10+{2-ORD[ODD(10)]} =
    12+[2-ORD(FALSE)] =
             12+(2-0) = 14
```

The allocation is:

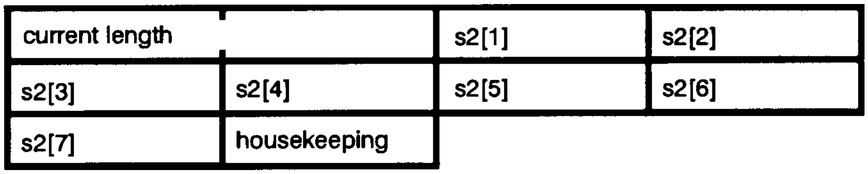


LG200173_036

The string `s2` takes 10 bytes:

```
2+7+{2-ORD[ODD(7)]} =
    9+[2-ORD(TRUE)] =
            9+(2-1) = 10
```

The allocation is:



LG200173_037

Packed Enumerations

This subsection explains how the Pascal/V packing algorithm allocates and aligns packed enumeration variables. A packed enumeration variable is either the element of a packed array or the field of a packed record. The algorithm treats the two cases differently.

Table A-6 shows the relationship between the number of bits that an enumeration element of a packed array requires, the number of bits that the Pascal/V packing algorithm allocates it, and its alignment. A bit-aligned element never crosses a 2-byte boundary.

Table A-6. Allocation and Alignment of Enumeration Elements of Packed Arrays (Pascal/V Packing Algorithm)

Required Number of Bits Per Element	Number of Bits Allocated Per Element	Element Alignment
1	1	Bit
2	2	Bit
3	3	Bit
4	4	Bit
5	5	Bit
6 to 8	8 (1 byte)	Byte
9 to 16	16 (2 bytes)	2-byte

Table A-7 shows the relationship between the number of bits that an enumeration field of a packed record requires, the number of bits that the Pascal/V mapping algorithm allocates it, and its alignment. A bit-aligned field never crosses a 2-byte boundary.

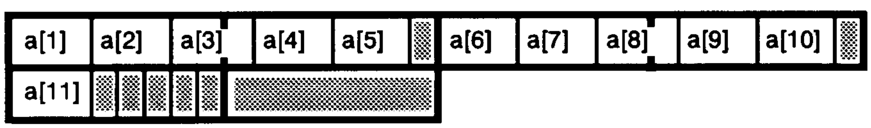
Table A-7. Allocation and Alignment of Enumeration Fields of Packed Records (Pascal/V Packing Algorithm)

Required Number of Bits	Number of Bits Allocated	Field Alignment
1	1	Bit
2	2	Bit
3	3	Bit
4	4	Bit
5	5	Bit
6	6	Bit
7	7	Bit
8	8	Bit
9	9	Bit
10	10	Bit
11	11	Bit
12	12	Bit
13	13	Bit
14	14	Bit
15	15	Bit
16 (2 bytes)	2 bytes	2-byte

Example

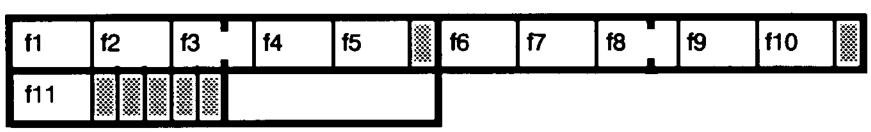
```
TYPE
   day = (sun,mon,tues,wed,thurs,fri,sat);
   enum_32 = (e1,e2,e3,e4,e5,e6,e7,e8,
              e9,e10,e11,e12,e13,e14,e15,e16,
              e17,e18,e19,e20,e21,e22,e23,e24,
              e25,e26,e27,e28,e29,e30,e31,e32);
VAR
   a : PACKED ARRAY [1..11] OF day;
   r : PACKED RECORD
          f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11 : day;
       END;
   aa : PACKED ARRAY [1..4] OF enum_32;
   rr : PACKED RECORD
           f1,f2,f3,f4 : enum_32;
        END;
```

Each element of the array a requires three bits, and no element can cross a 2-byte boundary. The entire array occupies 35 bits, and is allocated six bytes.



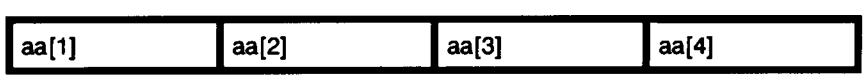
LG200173_038

Each element of the record r requires three bits, and no element can cross a 2-byte boundary. The entire record occupies 35 bits, and is allocated six bytes.



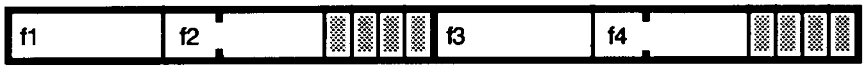
LG200173_039

Each element of the array aa requires six bits, but is allocated eight bits (one byte) and is byte-aligned. The entire array takes four bytes:



LG200173_040

Each field of the record `rr` requires and is allocated six bits, and no field can cross a 2-byte boundary. The entire record occupies 26 bits, and is allocated four bytes:



LG200173_041

Packed Subranges of Enumerations

This subsection explains how the Pascal/V packing algorithm allocates and aligns packed variables whose types are subranges of enumerations. These packed variables are either the elements of packed arrays or the fields of packed records. The algorithm treats the two cases differently.

The number of bits that an enumeration of a subrange type requires is determined by `ord(`*upper_bound_of_enumerated_subrange*`)`.

Table A-8 shows the relationship between the number of bits that an enumeration-of-subrange element of a packed array requires, the number of bits that the Pascal/V packing algorithm allocates it, and its alignment. No element crosses a 2-byte boundary.

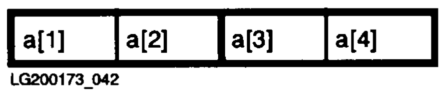
Table A-8.
Allocation and Alignment of Enumeration-of-Subrange Elements of Packed Arrays
(Pascal/V Packing Algorithm)

Required Number of Bits Per Element	Number of Bits Allocated Per Element	Alignment
1	1	Bit
2	2	Bit
3	3	Bit
4	4	Bit
5	5	Bit
6 to 8	8 (1 byte)	Byte
9 to 16	16 (2 bytes)	2-byte

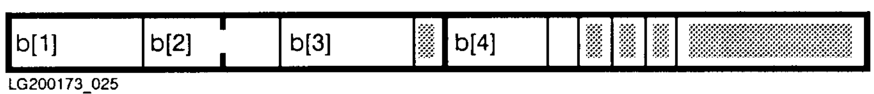
Example

```
TYPE
   enum_32 = (e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,
              e11,e12,e13,e14,e15,e16,e17,e18,e19,e20,
              e21,e22,e23,e24,e25,e26,e27,e28,e29,e30,
              e31,e32);

VAR
   a : PACKED ARRAY [1..4] OF e7..e15;
   b : PACKED ARRAY [1..4] OF e24..e31;
```

Each element of array `a` requires and is allocated four bits (see Table A-6). The elements are bit-aligned, and the entire array occupies 16 bits. It is allocated two bytes:



Each element of array `b` requires and is allocated five bits (see Table A-6). The elements are bit-aligned, and the entire array occupies 21 bits. It is allocated four bytes.

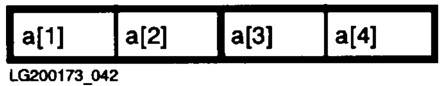


To the enumeration-of-subrange field of a packed record, the Pascal/V packing algorithm allocates the required number of bits. Any allocation from one bit to two bytes is possible. The field is bit-aligned, but never crosses a 2-byte boundary.

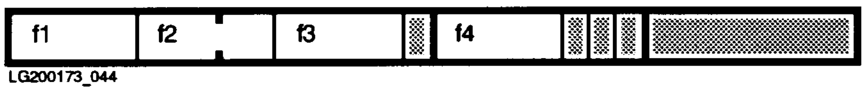
Example

