Compaq Pascal

User Manual for Tru64 UNIX

Order Number: AA-PV37B-TE

June 1999

This manual contains information about selected programming tasks using the *Compaq Pascal* programming language. It supersedes DEC Pascal User Manual for DIGITAL UNIX Systems, order number AA-PV37A-TE.

Revision/Update Information: This is an updated manual.

Software Version: *Compaq Pascal* Version 5.7

Compaq Computer Corporation
Houston, Texas

June 1999

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital Equipment Corporation or an authorized sublicensor.

© Digital Equipment Corporation 1999. All Rights Reserved.

Compaq, the Compaq logo, and the DIGITAL logo are Registered in the U.S. Patent and Trademark Office.

Alpha, AlphaServer, AlphaStation, Bookreader, DEC, DEC Pascal, DIGITAL, OpenVMS, Tru64 UNIX, ULTRIX, VAX and VMS are trademarks of Digital Equipment Corporation.

The following are third-party trademarks:

IEEE is a registered trademark of the Institute of Electrical and Electronics Engineers Inc.

Oracle Rdb, Oracle CODASYL DBMS, Oracle CDD/Repository, Oracle CDD/Administrator, Oracle RALLY, Oracle TRACE, Oracle Expert, Oracle InstantSQL, Oracle Graphical Schema Editor, Oracle RMU, Oracle RMUwin, Oracle TRACE Collector, Oracle SQL/Services, Oracle DBA Workcenter, and Oracle Module Language are trademarks of Oracle Corporation.

OSF/1 is a registered trademark of The Open Group.

PostScript is a registered trademark of Adobe Systems, Inc.

UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

X/Open is a registered trademark of The Open Group.

All other trademarks and registered trademarks are the property of their respective holders.

Compaq conducts its business in a manner that conserves the environment and protects the safety and health of its employees, customers, and the community.

ZK6084

This manual is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Preface ix

1 Getting Started

1.1 The Compaq Pascal Programming Environment 1–1
1.2 Commands to Create and Run an Executable Program 1–3
1.3 Creating and Running a Program Using a Separate Function . . . 1–4
1.3.1 Creating the Executable Program 1–5
1.3.2 Running the Sample Program 1–6
1.4 The pc Command and Related Software Components 1–6
1.4.1 The Driver Program 1–6
1.4.2 Compaq Pascal Compiler 1–7
1.4.3 Other Compilers 1–7
1.4.4 Linker (ld) 1–8
1.5 Program Development Stages and Tools 1–8

2 Compiling and Linking Compaq Pascal Programs

2.1 The pc Command 2–1
2.1.1 pc Command Examples 2–2
2.2 pc Command Options 2–4
2.2.1 pc Command Option Categories 2–5
2.2.2 pc Command Options Descriptions 2–6
2.3 Linking 2–26
2.3.1 Run-Time Libraries 2–26
2.3.2 Specifying Shared Object Libraries 2–27
2.4 Creating Shared Libraries 2–27
2.4.1 Creating a Shared Library Using the pc Command 2–28
2.4.2 Creating a Shared Library Using pc and ld Command 2–28
2.4.3 Choosing How to Create a Shared Library 2–29
2.5 Temporary Files 2–29
2.6 Using Multiple Input Files: Effect on Output Files 2–30
2.7 Interactions of File Name Suffix and Options 2–30

2.8 Using Listing Files 2–31

3 Separate Compilation

3.1 The ENVIRONMENT, HIDDEN, and INHERIT Attributes 3–2
3.1.1 Environment File Dependency Checking 3–5
3.2 Interfaces and Implementations 3–6
3.3 Data Models 3–10
3.4 Separate Compilation Examples 3–13

4 Optimizing Performance

4.1 Compiler Optimizations 4–1
4.1.1 Compile-Time Evaluation of Constants 4–3
4.1.2 Elimination of Common Subexpressions 4–4
4.1.3 Elimination of Unreachable Code 4–5
4.1.4 Code Hoisting from Structured Statements 4–5
4.1.5 Inline Code Expansion for Predeclared Functions 4–7
4.1.6 Inline Code Expansion for User-Declared Routines 4–7
4.1.7 Partial Evaluation of Logical Expressions 4–7
4.1.8 Value Propagation 4–8
4.1.9 Error Reduction Through Optimization 4–9
4.1.10 Strength Reduction 4–9
4.1.11 Split Lifetime Analysis 4–9
4.1.12 Code Scheduling 4–10
4.1.13 Loop Unrolling 4–10
4.1.14 Software Pipelining 4–11
4.2 Programming Considerations 4–12
4.3 Optimization Considerations 4–13
4.3.1 Compiling for Optimal Performance 4–13
4.3.2 Subexpression Evaluation 4–13
4.3.3 MAXINT and MAXINT64 Predeclared Constants 4–14
4.3.4 Pointer References 4–15
4.3.5 Variant Records 4–16
4.3.6 Effects of Optimization on Debugging 4–16
4.4 Analyze Program Performance 4–17
4.4.1 Use the time Command to Measure Performance 4–18
4.5 Profiling a Program 4–19
4.5.1 Program Counter Sampling (prof) 4–20
4.5.2 Call Graph Sampling (gprof) 4–21
4.5.3 Basic Block Counting (pixie and prof) 4–22
4.5.4 Source Line CPU Cycle Use (prof and pixie) 4–23
4.5.5 Creating and Using Feedback Files and Optionally cord 4–23