```
TYPE
   enum_32 = (e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,
              e11,e12,e13,e14,e15,e16,e17,e18,e19,e20,
              e21,e22,e23,e24,e25,e26,e27,e28,e29,e30,
              e31,e32);

VAR
   a : PACKED RECORD
          f1,f2,f3,f4 : e7..e15;
       END;


   b : PACKED RECORD
          f1,f2,f3,f4 : e24..e31;
       END;
```

Each field of record a requires and is allocated four bits. The fields are bit-aligned, but cannot cross 2-byte boundaries. The entire record is allocated two bytes:



Each field of record b requires and is allocated five bits. The fields are bit-aligned, but cannot cross 2-byte boundaries. The entire record occupies 21 bits. It is allocated four bytes:



Packed Subranges of Integers

This subsection explains how the Pascal/V packing algorithm allocates and aligns packed variables whose types are subranges of integers. These packed variables are either the elements of packed arrays or the fields of packed records.

To the integer subrange variable of a packed array or packed record, the Pascal/V packing algorithm allocates the required number of bits (if the subrange is, or is included in, -32768..32767) or four bytes (if the subrange is outside that range).

Table A-9 shows the relationship between the number of bits that an element of a PACKED array of subrange type requires, the number of bits that the Pascal/V mapping algorithm allocates it, and its alignment.

Table A-9.
Allocation and Alignment of Elements of Packed Arrays of Subrange Type (Pascal/V Packing Algorithm)

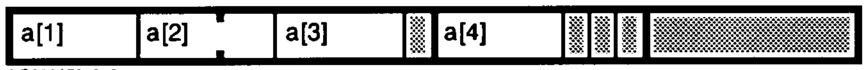
Required Number of Bits Per Element *	Number of Bits Allocated Per Element	Alignment
1	1	Bit
2	2	Bit
3	3	Bit
4	4	Bit
5	5	Bit
6 to 8	8 (1 byte)	Byte
9 to 16	16 (2 bytes)	2-byte
32	32 (4 bytes)	2-byte

* Only if the subrange is, or is included in, -32768..32767; four bytes otherwise.

Example

```
VAR
   a : PACKED ARRAY [1..4] OF 0..16;
   b : PACKED ARRAY [1..4] OF 0..32;
```

Each element of the array `a` requires and is allocated five bits, and is bit-aligned (see Table A-8). The entire array occupies 20 bits. It is allocated four bytes:



LG200173_043

Each element of the array `b` requires six bits, is allocated one byte, and is byte-aligned (see Table A-8). The entire array occupies four bytes.

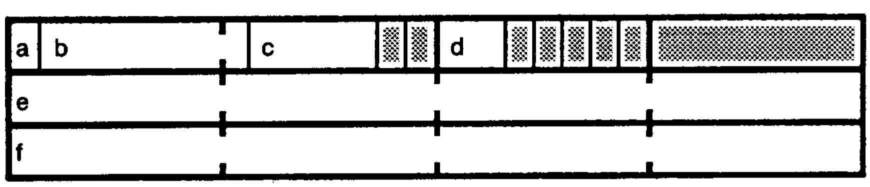
LG200173_026

For the integer subrange type of a packed record, any bit allocation from one bit to 15 bits is possible, as are allocations of two and four bytes. Bit allocations are bit-aligned, but never cross 2-byte boundaries. Two- and 4-byte allocations are 2-byte aligned. See "Records" for more information.

Example