4.6 Controlling the Size of Global Pointer Data 4–25

5 Programming Tools

5.1 The C Language Preprocessor 5–1
5.1.1 Including Headers and Other Files 5–2
5.1.2 Conditional Compilation 5–3
5.1.3 String Substitution 5–3
5.1.4 C-Style Comments 5–4
5.2 Coding for the Debugger 5–4
5.2.1 Debugging Optimized Programs 5–4
5.2.2 Debugging Preprocessed Programs 5–5
5.3 Running the dbx Debugger 5–5
5.3.1 Debugger Data Types 5–5
5.3.2 Compaq Pascal Data Names 5–6
5.3.3 Activation Levels 5–7
5.4 Debugging Tips 5–7

6 Calling Conventions

6.1 Tru64 UNIX Calling Standard 6–1
6.1.1 Parameter Lists 6–1
6.1.2 Function Return Values 6–2
6.1.3 Contents of Call Stack 6–3
6.1.4 Unbound Routines 6–3
6.2 Parameter-Passing Semantics 6–4
6.3 Parameter-Passing Mechanisms 6–4
6.3.1 By Immediate Value Passing Mechanism 6–5
6.3.2 By Reference Passing Mechanism 6–6
6.3.3 By Descriptor Passing Mechanism 6–6
6.4 Passing Parameters Between Compaq Pascal and Other Languages 6–7
6.4.1 Parameter Mechanisms Versus Parameter Semantics 6–7
6.4.2 Passing Non-Routine Parameters Between Compaq Pascal and Other Languages 6–8
6.4.3 Passing Routine Parameters Between Compaq Pascal and Other Languages 6–10
6.5 Calling C Routines from Compaq Pascal 6–11

7 Error Processing and Condition Handling

7.1 Condition Handling Terms 7–1
7.2 Overview of Condition Handling 7–2
7.2.1 Condition Signals 7–3
7.2.2 Handler Responses 7–3
7.3 Writing Condition Handlers 7–4
7.3.1 Establishing and Removing Handlers 7–4
7.3.2 Declaring Parameters for Condition Handlers 7–5
7.3.3 Handler Function Return Values 7–6
7.4 Example of a Condition Handler 7–6

8 Migrating from OpenVMS to Tru64 UNIX Systems

8.1 Sharing Environment Files Across Platforms 8–1
8.2 Default Size for Enumerated Types and Booleans 8–1
8.3 Default Data Layout for Unpacked Arrays and Records 8–1
8.4 IADDRESS and VOLATILE 8–2
8.5 Overflow Checking 8–2
8.6 Bound Procedure Values 8–3
8.7 Argument List Functions 8–3
8.8 %DICTIONARY Directive 8–3
8.9 VAX Floating Datatypes 8–4
8.10 Relative and Indexed Files 8–4
8.11 Data Layout and Conversion 8–5
8.11.1 Natural Alignment, VAX Alignment, and Enumeration Sizes 8–5
8.11.2 Compaq Pascal Features Affecting Data Alignment and Size 8–6
8.11.3 Optimal Record Layout 8–7
8.11.4 Optimal Data Size 8–9
8.11.5 Converting Existing Records 8–9
8.11.6 Applications with No External Data Dependencies 8–9
8.11.7 Applications with External Data Dependencies 8–11

9 Migrating from Pascal for RISC to Compaq Pascal

9.1 Pascal for RISC and Compaq Pascal Compile-Time Differences 9–1
9.2 Pascal for RISC and Compaq Pascal Run-Time Differences 9–3

A Errors Returned by STATUS and STATUSV Functions

B Entry Points to Compaq Pascal Run-Time Library

C Diagnostic Messages

C.1 Compiler Diagnostics C–1
C.2 Diagnostic Messages C–2
C.3 Run-Time Diagnostics C–65

Index

Examples

1–1 Sample Main Program 1–3
1–2 Sample Main Program That Uses a Separate Function 1–4
1–3 Sample Separate Function Declaration 1–5
3–1 An Interface Module for Graphics Objects and Routines 3–14
3–2 An Implementation Module for Graphics Objects and Routines 3–14
3–3 A Graphics Main Program 3–16

Figures

2–1 pc Command Processing by File Suffix 2–31
3–1 Cascading Inheritance of Environment Files 3–4
3–2 Inheritance Path of an Interface, an Implementation, and a Program 3–6
3–3 Cascading Using the Interface and Implementation Design 3–8

Tables

1	Conventions Used in This Manual	xi
1–1	Main Tools for Program Development and Testing	1–8
2–1	Categories of pc Command Options	2–5
2–2	-align Option Keywords	2–7
2–3	-arch Option Keywords	2–7
2–4	-C Option Keywords	2–10
2–5	-enumeration_size Option Keywords	2–13
2–6	-granularity Option Keywords	2–15
2–7	-inline Option Keywords	2–16
2–8	Library Search Path	2–17
2–9	-platforms Option Keywords	2–21
2–10	-show Option Keywords	2–22
2–11	-std Option Keywords	2–23
2–12	-tune Option Keywords	2–24
2–13	-usage Option Keywords	2–24
5–1	The dbx Equivalents of Compaq Pascal Data Types	5–6
6–1	Parameter-Passing Mechanisms Systems	6–4
6–2	Parameter-Passing Methods	6–5
8–1	Unpacked Sizes of Fields and Components	8–6
9–1	Pascal for RISC and Compaq Pascal Compile-Time Differences	9–1
9–2	Pascal for RISC and Compaq Pascal Run-Time Differences	9–3
A–1	STATUS and STATUSV Return Values	A–1

Preface

This manual describes selected programming tasks using the *Compaq Pascal* programming language. It contains information on using some *Compaq Pascal* language elements in combination, and it provides examples of how to improve programming efficiency.

You can use the information in this manual to write programs or modules for the *Tru64 UNIX* operating system, formerly known as DEC OSF/1. If you need to write portable *Compaq Pascal* programs or language modules, see the *Compaq Pascal Language Reference Manual* for a checklist of language extensions not included in the Pascal standard. The *Compaq Pascal Language Reference Manual* also provides information on the Pascal standard.

Intended Audience

This manual is intended for experienced applications programmers with a basic understanding of the Pascal language. Some familiarity with the operating system is helpful. This is not a tutorial manual for new Pascal users.

Document Structure

This manual consists of the following chapters and appendixes:

- Chapter 1 provides an overview of the major software components associated with *Compaq Pascal*.
- Chapter 2 provides information on compiling programs, linking programs, running programs, and using text and object-module libraries.
- Chapter 3 describes how to create compilation units that you can compile separately.
- Chapter 4 describes programming techniques that improve the efficiency of compilation and execution.
- Chapter 5 provides information on tools for program development.

- Chapter 6 describes how *Compaq Pascal* passes parameters and calls routines.
- Chapter 7 provides information on error processing and writing condition handlers.
- Chapter 8 provides information on migrating from *OpenVMS* to *Tru64 UNIX* systems.
- Chapter 9 provides information on migrating from Pascal for RISC to *Compaq Pascal* on *Tru64 UNIX* systems.
- Appendix A provides a list of possible error values returned by the STATUS and STATUSV functions.
- Appendix B describes the entry points to utilities in the *Tru64 UNIX* run-time library.
- Appendix C provides descriptions of diagnostic messages that can be generated by a *Compaq Pascal* program at compile time and run time.

Related Documents

The following manuals may also be useful when programming in *Compaq Pascal*:

- *Compaq Pascal Language Reference Manual*—Provides information on the syntax and semantics of the *Compaq Pascal* programming language. In addition, this manual provides information about the alignment, allocation, and internal representation of data types supported by *Compaq Pascal* and descriptions of the error messages.
- *Compaq Pascal Installation Guide for Tru64 UNIX Systems*—Provides information on how to install *Compaq Pascal* on your operating system.
- Manuals for your operating system provide full information about your operating system.

Conventions

Table 1 presents the conventions used in this manual.

Table 1 Conventions Used in This Manual

Convention	Meaning
%	The default user prompt is your system name followed by a right angle bracket. This manual uses a percent sign (%) to represent this prompt.
% **`pwd`** /usr/usrc/jones	This manual displays system prompts and responses using a monospaced font. Typed user input is displayed in a bold monospaced font.
`monospaced`	This typeface indicates the name of a command, flag, pathname, file name, directory path, or partition. This typeface is also used in examples of program code, interactive examples, and other screen displays.
`cat`(1)	A shell command name followed by the number 1 in parentheses refers to a command reference page. Similarly, a routine name followed by the number 2 or 3 in parentheses refers to a system call or library routine reference page. (The number in parentheses indicates the section containing the reference page.) To read online reference pages, use the `man` command. Your operating system documentation also includes reference page descriptions.
. . .	A vertical ellipsis in a figure or example means that not all of the statements are shown.
. . .	A horizontal ellipsis means that the item preceding the ellipsis can be repeated. For example: s[,s] . . .
UPPERCASE lowercase	The operating system shell differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.

(continued on next page)

Table 1 (Cont.) Conventions Used in This Manual

Convention	Meaning

Temp : INTEGER;
PRED(n)
	Lowercase letters represent user-defined identifiers or elements that you must replace according to the description in the text.

newterm	Boldface text represents the introduction of a new term.
variable	Italic type indicates important information, a complete title of a manual, or variable information, such as user-supplied information in command flag syntax.
{ }	Large braces enclose list from which you must choose one item. For example: $\left\{\begin{array}{l}\text{STATUS}\\\text{DISPOSE}\\\text{DISP}\end{array}\right\}$
[ ]	Square brackets enclose items that are optional. For example: BLOCK DATA [nam]

In this manual, complex examples have been divided into several lines to make them easy to read. *Compaq Pascal* does not require that you format your programs in any particular way.

Reader’s Comments

Compaq welcomes your comments. If you would like to comment on a *Compaq Pascal* manual, please send the manual title, order number, and your comments by one of the following methods:

- FAX: 603–884–0120
  Attn: Languages Documentation, ZK02–3/K35
- A letter sent to the following address:

  Compaq Computer Corporation
  Languages Documentation, ZK02–3/K35
  110 Spit Brook Road
  Nashua, NH 03062–2698
  USA

1 Getting Started

This chapter includes the following:

- A summary of aspects of the Pascal language related to the *Compaq Pascal* program development environment on *Tru64 UNIX* systems (Section 1.1)
- The commands used to create, compile, link, and run a small program (Section 1.2)
- The commands used to create, compile, link, and run a sample main program that has separate files for a function declaration (Section 1.3)
- An overview of the major software components associated with *Compaq Pascal* (Section 1.4)
- A summary of the important program development stages and tools (Section 1.5)

1.1 The Compaq Pascal Programming Environment

The following aspects are relevant to the compilation environment and should be considered before extensive coding begins:

- To install *Compaq Pascal* on your system, obtain the media *Tru64 UNIX* layered products Compact Disc (CD), and perform the installation as described in *Compaq Pascal Installation Guide for Tru64 UNIX Systems*.
- Once *Compaq Pascal* is installed, you can:
  - Use the `pc` command to compile and link programs.
  - Use the online `pc`(1) reference page and this manual to provide information about the `pc` command.
- Make sure you have adequate stack size, especially if your programs use large arrays as data. Users may be able to overcome this problem by increasing the per-process data limit, using the `limit` command (C shell) or `ulimit` command (Korn and Bourne shell) (see `csh`(1), `ksh`(1), or `sh`(1)).