```
VAR
   r : PACKED RECORD
          a : 0..1;        {Requires 1 bit}
          b : 0..255;      {Requires 8 bits}
          c : 0..16;       {Requires 5 bits}
          d : 0..4;        {Requires 3 bits}
          e : 10..40000;   {Requires 4 bytes}
          f : 0..MAXINT;   {Requires 4 bytes}
       END;
```

The fields of the record *r* are allocated the bits that they require. Fields *a, b, c,* and *d* are bit-aligned, but cannot cross 2-byte boundaries (notice where *d* and *e* start). Fields *e* and *f* are 2-byte-aligned.



LG200173_027

Compiling, Linking, and Running Your Program

To make your HP Pascal program a valid MPE/iX process, you must compile, link, and run it.

The HP Pascal compiler compiles your source program, which is in a textfile. It translates your source code to binary form and stores it in an object file or in an RL.

The MPE/iX linker prepares the object file for execution by binding the procedures in the object modules together and defining the initial requirements of the user data stack.

The MPE/iX operating system allocates space for the program, binds its external routines to it, and runs it. (The external routines are in executable libraries).

Additionally, the compiler looks for a system-wide file called `PASCNTL.PUB.SYS`. If the file exists and is not empty, the compiler opens and reads the file. The file should contain only compiler options and comments. If there is anything else in the file, the compiler emits an error message. If the file is empty, which is the default, the compiler does not attempt to open it. For more information on the system-wide file, refer to the section on compiler options in the *HP Pascal/iX Reference Manual*.

Figure A-1 shows how a source program (in a *textfile*) becomes a running program on MPE/iX.

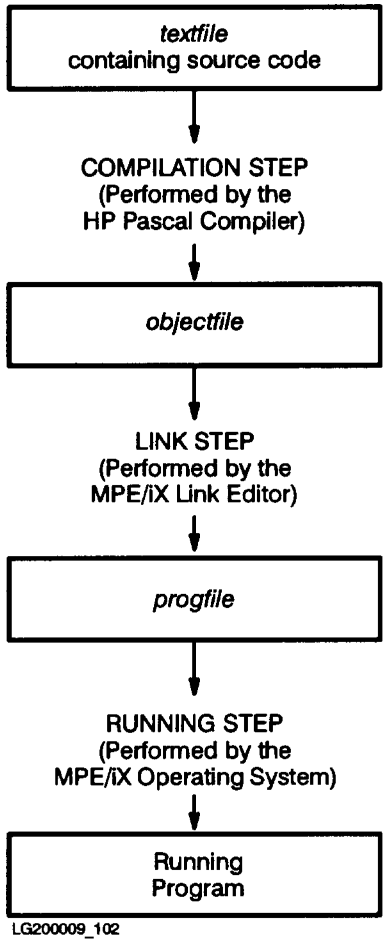


Figure A-1. How Source Code Becomes a Running Program on MPE/iX

This section explains:

- The MPE/iX command files that perform the steps shown in Figure A-1.
- How to run the HP Pascal compiler with the MPE/iX command :RUN PASCALXL.PUB.SYS.
- How to pass run-time parameters to your program.

Command Files

Table A-10 shows the MPE/iX command files that you can use to perform the steps shown in Figure A-1. Each command or command file in the right-hand column of Table A-1 performs the step or steps in the left-hand column (for example, the command :PASXL performs the compilation step, the command :PASXLLK performs the compilation and linking steps, and the command :PASXLGO performs the compilation, linking, and running steps).

Table A-10. MPE/iX Command Files That Compile, Link, and Run a Program

Steps	MPE/iX Commands or Command Files
To Compile	`:PASXL`
To Link	`:LINK`
To Run	`:RUN`
To Compile and Link	`:PASXLLK`
To Compile, Link, and Run	`:PASXLGO`

If you plan on linking as a separate step and would like more information on linking, refer to the *HP Link Editor/XL Reference Manual*.

Table A-11 gives the MPE/iX command files that are equivalent to the MPE V commands PASCAL, PASCALPREP, and PASCALGO. (Each command file name has *group* "`pub`" and *account* "`sys`"—see "File Names".)

Table A-11. Equivalent MPE V Commands and MPE/iX Command Files

MPE V Command	MPE/iX Command File
`:PASCAL`	`:PASXL`
`:PASCALPREP`	`:PASXLLK`
`:PASCALGO`	`:PASXLGO`

Syntax

PASXL [*textfile*][,[*objectfile*][,[*listfile*][,*libfile*]]][; INFO="*options*"]

PASXLLK [*textfile*][,[*progfile*][,[*listfile*][,*libfile*]]][; INFO="*options*"]

PASXLGO [*textfile*][,[*listfile*][,*libfile*]][; INFO="*options*"]

Parameters

textfile The name of the *textfile* that contains the source code to be compiled.

If you are running HP Pascal from your terminal, *textfile* is usually a file, but the default is `$STDIN`. `$STDIN` is the current input device, usually your terminal.

When *textfile* is the terminal, you can enter source code interactively in response to the prompt ">." When you have entered every line of your source code, type a colon (:) in response to the prompt.

The source code to be compiled can be either a program or a list of modules. A list of modules has the syntax:

module1[; *module2* [; ... [; *modulen*]] ...]

where *module1* through *modulen* are module bodies.

objectfile The name of the object file or RL on which the compiler writes the binary form of the source code that is in *textfile*. The default is `$OLDPASS` or `$NEWPASS`.

listfile The name of the file on which the compiler writes the program listing. It can be any ASCII file. The default is `$STDLIST`. `$STDLIST` is usually the terminal if you are running HP Pascal from a terminal; it is usually the job spool file if you are running a batch job.

If your terminal is both *textfile* and *listfile*, the compiler does not write the program listing on the terminal. If this is a permanent disk file, excess space is released with the CRUNCHED close option. See "Additional Features" earlier in this appendix.

If *listfile* is $NULL or a file other than `$STDLIST`, the compiler displays lines that contain errors on `$STDLIST`.

options A string of 132 or fewer characters, whose value is a list of compiler options. The compiler encloses the list in dollar signs and inserts it before the first line of code in *textfile*. The default is the empty string.

progfile The name of the program file on which the MPE/iX linker writes the linked program. The default is `$NEWPASS`.

libfile The name of the Pascal library file that the compiler searches if a search path is not specified with the compiler option SEARCH. The default is PASLIB in your group and account.

:RUN PASCALXL.PUB.SYS

The HP Pascal/XL compiler is a program file named PASCALXL.PUB.SYS. You can use the MPE/iX command :RUN to execute PASCALXL.PUB.SYS (that is, to invoke the HP Pascal/iX compiler).

The compiler files and their defaults are:

File	Default
Source file	`$STDIN`
Object file	`$OLDPASS` or `$NEWPASS`
Listing file	`$STDLIST`
Library file	PASLIB

To override the defaults:

1. Use the MPE/iX command :FILE to equate the nondefault file with its formal file designator (the :FILE parameter *formaldesignator*). Use one :FILE command for each nondefault file.
2. Tell the MPE/iX command :RUN which files are not to be defaulted by passing the appropriate value to its PARM parameter.

The compiler files and their formal file designators are:

Compiler File	Formal File Designator
Source file	PASTEXT
Object file	PASOBJ
Listing file	PASLIST
Library file	PASLIB

Table A-12 lists the possible values for the PARM parameter and gives their meanings.

Table A-12. PARM Values and Their Meanings

PARM Value	Means "File equations exist for the following files:"		
	Object	Listing	Source
0 †			
1			*
2		*	
3		*	*
4	*		
5	*		*
6	*	*	
7	*	*	*

† PARM=0 is equivalent to the command :PASXL (without parameters).

Example

```
:RUN PASCALXL.PUB.SYS

:FILE PASTEXT=Program1
:FILE PASOBJ=Object1
:FILE PASLIST=List1
:FILE PASLIB=Library1
:RUN PASCALXL.PUB.SYS;PARM=7;INFO="TABLES ON"

:FILE PASTEXT=Program2
:FILE PASLIST=List2
:RUN PASCALXL.PUB.SYS;PARM=3

:FILE PASLIST=List3
:FILE PASOBJ=Object3
:RUN PASCALXL.PUB.SYS;PARM=6;INFO="TABLES ON,TITLE 'Program 3'"
```

It is an error if you specify in the :RUN command that the compiler not use the default for one of the compiler files, and you do not provide a file equation for that file.

Example

```
:FILE PASTEXT=Program2
:FILE PASLIST=List2
:RUN PASCALXL.PUB.SYS;PARM=7
```

The above command sequence causes the compilation to abort with an error because `PARM=7` specifies that the :RUN statement not default the source, listing, or object file and no file equation is provided for the object file.

Run-Time Parameters

You can pass the run-time parameters PARM and INFO to your program with the RUN command. For each parameter that you want your program to access, you must:

- Specify a program parameter in the program heading (the position of the variable is not important).
- Declare the program parameter as a global variable.

The program parameter that corresponds to PARM must be of type *shortint*.

The program parameter that corresponds to INFO must be of type *string* or *PAC*.

MPE/iX checks the ranges of the actual program parameters for PARM and INFO if the RANGE compiler option is ON when the compiler encounters the first line of the statement part of the main program. (For more information on the RANGE compiler option, see the *HP Pascal/iX Reference Manual*.)

Example

If the `progfile` named `ex1` contains code for the program:

```
PROGRAM example_1 (parm,info);
VAR
   parm : integer;
   info : PACKED ARRAY [1..255] OF char;
BEGIN
END.
```

then the command:

```
:RUN ex1; PARM=3; INFO="abc"
```

assigns the value `3` to `parm` and the value `abc` to `info` before executing the program `example_1`.

B

HP-UX Dependencies

This appendix explains how the HP Pascal compiler works on the HP-UX operating system. It explains:

- How HP-UX affects system dependent HP Pascal features.
- HP-UX extensions to HP Pascal.
- How to compile, prepare, and run your HP Pascal program on HP-UX.

System Dependent Features

System dependent features are available to all HP Pascal programs (regardless of the system on which the compiler is running), but the system affects their definitions and behavior. System dependent HP Pascal features fall into these categories:

- Compiler options.
- File names.
- Input/output.
- Miscellaneous.

Compiler Options

The following compiler options are available to programs compiled by the HP Pascal compiler running on either the HP-UX or MPE/iX operating system, but they work differently on the two systems.