Determine whether the maximum per-process data size is already allocated by checking the value of the **maxdsiz** parameter in the system configuration file. If necessary, increase its value. Changes to the configuration file do not take effect until the operating system kernel has been rebuilt and the system has been rebooted.

For example, the following C shell commands check the current limits and then increase the size to a larger value for cases where this limit can be increased from your process:

```
% limit stacksize
stacksize       4096 kbytes
% limit stacksize 32676
```

- Make sure you have an adequate process file descriptor limit, especially if your programs use a large number of included files.

  During compilation, your application may attempt to use more included files than your descriptor limit allows.

  Users can view and usually increase the per-process limit on the number of open files by using the `limit` command (C shell) or `ulimit` command (Korn and Bourne shell) (see `csh`(1), `ksh`(1), or `sh`(1)).

  Determine whether the maximum per-process limit is already allocated by checking the value of the appropriate descriptor parameter in the system configuration file. If necessary, increase its value. Changes to the configuration file do not take effect until the operating system kernel has been rebuilt and the system has been rebooted.

  For example, the following C shell commands check the current limits and then increase the size to a larger value for cases where this limit can be increased from your process.

  ```
  % limit descriptor
  descriptors 100 files
  % limit descriptor 4096
  ```

- *Compaq Pascal* supports the use of the environment variable TMPDIR to specify a working directory (instead of the current directiry) to contain temporary files created during compilation. Several other environment variables can similarly be used during program execution.

  If you need to set environment variables frequently, consider setting these in your `.login` file or appropriate shell initialization file (`.cshrc` or `.profile`).

- The `pc` command recognizes certain file suffixes as files containing Pascal source code.

Compaq Pascal provides extensions to the ANSI PASCAL standard. When creating new programs that need to be standard-conforming for portability reasons, you should avoid or minimize the use of extensions to the PASCAL standard. Extensions to the PASCAL standard are identified visually in the language reference manual, which defines the *Compaq Pascal* language.

1.2 Commands to Create and Run an Executable Program

Example 1–1 shows a short Pascal main program.

Example 1–1 Sample Main Program

```
! File hello.pas
PROGRAM HELLO(output);
BEGIN
  writeln ('hello world')
END.
```

To create and revise your source files, use a text editor, such as `vi` or `emacs`. For instance, to use `vi` to edit the file `hello.pas`, type:

```
% vi hello.pas
```

The following `pc` command compiles the program named `hello.pas` and automatically uses `ld` to link the main program into an executable program file named `a.out`:

```
% pc hello.pas
```

The `pc` command automatically passes a standard default list of *Tru64 UNIX* and *Compaq Pascal* libraries to the `ld` linker. In this example, because all external routines used by this program reside in these standard libraries, additional libraries or object files are not specified on the `pc` command line.

If your path definition includes the directory containing `a.out`, you can run the program by simply typing its name:

```
% a.out
```

If the executable image is not in your current directory path, specify the directory path in addition to the file name.

When compiling the program, you can use a `pc` command-line flag to specify a name other than `a.out` for the executable program.

1.3 Creating and Running a Program Using a Separate Function

Example 1–2 shows a sample Pascal main program that uses an external function.

The function CALC_AVERAGE, as shown in Example 1–3, is contained in a separately created file.

Example 1–2 Sample Main Program That Uses a Separate Function

```
! Main_average.pas
PROGRAM Main_Average(Input, Output);

TYPE
   Arr_Type = ARRAY [1..5] OF Real;

VAR
   Arr : Arr_Type;
   I   : Integer;

[EXTERNAL] FUNCTION Calc_Average (Arr : Arr_Type) : Real; External;

BEGIN
  Write(' Enter 5 numbers: ');
  FOR I : = 1 to 5 DO
      Read(Arr[I]);
  Writeln(' The average is ', Calc_Average(Arr) :1)
END.
```

Example 1–3 Sample Separate Function Declaration

```
! Calc_average.pas
MODULE  Calc_Average;

TYPE
   Arr_Type = ARRAY [1..5] OF Real;

VAR
   I   : Integer;
   Sum : Real;
[GLOBAL] function calc_average (arr:arr_type):Real;

BEGIN
  Sum := 0.0;
  FOR I : = 1 to 5 DO
      Sum := Sum + Arr[I];
  Average := Sum / 5.0;
END;

END.
```

1.3.1 Creating the Executable Program

Edit each source file with a text editor, such as `vi` or `emacs`.

During the early stages of program development, the two sample program files might be compiled separately and then linked together, using the following commands:

```
% pc -c calc_aver.pas
% pc -c main_average.pas
% pc -o calc main_average.o calc_aver.o
```

In this sequence of `pc` commands:

- The `-c` flag (used in the first two commands) prevents linking and retains the `.o` files.
- The first command creates the file `calc_aver.o`.
- The second command creates the file `main_average.o`.
- The last command links all object files into the executable program named `calc`. To link files, use the `pc` command instead of the `ld` command.

The two source files can be compiled and linked together with a single `pc` command:

```
% pc -o calc calc_aver.pas main_average.pas
```

This `pc` command:

- Compiles the file `calc_aver.pas`, which contains the external function CALC_AVERAGE.
- Uses `ld` to link the main program and all object files into an executable program file named `calc`.