```
INCLUDE
SYMDEBUG
```

See the *HP Pascal/HP-UX Reference Manual* for more information on these compiler options.

File Names

Syntax

`[/]`[*pathname*]`...` {*identifier*}

Parameter

pathname Refer to the *HP-UX Reference* for syntax of *pathname*.

identifier The name of the main source file must end with "`.p`". Included files need not end with "`.p`".

Example

```
x.p
Pascal/tsource/tabort.p
/mnt/shankar/junk/t.p
```

For more information on HP-UX file names, refer to the *HP-UX Reference* manual.

Note The HP-UX operating system is case-sensitive. HP Pascal is not case-sensitive, except within string literals (such as "HP Pascal") and when you open a file without explicitly associating it with a physical file (that is, when you do not specify the second parameter to *open* or *reset*). In the latter case, the file name (*identifier*) is upshifted. The HP-UX operating system may not recognize the file by this new name. To avoid this problem, use all-capital names in the operating system environment for files that HP Pascal programs will use (for example, name an external file `FILE1`, not `File1`).

Standard Modules

Three standard modules are available on HP-UX: *stdinput*, *stdoutput*, and *stderr*.

If a module imports the *stdinput* module, it can use the predefined file *input* in I/O statements such as *read* and *readln*.

If a module imports the *stdoutput* module, it can use the predefined file *output* in I/O statements such as *write* and *writeln*.

If a module imports the *stderr* module, it can use the predefined file *stderr* in I/O statements such as *write* and *writeln*.

Example

```
MODULE mymod;
IMPORT
   StdInput, StdOutput;

EXPORT
   FUNCTION myproc : integer;

IMPLEMENT
   FUNCTION myproc : integer;
   VAR
      i : integer;
   BEGIN
      prompt('enter number:');  {need not specify output file}
      readln(i);                {need not specify input file}
      myproc := i;
   END;
END.
```

Additional Features

The HP Pascal features in the left-hand column depend on the HP-UX operating system in the ways explained in the right hand column.

Feature	HP-UX Dependency
Close options	The optional third parameter of the predefined procedure *close* can be SAVE, LOCK, TEMP, NORMAL, CRUNCH, or PURGE, whose meanings are: SAVE, LOCK, TEMP, NORMAL — The file is saved as a permanent file after it is closed. CRUNCH — This option is ignored. PURGE — The file is purged after it is closed.
Halt	HP-UX calls the system routine *exit(2)* with an integer parameter.
Input	The standard program parameter and textfile input is *stdin*.
Internal table size	The environment variable PASXDATA is the number of pages to allocate to each internal table (there is one internal table for identifiers and another for structured constants). The default internal table size is 100 pages. To set the internal table size to *n* pages, use the command: `setenv PASXDATA` *n* or the command: `PASXDATA=`*n* `export PASXDATA`
Maxpos	The call *maxpos(f)* returns *maxint*, regardless of *f*.
Open options	The third parameter of the predefined file-opening procedures *append, associate, open, read, reset, rewrite,* and *write*. It is optional for all but *associate*, for which it must have one of the values listed in "Associate Procedure" in Chapter 3.
Ord	At the STANDARD_LEVEL 'EXT_MODCAL' *ord* allows short pointers as arguments.

Output	The standard program parameter and textfile *output* is *stdout*.
Stderr	The standard program parameter and textfile *stderr* is the HP-UX file *stderr*.
System intrinsic file	`../../sys/pub/sysintr`
System default module library	`/usr/lib/paslib`
Temporary files	If the environment variable TMPDIR is defined (as a path to a directory to hold temporary files), temporary files are placed in that directory; otherwise, temporary files are created in the directory */usr/tmp*. (See the standard HP-UX entry point *tempdir(2)*.)
Write	If the file being written is a terminal, the output is unbuffered. This means that *write* to a terminal has the same behavior as *prompt*.

HP-UX Extensions

HP-UX extensions are available only to programs that are compiled by the HP Pascal compiler running on the HP-UX operating system. The programs themselves must also run on the HP-UX operating system. The HP-UX extensions are:

- Access to special global variables through the EXTERNAL directive.
- The predefined function *get_alignment*, which returns the alignment requirement of a given type or variable.
- The predefined function *statement_number*, which returns the statement number of the statement that calls it.

Accessing Special Global Variables

The global variable *errno* is special in that a program can access it through the EXTERNAL directive.

Example

```
$EXTERNAL$
PROGRAM ErrorNo_Example;

VAR
   ErrorNumber $ALIAS 'errno'$ : INTEGER;

FUNCTION Pas_Errno : integer;
BEGIN
   Pas_Errno := ErrorNumber;
END;

BEGIN
END.
```

When another compilation unit is linked with the preceding program, it can access the function *Pas_Errno,* which returns the value of the global variable *errno*.

Fnum Function

The predefined function *fnum* returns the HP-UX file number of the physical file currently associated with a given logical file. You can use this file number in system calls.

Syntax

fnum (*filename*)

Parameter

filename The name of the logical file. This parameter is required, even if the logical file is the standard file *input* or *output*. The logical file must be associated with a physical file.

Example

```
program xref(output);
const SEEK_SET=0;       { Set file pointer to "offset" }
      SEEK_CUR=1;       { Set file pointer to current plus "offset" }
      SEEK_END=2;       { Set file pointer to EOF plus "offset" }

var s_file : text;
    max    : integer;
    f      : integer;

function lseek(fildes:integer; offset:integer; whence:integer): integer;
   external;

begin
reset(s_file,'foo');
f:=fnum(s_file);
max:=lseek(f,0,seek_end);
writeln('file#:',f:1,', max bytes=',max:1);
end.
```

Output:

```
file#:3, max bytes=487
```

Get_alignment Function

The predefined function *get_alignment* returns the alignment requirement of a given type or variable.

Syntax

$$\texttt{get\_alignment (}\left\{\begin{array}{l} \textit{variable} \\ \textit{type} \end{array}\right\}\texttt{)}$$

Parameters

variable Any variable. The function `get_alignment` returns its alignment requirement.

type Any type identifier (the name of any type). The function `get_alignment` returns its alignment requirement.

Example

```
PROGRAM prog;

TYPE
   Rec = $ALIGNMENT 8$
         RECORD
            f1 : integer;
            f2 : shortint;
            f3 : real;
         END;

   integer_ = $ALIGNMENT 2$ integer;

VAR
   ptr : ^integer_;

BEGIN
   i := get_alignment(rec);

   IF get_alignment(ptr^) <> 2 THEN
      .
      .
      .
END.
```

Statement_number Function

The predefined function *statement_number* returns the statement number of the statement that calls it, as shown on the compiled listing. It is a useful debugging aid, especially when used with the predefined procedure *assert*.

Syntax

```
statement_number
```

Example

```
PROGRAM prog (output);

VAR
   i : integer;

BEGIN
   i := statement_number;
   writeln('Current Statement Number is ', i);
   assert(a > b, statement_number);
END.
```

Compiling, Linking, and Running Your Program

To make your HP Pascal program a valid HP-UX process, you must compile, link (and load), and run it.

The HP-UX command *pc* coordinates the HP Pascal compiler (`/usr/lib/pascomp`) and the HP-UX linker loader (`/bin/ld`).

The name of the file containing your source program must end with `.p` (for example, `prog.p`). The extension `.p` causes the `pc` command to call the HP Pascal compiler, which compiles your program and stores the resultant code in an object file. The name of the object file ends in `.o` (if the source file name is `prog.p`, the object file name is `prog.o`). If `prog.p` is the only file parameter of a particular `pc` command, and it compiles and links successfully, then the object file is not saved.

If the compiler does not find errors in the program, the *pc* command calls the linker, *ld*, which links the object file with required library files into a program file. The name of the program file is `a.out` (unless you specify another name in the *pc* command) and it resides in the directory from which *pc* was invoked. The program file is ready to run.

Figure B-1 shows how a file named *prog.p* becomes a running program on HP-UX.

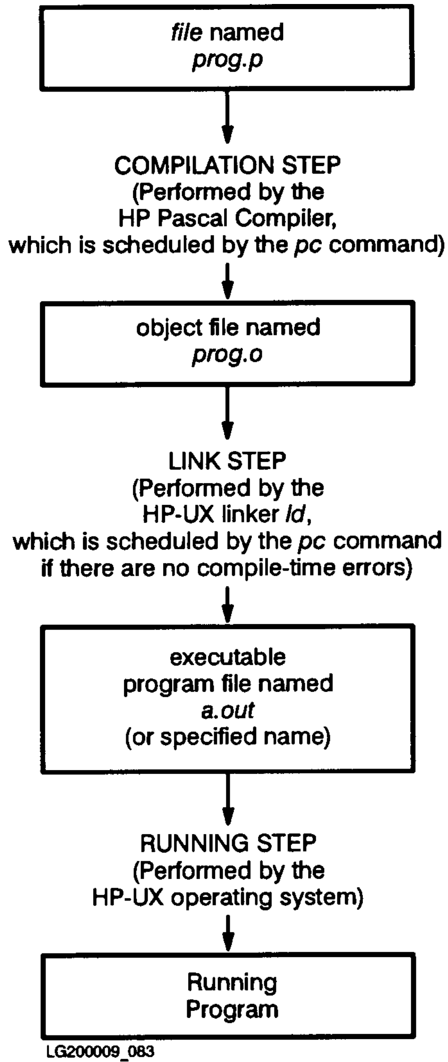


Figure B-1. How a File Becomes a Running Program on HP-UX

This section explains:

- The HP-UX *pc* command.
- How to pass run-time parameters to your program.
- How HP-UX handles interrupts.
- How HP-UX handles run-time errors.

pc Command

The HP-UX command *pc* coordinates the HP Pascal compiler (`/usr/lib/pascomp`) and the HP-UX linker loader (`/bin/ld`).

Additionally, the compiler looks for a system-wide file called `/usr/lib/pasopts`. If the file exists and is not empty, the compiler opens and reads the file. The file should contain only directives and comments. If there is anything else in the file, the compiler emits an error message. If the file is empty, which is the default, the compiler does not attempt to open it. For more information on the system-wide file, refer to the section on compiler options in the *HP Pascal/HP-UX Reference Manual*.

Syntax

pc [*file* | *option*] [...]

Parameters

file At least one file is required.

option Any of the following instructions to the compiler:

`-A` Produce warnings when non-ANSI Pascal features are found (same as ANSI ON).

`+a` Cause the compiler to generate archived object (`.a`) files instead of simple object (`.o`) files. This option exists for compatibility with the Series 300 *pc* command.

`-C` Suppress code generation. No object (`.o`) files will be created and linking will be suppressed. This is effectively a request for syntax/semantics checking only (same as CODE OFF).

`+C` Convert MPE/iX format file names in the compiler options BUILDINT, INCLUDE, LISTINTR, SPLINTR, and SYSINTR to HP-UX-format file names. Fully qualified HP-UX-format file names (those that begin with slash, like `/mnt/srf/file`) are not converted. This option is the same as the compiler option CONVERT_MPE_NAMES

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 lower case 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