1.3.2 Running the Sample Program

If your path definition includes the directory containing `calc`, you can run the program by simply typing its name:

```
Enter 5 numbers: 1 2 3 4 5
The average is 3.0E+00
```

1.4 The pc Command and Related Software Components

It provides the standard features of a compiler and linker. *Compaq Pascal* also supports the use of preprocessors and other compilers.

Compiling and linking are usually done by a single `pc` command. The `pc` command allows you to use:

- The `cpp` preprocessors (if needed)
- The *Compaq Pascal* compiler
- The `cc` compiler (if needed)
- The `ld` linker

1.4.1 The Driver Program

The `pc` command invokes a **driver program** that is the actual user interface to the *Compaq Pascal* compiler. It accepts a list of command flags and file names and causes one or more programs (preprocessor, compiler, assembler, or linker) to process each file.

After the *Compaq Pascal* compiler processes the appropriate files to create one or more object files, the driver program passes a list of files, certain flags, and other information to the `cc` compiler. The `cc` compiler processes relevant non-Pascal files and information (including running `cpp` on C language files) and passes certain information (such as `.o` object files) to the `ld` linker.

The `pc` command executes each related program (preprocessor, compiler, assembler, or linker) in a sequential manner. If the `cpp` preprocessor is used, the preprocessor is executed once for each file.

If any processor does not return a normal status, further processing is discontinued and the `pc` command displays a message identifying the program (and its returned status, in hexadecimal) before terminating its own execution.

1.4.2 Compaq Pascal Compiler

The *Compaq Pascal* compiler provides the following primary functions:

- Verifying the correctness of *Compaq Pascal* source statements and displaying any warnings or error messages.
- Generating machine-level object language instructions from the source statements.
- Grouping the instructions to generate an object file that can be processed by the linker.

The object file created by the compiler contains information used by the linker, including the following:

- *A list of global symbols declared in the object file*

  The linker uses this information when it binds two or more program units together and must resolve references to the same names in the program units. Such global symbols include entry points and common block names.
- *A symbol table* (if specifically requested by the `-g`, `-g2`, or `-g3` flags on the `pc` command line)

  A symbol table lists the names of all external and internal variables within an object file, with definitions of their locations. The table is of primary use in program debugging.

The file name of the *Compaq Pascal* compiler is `decpascal`, which may appear in certain messages.

1.4.3 Other Compilers

You can compile and link multilanguage programs using a single `pc` command.

The `pc` command recognizes C or Assembler program files by their file suffix characters and passes them to the `cc` driver and compiler for compilation. Before compilation, `cc` applies the `cpp` preprocessor to files that it recognizes, such as any file with a `.c` suffix, and passes appropriate files to other compilers or the assembler.

Certain flags passed to `cc` are passed by `cc` to the `ld` linker.

1.4.4 Linker (ld)

When you enter a `pc` command, the `ld` linker is invoked automatically unless a compilation error occurs or you specify the `-c` flag on the command line. The linker produces an executable program image with a default name of `a.out`.

The `ld` linker provides such primary functions as:

- Adding information for virtual memory allocation in the executable program
- Resolving symbol references among object files
- Assigning values to relocatable global symbols
- Performing relocation

The `ld` linker on *Tru64 UNIX* systems performs other functions related to shared object libraries.

1.5 Program Development Stages and Tools

This manual primarily addresses the program development activities associated with implementation and testing phases. For information about topics usually considered during application design, specification, and maintenance, see your operating system documentation, appropriate reference pages, or appropriate commercially published documentation.

Table 1–1 lists and describes some of the software tools you can use when developing and testing a program:

Table 1–1 Main Tools for Program Development and Testing

Task or Activity	Tool and Description
Manage source files	Use `rcs` or `SCCS` to manage source files. For more information, see the *Tru64 UNIX Using Programming Support Tools* or the appropriate reference page.
Create and modify source files	Use a text editor, such as `vi` or `emacs`. For more information, see your operating system documentation.
Analyze source code	Use searching commands such as `grep` and `diff`. For more information, see the *Tru64 UNIX Using Programming Support Tools* and the appropriate reference page.

(continued on next page)

Table 1–1 (Cont.) Main Tools for Program Development and Testing

Task or Activity	Tool and Description
Build program (compile and link)	You can use the `pc` command to create small programs, perhaps using shell scripts, or use the `make` command to build your application in an automated fashion using a **makefile**.
Debug and Test program	Use `dbx` to debug your program or run it for general testing.
Analyze performance	To perform profiling of code, use the `prof` and `pixie` programs. The `pc` command flag needed to use `prof` is `-p` (same as `-p1`).
	To perform call graph profiling, use the `gprof` tool. The `pc` command flag needed to use `gprof` is `-pg`.
	Related profiling tools include `cord` and the use of feedback files.
Install program	Use `setld` and related commands such as `tar`. For more information, see the *Tru64 UNIX Using Programming Support Tools*.

To view information about an object file or an object library, use the following shell commands:

- The `file` command shows the type of a file (such as which programming language, whether it is an object library, ASCII file, and so forth).
- The `strings` command shows whether the object (`.o`) file was compiled by *Compaq Pascal* and if it was, the version number used.
- The `nm` command shows symbol table information, including the identification field of each object file.
- The `odump` command shows the contents of a file and other information.
- The `size` command shows the size of the code and data sections.

For more information on these commands, see the appropriate reference page or the *Tru64 UNIX Programmer's Guide*.

To perform other program development functions at various stages of program development:

- Use the `ar` command to:
  - Create an archive object library.
  - Maintain the object modules in the library.
  - List the object modules in the library.
  - Perform other functions.