```
$INCLUDE 'F.Group.Account'$
```

then the compiler appends the appropriate path information to `f` and searches for the resulting name (for example, `root/account/group/f`, where `root` is the parent of the account-level directories).

`-c` — Suppress linking and only produce object (`.o`) files from source files.

`-D`*name*=*bool*,
`-D`*name* — Defines *name* is as if it has been set (with `$SET`) to the *n*th line on the source file. *bool* can be either TRUE or FALSE; if *bool* in not specified, *name* is set to TRUE. *name* and *bool* can be uppercase or lowercase. The order in which the compiler encounters `$SET`s (regardless of relative order on the command line) is:

1. `-D`*name*`=`*bool*
2. `+Q` *dfile*
3. source file

The compiler overrides `-D`*name*`=`*bool* with any subsequent duplicate use of `$SET`, always taking the last one and issuing a warning.

`+DA`*model* — Generates object code for a specific version of the PA-RISC architecture. *model* can be a model number such as 750 or 870, or one of the following architecture specifications:

`1.0` — Generates object code for PA-RISC 1.0 architecture or higher. This is the default for all Series 800 models.

`1.1` — Generates object code for PA-RISC 1.1 architecture. This is the default for all Series 700 models.

Note that object code generated for PA-RISC 1.1 will not execute on PA-RISC 1.0 implementations.

See the file `/usr/lib/sched.models` for model numbers and their architectures. You can use the command `uname -m` to determine the model number of your system.

`+DA`*model* also specifies the appropriate library search path for HP-UX math libraries. If your program calls any of the standard Pascal Arithmetic functions, using `+DA1.0` links the PA-RISC 1.0 version of the math library and using `+DA1.1` links the PA-RISC 1.1 version of the library. The PA-RISC 1.1 libraries have performance enhancements and new routines that the PA-RISC 1.0 libraries lack. See the *HP-UX Floating-Point Guide* for more information about using math libraries.

`+DS`*model*

Perform instruction scheduling appropriate for a specific implementation of the PA-RISC architecture. *model* can be a model number such as 750 or 870, or one of the following architecture specifications:

`1.0` Perform scheduling tuned to one implementation of PA-RISC 1.0.

`1.1` Perform scheduling tuned to one implementation of PA-RISC 1.1.

See the file `/usr/lib/sched.models` for model numbers and their architectures. The compiler determines the default scheduling based on the model number returned by *uname(2)*.

This option affects only performance of the object code by scheduling the code based on the specific latencies of the target implementation. The resulting code executes correctly on other PA-RISC implementation, subject to the `+DA` option.

`+FP`*flag*

Specify how the run time environment for floating-point operations should be initialized at program start up. *flag* is a series of upper or lower case letters from the set `[VvZzOoUuIiDd]` with no embedded white-space. If the upper-case letter is selected, that behavior is enabled. If the lower-case letter is selected or if the letter is not present in *flag*, the behavior is disabled. The default is that all behaviors are disabled. The list below describes the behaviors:

`V` Trap on invalid floating-point operations.

`Z` Trap on divide by zero.

`O` Trap on floating-point overflow.

`U` Trap on floating-point underflow.

`I` Trap on floating-point operations that produce inexact results.

`D` Enable sudden underflow (flush to zero) of denormalized values.

Enabling sudden underflow is possible only on implementations of PA-RISC 1.1 or higher; it is not possible on PA-RISC 1.0.

To dynamically change these settings at run time, refer to "ARITRAP and HPENBLTRAP Intrinsics" in Chapter 11 or *fpgetround(3M)*.

`-G` — Prepare object files for profiling with the *gprof* utility (see "GPROF" in the *HP Pascal/HP-UX Reference Manual*).

`-g` — Generate additional information for the symbolic debugger, and ensure that the program is linked as required for the symbolic debugger.

`-I`*dir* — Add *dir* to the list of directories that search for `$INCLUDE` files whose names do not begin with `/`. The search is performed in the following order:

1. The directory containing the source file.
2. Directories specified with the `-I` option.
3. The current working directory.
4. The standard directory `/usr/include`.

`-L` — Write a program listing to `stdout`.

`-l`*x* — Cause the linker to search the `lib`*x*`.sl` or `lib`*x*`.a` libraries in an attempt to resolve currently unresolved external references. Because a library is searched when its name is encountered, placement of a `-l` is significant. If a file contains an unresolved external reference, the library containing the definition must be placed *after* the file on the command line. See *ld(1)* for more information.

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

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

`-O` — Turn on optimization. The compiler performs level 2 optimization. See `+O`*opt*.

`+O`*arg* — Perform optimizations selected by *arg*. There are two kinds of arguments to the `+O` optimization option. Those in the first group can have *arg* defined as:

`1` — Perform level 1 optimizations. These include branch optimizations, dead code elimination, instruction scheduling, and peephole optimization.

`2` — Perform level 2 optimizations. These include common subexpression elimination, constant folding, loop invariant code motion, coloring register allocation, and store-copy optimization. Level 2 optimizations are a superset of level 1 optimizations. The `-O` option is equivalent to the `+O2` option.

`3` — Perform level 3 optimizations. These include, but are not limited to, interprocedural global optimizations. Level 3 optimizations are a superset of level 2 optimizations.

Those in the second group can have *arg* defines as:

`s`	Suppress optimizations which tend to increase the generated code size. Currently, these optimizations include software pipelining and loop unrolling.
`bb`*num*	Specify the maximum number of basic blocks allowed in a procedure that is to be optimized at level 2. If a procedure contains more than *num* basic blocks, level 1 optimization is performed for that procedure. The default value for *num* is 500 (same as `$OPTIMIZE 'BASIC_BLOCKS`*num*`'$`).

The arguments in the second group implicitly request level 2 optimizations, but an argument from the first group overrides the implicit level 2 regardless of their relative positions on the command line.

`-o` *outfile*	Name the output file from the linker *outfile* instead of *a.out*.
`-P` *lines*	Allow *lines* lines per page of compiler listing, including header or trailer (same as the LINES compiler option).
`-p`	Prepare object files for profiling with the *prof* utility.
`-Q`	Cause the output file from the linker to be marked as not demand loadable (see `-q`).
`-q`	Cause the output file from the linker to be marked as demand loadable (see `-Q`).
`+Q` *dfile*	Cause *dfile* to be read before compilation of each source file. The file *dfile* can only contain compiler options.
`+R`	Turns off range checking (same as the compiler option RANGE OFF).
`-S`	Output an assembly file. This file is named *filename*`.s`, where *filename* is the base name of the source file.
`-s`	Cause the output of the linker to be stripped of symbol table information. See *strip(1)* in linker documentation. This option is incompatible with symbolic debugging.

-t *x,name* Substitute or insert subprocess *x* with *name* where *x* is one or more of an implementation-defined set of identifiers indicating the subprocesses. This option works in the following modes:

- If *x* is a single identifier, *name* represents the full path name of the new subprocess.
- If *x* is a set of identifiers, *name* represents a prefix to which the standard suffixes are concatenated to construct the full pathname of the new subprocesses.

The values *x* can assume are:

c Compiler body (standard suffix is *pascomp*).

0 Same as *c*.

l Linker (standard suffix is *ld*).

-v Enable verbose mode, producing a step-by-step description of the compilation process on *stderr*.

-w Turn off warning messages (same as the compiler option WARN OFF).

-W*c,arg1 [,arg2,... argn]* Cause *arg1* through *arg*n to be handed off to subprocess *c*. The *arg* parameters are of the form:

-*argoption*[,*argvalue*]

where *argoption* is the name of an option recognized by subprocess *c* and *argvalue* is a parameter for *argoption* (if it has one). The parameter *c* can have these values:

Value	Meaning
c	Compiler body (standard suffix is *pascomp*).
0	Same as *c*.
d	Driver program.
l	Linker (standard suffix is *ld*).

For example, the specification to pass the *-r* option (preserve rotation information) to the linker is *-Wl,-r*.

-Y Enable 16-bit Native Language Support when parsing string literals and comments (same as the compiler option NLS_SOURCE). Note that 8-bit parsing is always supported.

Other options—instructions to the linker—are also allowed. See *pc(1)* in the *HP-UX Reference* for details.

`-y` Generate additional information needed by static analysis tools and ensure that the program is linked as required for static analysis. This option is incompatible with optimization.

`+z`, `+Z` Both of these options cause the compiler to generate position independent code (PIC) for use in building shared libraries. However, you must use `+z` to generate PIC, unless certain limits are exceeded. Use `+Z` when limits are exceeded. If both `+z` and `+Z` are specified, only the last one encountered will apply. Note that `+z` is the same as `$SHLIB_CODE ON$` and `+Z` is the same as `$SHLIB_CODE 2$`.

The options `-G` and `-p` are ignored if you use either `+Z` or `+z`.

For more information about PIC , refer to *Programming on HP-UX*.

file The name of a *textfile* that contains source code for an HP Pascal program, or the name of an object file. The *textfile* name ends with *.p*; the object file name ends with *.o*.

For each *textfile*, the *pc* command calls the HP Pascal compiler, which tries to compile it. If the compiler compiles the *textfile* named `prog1.p` without errors, it produces an object file named `prog1.o` (which resides in the current directory).

If each *textfile* compiles successfully, the *pc* command calls the HP-UX Linker Loader, *ld*, which links all of the object files (*pc* command parameters and those resulting from compiles) into the final program file.

If `prog.p` is the only file parameter of a particular *pc* command, and it compiles and links successfully, then its object file, `prog.o`, is not saved.

Example

The command:

```
pc main.p ext1.p ext2.p
```

compiles the object files `main.o`, `ext1.o`, and `ext2.o`, into the final program file `a.out.` It is equivalent to the command sequence:

```
pc -c main.p
pc -c ext1.p
pc -c ext2.p
pc main.o ext1.o ext2.o
```

Note The HP Pascal compiler ignores the following Series 300 *pc* command options without warning:

```
+X
+x
+M
+b
+bfpa
+f
+ffpa
```

Run-Time Parameters

You can pass run-time parameters to your program as HP-UX command-line arguments when starting your program.

No arguments are automatically bound to program parameters. Even the three pre-opened (standard) files, *stdin*, *stdout,* and *stderr* are only bound to the HP Pascal textfiles *input, output,* and *stderr* if the program heading declares the textfiles.

Other run-time parameters must be obtained from the command line arguments by importing the predefined module *arg* and using the routines that it exports, which are:

Function	**Return Value**
argc	The total number of program arguments. (This integer is greater than or equal to one, because every HP-UX program has at least one program parameter, the program name.)
argn	An HP Pascal string that contains the *n*th program argument, where *n* is an argument to *argn* and must be in the range 0..*argc* - 1. If *n* is outside this range, the run-time library generates a range error. The call *argn(0)* returns the program name.
argv	A pointer to a null-terminated array of pointers, each of which points to a null-terminated PAC that contains an argument (see the export section of the *arg* module, on the next page).

The module *arg* belongs to the default module library `/usr/lib/paslib`; therefore, your program can import it without specifying a library with the SEARCH compiler option.

The export section for the module *arg* is:

```
MODULE arg;

EXPORT

TYPE
   arg_string1024 = string[1024];
   arg_type = PACKED ARRAY[1..32000] OF char;
   argarray = ARRAY[0..32000] OF ^argtype;
   argarrayptr = ^argarray;

FUNCTION argv : argarrayptr;
FUNCTION argc : integer;
FUNCTION argn (n : integer) : arg_string1024;

IMPLEMENT
   .
   .
   .
   .
END.
```

Example

```
$STANDARD_LEVEL 'HP_MODCAL'$
PROGRAM arg_demo (input, output);

VAR
   f     : text;
   line  : string[255];
   fname : string[80];

IMPORT arg;

BEGIN
   IF argc > 1 THEN BEGIN        {If a program argument was passed ...}
      fname := argn(1);          {assign it to fname ...}
      reset(f,fname);            {reset the file fname ...}

      WHILE NOT eof(f) DO BEGIN  {and list its contents.}
         readln(f,line);
         writeln(line);
      END;
   END; {IF}
END. {arg_demo}
```

Associating Program Header Files with Run-Time Parameters

On HP-UX, files defined in the program header are implicitly associated with run-time parameters. For example, if the program header is:

```
PROGRAM myprog (input, output, file1, file2);
```

then when the program `myprog` is run with command-line arguments, `file1` is bound to the first argument, and `file2` is bound to the second. The predefined files *input*, *output*, and *stderr* are not subject to this implicit association.

Other command-line arguments that are not subject to this implicit association are those that begin with plus (`+`) and minus (`-`). For example, if the compiled program produced from the above example is run with the command:

```
a.out -opt1 arg1 +opt2 arg2 arg3
```

then `file1` is bound to `arg1` and `file2` is bound to `arg2`. Therefore, if the program executes the statement:

```
reset (file1);
```

it is equivalent to the statement:

```
reset (file1, 'arg1');
```

If there is no run-time argument for a program header file, then the upshifted formal name of the file is implicitly associated with it. For example, if the program above is run with the command:

```
a.out arg1
```

then there is no run-time argument for `file2`, so it is associated with the file named FILE2. Of course, if you provide an explicit association, it overrides this implicit association. Also, if the file is already open before the statement executes, the usual rules apply (that is, the previous association is maintained).

Interrupt Handling

Your program can trap HP-UX interrupts (SIGINT and SIGQUIT, for example). The recommended way to trap these signals is to make explicit calls to the HP-UX system routine *signal*.

Note The HP9000 series 200 run-time routine *catch_signals* is supported, but a call to this routine will severely affect the error-handling mechanisms described in Chapter 11, because those depend on trapping certain HP-UX signals themselves (namely, SIGILL, SIGFPE, SIGBUS, SIGSEGV, and SIGSYS). **Use of this routine is strongly discouraged**.

Example

```
PROGRAM prog;

CONST
   BADSIG = -1;
   SIG_DFL = 0;
   SIG_IGN = 1;

   SIG_INT = 2;
   SIG_QUIT = 3;

VAR
   Old_Action : integer;

FUNCTION signal (SignalNum : integer;
                 ProcAddress : integer) : integer; EXTERNAL;
```

The function `signal` accepts a signal number, `SignalNum`, and a procedure address, `ProcAddress`. Whenever the signal with the number `SignalNum` is raised, the function transfers control to the procedure with the address `ProcAddress`. The function signal returns the old stored value of `ProcAddress`.

```
PROCEDURE InterruptHandler (SignalNum : integer); EXTERNAL;

BEGIN
   Old_Action := signal (SIGINT, Baddress (InterruptHandler));

   IF Old_Action = SIG_IGN THEN
      Old_Action := signal (SIGINT, SIG_IGN)
   ELSE IF Old_Action = BADSIG THEN
      {An invalid SignalNum or ProcAddress was passed};

   Old_Action := signal (SIGQUIT, Baddress (InterruptHandler));