  Use `ranlib` to add a table of contents to the object library for linking purposes. For more information, see `ar`(1) or the *Tru64 UNIX Programmer's Guide*.
- Use `pc` or `ld`, not the `ar` command, to create shared libraries on *Tru64 UNIX* systems. For more information, see the *Tru64 UNIX Programmer's Guide*.
- Use the `strip` command to remove symbolic and other debugging information to minimize image size. For more information, see `strip`(1).

2 Compiling and Linking Compaq Pascal Programs

This chapter provides information on the following topics:

- The `pc` command (Section 2.1)
- Compiler options (Section 2.2.2)
- Linking Pascal programs (Section 2.3)
- The format of the `pc` command, the file name suffix characters that identify the type of file, and `pc` command options related to input and output files (Section 2.1)
- Detailed information on `pc` command options, including option categories, a description of each option, handling run-time exceptions, and using libraries (Section 2.2)

2.1 The pc Command

In most instances, use the `pc` command to invoke both the *Compaq Pascal* compiler and the `ld` linker. To link one or more object files created by the *Compaq Pascal* compiler, you should use the `pc` command instead of the `ld` command, because the `pc` command automatically references the appropriate *Compaq Pascal* Run-Time Libraries when it invokes `ld`.

When you create your source files using a text editor, use file name suffix conventions expected by the `pc` command.

Compaq Pascal programs usually have a file name suffix of `.p`,`.P`, `.pas`, or `.PAS`, and will be automatically passed to the C language preprocessor `cpp` before compilation.

The `pc` command has the following form:

pc [–*options* [*args*]]... *filename* [*filename*] ... [–*options* [[*args*]]...

***–options* [*args*]**
Indicates either special actions to be performed by the compiler or linker, or special properties of input or output files. For details about command line options, see Section 2.2. If you specify the `-l`*string* option (which indicates libraries to be searched by the linker) or an object library file name, place it after the file names and after other options.

filename
Specifies the source files containing the program units to be compiled, where the file name has a suffix that indicates the type of file used.

If you omit the suffix or it is not one of the preceding types recognized by the `pc` command, the file is assumed to be an object file and is passed directly to the linker.

An example `pc` command line follows:

```
% pc -v test.p calc_loop.o -ldxml
```

This command specifies the following:

- The `-v` option displays the compilation and link passes with their arguments and files, including the libraries passed to `ld`.
- The file `test.p` is preprocess by `cpp` and then passed to the *Compaq Pascal* compiler for compilation. The resulting object file is passed to the linker.
- The object file `calc_loop.o` is passed to the linker. The linker links both object files into an executable program.

2.1.1 pc Command Examples

The following examples show the use of the `pc` command.

Compiling and Linking Multiple Files

The following `pc` command compiles the *Compaq Pascal* source files (`aaa.p`, `bbb.p`, `ccc.p`) into three temporary object files:

```
% pc -V aaa.p bbb.p ccc.p
```

Next, the `ld` linker is invoked and passes the three temporary object files, which it uses to produce the executable file `a.out`. The *Compaq Pascal* compiler (`-V` option) creates the listing file of each source file.

The following `pc` command compiles all *Compaq Pascal* source files with file names that end with `.p` into individual object files:

```
% pc *.p
```

The `ld` linker produces the `a.out` file from these object files.

Retaining an Object File and Preventing Linking

The following pc command compiles, but does not link, the source file typedefs_1.pas:

```
% pc -c typedefs_1.pas
```

Specifying the -c option retains the object file named typedefs_1.o and prevents linking.

Compiling Pascal and C Source Files and Linking an Object File

The following pc command compiles the *Compaq Pascal* main program (myprog.pas). The main program calls a function written in C and uses the object file created in the previous example. The C routine named utilityx_ is declared in a file named utilityx.c. All sources files are compiled and the object files are passed to the linker:

```
% pc myprog.pas typedefs_1.o utilityx.c
```

This command does the following:

- Compiles myprog.pas with the *Compaq Pascal* compiler.
- The C compiler compiles utilityx.c.
- The ld linker links all three object files together into the executable program named a.out.

Renaming the Output File

The following pc command compiles multiple source files and produces an executable program named circle.out.

```
% pc -o circle.out circle-calc.pas sub.pas
```

Requesting Additional Optimizations

The following pc command compiles the *Compaq Pascal* source files circle-calc.pas and sub.pas together using software pipelining optimizations (-O5):

```
% pc -O5 -unroll 3 circle-calc.pas sub.pas
```

The loops within the program are unrolled 3 times (-unroll 3). Loop unrolling occurs at optimization level -O3 or above.

Loop Unrolling Optimization Control

The *Compaq Pascal* compiler automatically unrolls loops for better performance. Loop unrolling involves making multiple copies of loop bodies to allow the instruction scheduler to schedule more instructions between branches. By default, loop unrolling makes four copies of an unrolled loop. You can change the number of copies from 1 to 16 by using the following on the `pc` command line:

```
-unroll "number"
```

Numbers larger than 4 may improve performance at a cost of additional code size. However, larger numbers may decrease performance due to cache requirements, register conflicts.

Software Pipelining Optimization

Software pipelining and additional software dependency analysis are enabled using the `-O5` command line option, which in certain cases improves run-time performance. `-O5` is not the default; `-O4` remains the default.

As compared to regular loop unrolling (enabled at optimization level 3 or above), software pipelining uses instruction scheduling to eliminate instruction stalls within loops, rearranging instructions between different unrolled loop iterations to improve performance.

For instance, if software dependency anaylsis of data flow reveals that certain calculations can be done before or after that iteration of the unrolled loop, software pipelining reschedules those instructions ahead or behind that loop iteration at places where their execution can prevent instruction stalls or otherwise improve performance.

Loops chosen for software pipleling for *Compaq Pascal* :

- Are always innermost loops, those executed the most
- Do not contain branches of procedure calls

2.2 pc Command Options

Options to the `pc` command affect how the compiler processes a file in conjunction with the file name suffix information.

You can override some options specified on the command line by using attributes in your Pascal source program.

If you compile parts of your program by using multiple `pc` commands, options that affect the execution of the program should be used consistently for all compilations, especially if data will be shared or passed between procedures. For example, the same data alignment needs to be used for data passed or shared by included files (such as record structures). Use the same version of the `-align` option for all compilations.

Some options consist of two words separated by a space. For options that consist of two words separated by a space, the second word can be abbreviated. For example, you can abbreviate `-C bounds` to `-C b` (usually 4 characters or more). Table 2–1 lists the various options, their applicable categories, and the section in which they are described in detail.

2.2.1 pc Command Option Categories

Table 2–1 categorizes the available options to the `pc` command. For information on the individual options, see section Section 2.2.2.

Table 2–1 Categories of pc Command Options

Category	Option Name

Compiler files and passes	`-c`
	`-o` *file*
	`-v`
	`-env`

Data size defaults and alignments	`-align` *keyword* `-enumeration_size` *keyword*
Debugging and symbol table use	`-g`, `-g0`, `-g1`, `-g2`, `-g3`
Floating-point exceptions, and accuracy	`-assume accuracy_sensitive`, `-C` *keyword*, `-math_library` *keyword*, `-synchronous_exceptions`
Language compatibility	`-std` *keyword* `-platforms` *keyword*

Linker and library searching	`-non_shared`
	`-l`*string*
	`-L`
	`-L`*dir*
	`-v`

Listing file and contents	`-show` *keyword* `-V`

(continued on next page)

Table 2–1 (Cont.) Categories of pc Command Options

Category	Option Name

Performance and optimizations	`-align` *keyword*
	`-arch` *keyword*
	`-inline` *keyword*, `-noinline`
	`-O`, `-O1`, `-O2`, `-O3`, `-O4`, `-O5`
	`-synchronous_exceptions`
	`-unroll` *nn*
	`-instruction_set floating_point`
	`-nozero_heap`
	`-tune` *keyword*
	`-usage performance`
	`-om`

Preprocessor and source file searching	`-constant` *name*=*value*
	`-cpp`
	`-Dname`, `-Dname=def`
	`-E`
	`-I`
	`-I`*dir*
	`-nocpp`

Profiling, feedback files, and `cord`	`-cord`, `-feedback` *file*, `-p0`, `-p1` or `-p`, `-pg`, `-gen_feedback`
Run-time checking and messages	`-C` *keyword* `-synchronous_exceptions`
Shared data use	`-granularity` *keyword*
Variable usage information	`-usage` *keyword*
Warning messages at compile time	`-error_limit` *keyword* `-nowarn`

2.2.2 pc Command Options Descriptions

The following sections list the *Compaq Pascal* options supported by the `pc` command.

-align *keyword*

Controls the default alignment rules. Specifying the ALIGN attribute overrides any value that was previously specified for the `-align` option.

Table 2–2 lists the values for keyword.

Table 2–2 -align Option Keywords

Keyword	Action	Default Information
`alpha_axp`	Uses natural alignment when positioning record fields or array components. Natural alignment is when a record field or an array component is positioned on a boundary based on its size. For example, 32-bit integers are aligned on the nearest 32-bit boundary.	Default—if `-align` is not specified.
`vax`	Uses byte alignment when positioning record fields or array components. Record fields or array components larger than 32 bits are positioned on the nearest byte boundary.	

-arch *keyword*

Specifies the lowest version of the Alpha architecture where this code will run and directs the compiler to generate the most efficient code, with the tradeoff that code may not run on older systems (unlike the `-tune` option).

All Alpha processors implement a core set of instructions and, in some cases, the following extensions: BWX (byte- and word- manipulation instructions) and MAX (multimedia instructions). (The *Alpha Architecture Reference Manual* describes the extensions in detail.)

The option specified by the `-arch` flag determines the instructions the compiler can generate and the coding rules it must follow, as shown in Table 2–3.

Table 2–3 -arch Option Keywords

Keyword	Action	Default Information
generic	Generate instructions that are appropriate for all Alpha processors.	Default; equivalent to `-arch ev4`.
host	Generate instructions for the processor on which the compiler is running (for example, EV56 instructions on an EV56 processor, and EV4 instructions on an EV4 processor).	

(continued on next page)

Table 2–3 (Cont.) -arch Option Keywords

Keyword	Action	Default Information
ev4	Generate instructions for the EV4 processor (21064, 20164A, 21066, and 21068 chips).	
ev5	Generate instructions for the EV5 processor (some 21164 chips). (Note that the EV5 and EV56 processors both have the same chip number: 21164.)	
ev56	Generate instructions for EV56 processors (some 21164 chips). This option permits the compiler to generate any EV4 instruction. Applications compiled with this option may incur emulation overhead on EV4 and EV5 processors.	
pca56	Generate instructions for PCA56 processors (21164PC chips). This option permits the compiler to generate any EV4 instruction, plus any instructions contained in the BWX or MAX extensions. However, *Compaq Pascal* does not generate any of the instructions in the MAX (multimedia) extension to the Alpha architecture. Applications compiled with this option may incur emulation overhead on EV4 and EV5 processors.	
ev6	Generate instructions for EV6 processors (21264 chips). This option permits the compiler to generate any EV4 instruction, any instruction contained in the BWX and MAX extensions, plus any instructions added for the EV6 chip. These new instructions include a floating-point square root instruction (SQRT), integer/floating-point register transfer instructions, and additional instructions to identify extensions and processor groups. Applications compiled with this option may incur emulation overhead on EV4, EV5, EV56, and PCA56 processors.	