   IF Old_Action = SIG_IGN THEN
      Old_Action := signal (SIGQUIT, SIG_IGN)
   ELSE IF Old_Action = BADSIG THEN
      {An invalid SignalNum or ProcAddress was passed};
END.
```

When either of the signals SIGINT or SIGQUIT is raised (by entering CONTROL `C` on the keyboard, for example), the procedure `InterruptHandler` is called.

Note In the preceding example, if `InterruptHandler` is to return to the main program, its first action must be to rearm the signal mechanism (in the manner described above) for the signal that was trapped. This is necessary because every time a signal is trapped, the HP-UX operating system resets its action information (the stored value of `ProcAddress`) to SIG_DFL (the default action). The program cannot resume normal execution and trap interrupts again unless it rearms the signal handler.

Run-Time Error Handling

If HP-UX detects a run-time error, it aborts the program unless the program defines error recovery code. Error recovery code can catch run-time errors that originate from:

- In-line compiled code (for example: range violation errors, nil pointer errors, math overflow errors).
- Run-time support routines (for example: *string, set, math*).
- Pascal file system (I/O errors).
- HP-UX file system support (system errors).
- Hardware (signals), except the *kill* signal.

When compiling a program, the compiler generates code that will call the predefined procedure *escape* if HP-UX detects a run-time error in the compiled program. The procedure *escape* transfers control to the program's error recovery code (if the program has no error recovery code, the program aborts). For a complete explanation of error recovery code, see Chapter 11.

Run-time errors in in-line compiled code are unique in that they can be suppressed—that is, you can tell the compiler not to generate code to catch them (see the compiler option RANGE in the *HP Pascal/HP-UX Reference Manual*). Run-time errors from other sources cannot be suppressed.

Most run-time errors that arise from interaction between in-line compiled code and run-time support routines are I/O errors. A few are system errors.

Compiling for Different Versions of the PA-RISC Architecture

Different HP 9000 systems use different versions of the PA-RISC architecture. Some models use PA-RISC 1.0 while other models use PA-RISC 1.1. The instruction set on PA-RISC 1.1 is a superset of the instruction set on PA-RISC 1.0. As a result, code generated for PA-RISC 1.0 systems will run on PA-RISC 1.1 systems, though possibly less efficiently than if it were specifically generated for PA-RISC 1.1. However, code generated for PA-RISC 1.1 systems will *not* run on PA-RISC 1.0.

By default, compiling on any series 800 system generates PA-RISC 1.0 code and compiling on any series 700 system generates PA-RISC 1.1 code. Use the `+DA` option to change this default behavior.

In addition, the instruction scheduling is different on some implementations of these architectures. You can improve performance on a particular model of the HP 9000 by requesting that the compiler use instruction scheduling tuned to that particular model. However, in contrast with the different instruction sets discussed above, using scheduling for one model does not prevent your program from executing on another model.

By default, the compiler uses scheduling tuned for the system where you are compiling. Use the `+DS` option to change this default behavior.

Using +DA to Generate Code for a Specific Version of PA-RISC

Use the `+DA` option to specify which PA-RISC instruction set the compiler should use when generating code. Specifying `+DA1.0` ensures your code will run on all HP 9000 models, although the performance of your program may not be as good as it could be on PA-RISC 1.1 systems. Specifying `+DA1.1` may give better performance on PA-RISC 1.1 systems, but the executable file generated with this option will not run on PA-RISC 1.0 systems.

Using +DS to Specify Instruction Scheduling

Use the `+DS` option to specify instruction scheduling tuned to a particular implementation of PA-RISC. For example, to specify instruction scheduling for the model 867, use `+DS867`.

Guidelines for Using +DA and +DS

When you use the `+DA` and `+DS` options depends on your particular circumstances. Here are some possibilities.

- If you plan to run your program on the same system where you are compiling, you do not need to use either the `+DA` or `+DS` option. The compiler generates code tuned for your system.
- If you plan to run your program on one particular model of the HP 9000 and that model is different from the one where you compile your program, use the following combination:
  - `+DA`*model* with the model number of the target system, and
  - `+DS`*model* with the model number of the target system.

  For example, if you are compiling on a 720 and your program will run on an 855, you should use `+DA855 +DS855`. This will give you the best performance on the 855.

- If you plan to run your program on many models of the HP 9000, you could use the following combination:
  - `+DA1.0` to ensure portability, and
  - `+DS`*model* with the model number of the fastest system you will be running your application on.

  For example, using `+DA1.0 +DS897` ensures your program can run on all series 700s and 800s, and uses scheduling for the model 897. You might want to use scheduling for a high-performance system (such as the 897), assuming your customers with high-performance systems want the fastest performance from your application.

See the file `/usr/lib/sched.models` for model numbers and their architectures. You can use the command `uname -m` to determine the model number of your system.

Compiling in Networked Environments

When compiles are performed using diskless workstations or NFS-mounted file systems, it is important to note that the default code generation and scheduling are based on the local host processor. The system model numbers of the hosts where the source or object files reside do not affect the default code generation and scheduling.

Glossary

actual parameter
An argument that is passed to a procedure, function, or subprogram. Contrast with *formal parameter*.

address
An exact location in memory. A program can store or retrieve data from this address.

algorithm
A procedure used to solve a task. It describes the sequence of steps or operations, done in a finite number of steps.

allocate
To set up a memory location to hold variable values.

alpha character
A character in the range of A through Z and a through z.

alphanumeric character
A character in the range of A through Z, a through z, and 0 through 9.

argument
A variable or constant whose value is passed to a procedure or function. See *actual parameter*, *formal parameter*, or *parameter*.

arithmetic expression
An expression that performs arithmetic operations and consists of constants, variables, and arithmetic operators.

array
A data structure in which consecutive memory locations contain data items of the same type.

ASCII
American Standard Code for Information Interchange; a seven-bit code representing a prescribed set of characters.

assembly language
A programming language in which each operation performed by the Central Processing Unit (CPU) is written as a symbolic instruction. Assembly language is a convenient means of representing machine language. A program known as an assembler translates instructions written in assembly language into machine language.

assignment statement

Assigns a value to a variable or function by using the special Pascal symbol "`:=`".

binary

The method used to represent numbers, alphabetic characters, and symbols in digital computers. It is a base two numbering system that uses only two digits, 0's and 1's, to express numeric quantities.

bit

A unit of information with a value of 1 or 0. Usually eight bits equal one byte. A bit is the smallest unit of information in a digital computer.

block

Blocks contain groups of statements for programs, procedures, and functions, and are enclosed with the reserved words *begin* and *end*.

boolean expression

An expression that evaluates to a value of true or false.

buffer

The part of a computer or device memory where data is held temporarily until it can be processed or transmitted elsewhere. A buffer usually refers to a memory area that is reserved for I/O operations.

byte

A combination of eight consecutive bits treated as a unit. A byte represents one letter or number within the computer.

C

A high-level computer programming language that can do low-level manipulations.

COBOL

COmmon Business Oriented Language. A high-level computer language primarily used for business applications.

collating sequence

The "alphabetical order" of all characters used by a computer. They include digits, punctuation marks, and special characters. The collating sequence uses the same order of precedence as the numeric codes for characters, either in ASCII or EBCDIC.

comment

Information in a computer program that is ignored by the compiler, but is included for documenting the program for human readers.

compile time

The time during which a source program is translated by a compiler to an object program. Compile time is usually used to indicate things that happen when a program is compiled.

compile-time error

An error that occurs or that is detected at compile time.

compiler
A program that translates source code into machine instructions. The compiler also diagnoses and reports syntax errors found in the application program.

compound statement
A group of statements enclosed with the reserved words *begin* and *end*, and which are treated as a single statement.

concatenation
The operation of joining two or more character strings together.

constant
A fixed value, as opposed to a variable which is a symbol for a changing value.

construct
A structured constant; a construct specifies the value of a declared constant.

data
One or more items of information.

debug
To find and correct mistakes in a computer program.

decimal
The base 10 numbering system in which the numbers 0 through 9 are used.

default
A value or condition that is assumed by the operating system or compiler if no other value or condition is specified.

delimiter
A symbol that marks the beginning and end of a syntactic unit in source code.

disk
A circular plate used to store computer data; the disk can be fixed, removable, hard, or flexible.

dynamic variable
A variable which is not declared and cannot be referred to by name. A dynamic variable is created during execution of a program.

error recovery
The process of writing code that prevents a program from aborting due to run-time errors. Error recovery code does not catch compile-time errors, warnings, or notes.

executable object
A program or procedure that is ready to be executed.

execute
The act of a computer carrying out a set of instructions given by a program.

expression
A construct composed of operators and operands that represent the computation of a result of a particular type.

external routine
A routine defined in another compilation unit.

file-equate
To redirect the association of one physical file to another physical file, or to specify additional file attributes using the MPE XL `FILE` command.

formal parameter
A parameter which is defined in a procedure, function, or subprogram header.

function
A block that is invoked with a function call and returns a value.

function call
A call that invokes the block of a function and returns a value to the calling point of the program

function heading
Consists of the reserved word FUNCTION, an identifier that specifies a function name, an optional formal parameter list, and a result type.

hexadecimal
The base 16 numbering system in which the numbers 0 through 15 are used. 10 through 15 are represented by the letters A through F.

identifier
Used to denote declared constants, types, variables, procedures, functions, modules, and programs, and consists of a letter preceding an optional character sequence of letters, digits, or the underscore character (_).

initialize
To give an initial value to a variable in a program.

intrinsic
An external routine that can be called by a program written in any language that your operating system supports.

literal
A value in a program that is represented by it's actual value rather than a variable or a constant.

loop
When a program performs a statement over and over a specified number of times or while certain conditions are met.

maxint
The maximum value that an integer variable can contain.

minint
The minimum value that an integer can contain.

NLS
An acronym for Native Language Support.

operand
The variables, constants, or literals that are used in an operation.

operator
Defines the action to be performed on one or more operands.

optimization
The process which the compiler uses to modify your program so that it uses machine resources more efficiently.

parameter
The argument used for sending and receiving information to and from functions and procedures.

parameter list
The location in a program where the parameters and their values are declared.

PIC
An acronym for Position Independent Code.

precedence
Rules that determine the required order of operations.

procedure
A block of statements that are invoked with a *procedure call*.

procedure call
The call in a program that invokes the procedure block.

real number
Numbers that are whole or fractional. A real number can also have an exponent.

recursion
A programming technique in which a procedure calls itself.

relational operator
An operator that compares two operands and returns a Boolean result.

reserved word
Predefined terms that have special meaning to the Pascal language, and which can only be used for their specified purpose.

run-time error
An error the computer system finds in a program during run time.

semantic error
An error which is caused by using the wrong wording in a program.

separate compilation
The process of separating the source for a large program into pieces that can be compiled independently of other pieces.

source code
The input program that is to be translated by the compiler.

Standard Pascal
All of the rules and definitions of Pascal as defined by the ANSI standard.

statement
Pascal's single unit of activity. Each statement is separated by a semicolon.

static variable
A variable which is declared in the declaration part of a program block.

subprogram
See *procedure*.

top-down design
The process of breaking a problem into pieces that can be easily solved.

variable
A memory location that holds data values, and which is referenced by a variable name. Information in this location can be changed.

warning
The compiler produces warnings to indicate a possible source of run-time errors.

word
Four consecutive bytes. Some numeric items are defined in terms of words, and many items must start at a word boundary in memory.

Index

D

E

F

G

L

M

N

O

P

R

S

W

X