A program compiled with any of the above options runs on any Alpha processor with a superset of capabilities compared to the specified target processor. In many cases, code will also move to older processors, although potentially with significant overhead due to emulation, provided the processor is running a (recent) version of the operating system that provides emulation. In particular, Digital UNIX V4.0 emulates the BWX instruction set group on older processors, so `-arch ev56` code will run on ev4 and ev5 systems (except for `-ieee` code).

Compaq very strongly discourages compiling code with `-arch` that is above any projected target system for an application. You should compile for the lowest-level architecture on which your application will run. The `-tune` option should be used for performance tuning.

The `-arch` and `-tune` options can be used together to describe the lower bound and the common case for compiling code. For example, `-arch ev5 -tune ev6` specifies that the code will never run on a processor older than an EV5 system, but should be tuned to run on ev6 systems preferentially.

Note that `-tune` may cause the compiler to generate instructions not present in the lowest-specified architecture level, but if it does, it will generate code such that the new instructions will be guarded against execution on older processors.

Note also that `-tune` defaults to the value of `-arch`, if that value isn't specified.

The psrinfo `-v` command can be used to determine which type of processor is installed on any given Alpha system.

-assume accuracy_sensitive

Specifies whether certain code transformations that affect floating-point operations are allowed. These changes can affect the accuracy of the program results.

If you specify noaccuracy_sensitive, the compiler is free to reorder floating-point operations based on algebraic identities (inverses, associativity, and distribution). This allows the compiler to move additional floating-point operations outside of loops, or reduce or remove floating-point operations totally, thereby improving performance.

The default, `-assume accuracy_sensitive`, directs the compiler to avoid certain floating-point trasformations that might slightly affect the program accuracy.

-B

Strip C++ style comments during preprocessing.

See manpage `pc`(1).

-C *keyword*

Performs run-time checks.

The arguments to the `-C` option, listed in Table 2–4, control which aspects of the compilation unit the compiler will produce checking code for. No checking is the default action of the compiler.

Table 2–4 -C Option Keywords

Keyword	Action
`all`	Turns on all checking options.
`bounds`	Verifies that an index expression is within the bounds of an array's index type, that character-string sizes are compatible with the operations being performed, and that schemata are compatible.
`case_selectors`	Verifies that the value of a case selector is contained in the corresponding case-label list.
`declarations`	Verifies that schema definitions yield valid types and that uses of GOTO from one block to an enclosing block are correct.
`overflow`	Verifies that the result of an integer computation does not exceed the machine representation.
`pointers`	Verifies that the value of a pointer variable is not NIL.
`subrange`	Verifies that the values assigned to variables of subrange types are within the subrange; verifies that a set expression is assignment compatiable with a set variable; verifies that MOD operates on positive numbers.

-c

Retains the object file and prevents linking.

The following `pc` command compiles, but does not link, the source file `typedefs_1.pas`, retaining the object file `typedefs_1.o`:

```
% pc -c typedefs_1.pas
```

-call_shared

The `-call_shared` option specifies that the linker searches `.so` files before `.a` files. References to symbols found in a `.so` library are dynamically loaded into memory at run time. References to symbols found in `.a` libraries are loaded into the executable image file link time.

For More Information:

- About creating a shared library using either the `pc` command or the `ld` command (Section 2.4)
- On maintaining shared libraries, see your operating system documentation
- On shared library processing, see `-shared` and `-non_shared` sections

-constant *name=value*

Allows a limited set of Pascal constants to be defined from the command line. This feature can be used with the conditional-compilation facility, and can also be used in any situation where defining constants from the command line is useful.

name is the name of a Pascal constant to create. You cannot redefine any predeclared Pascal name on the command line.

value can be one of the following:

- integer-literal
- –integer-literal
- TRUE
- FALSE
- "string-literal"
- 'string-literal'

For example:

```
-constant debug=true -constant maxsize=10 -constant ident="V1.0"
```

Non-base-10 integer literals are not supported.

-cord

Runs the cord procedure-rearranger on the resulting file after linking. The rearrangement reduces the cache conflicts of the program's text. The output of cord is left in the file specified by the `-o` option or `a.out`, by default. At least one `-feedback` file must be specified.

-ccp

Runs the `cpp` preprocessor on Pascal source files before compiling. This is the default.

-D*name=def*, -D*name*

Defines preprocessor symbol with the specified value, if given.
The `-E`, `-P`, and `-nocpp` options also affect the preprocessing phase.

Typical *Compaq Pascal* compilations first submit a Pascal source file to the `cpp` C language preprocessor, which enables you to define symbols and specify conditional compilation. The preprocessor does string substitutions and adds or deletes source file lines.

In a typical use of the preprocessor, a header file defines (or fails to define) a certain symbol. Many Pascal source files include this header file. By changing a preprocessor declaration in this single header file, you can change many different compilations. For example, an application might contain lines such as these in many locations in many source files:

```
#ifdef debug
 WRITELN(ErrorLogFile, 'Error occurred at ', TimeOfDay, ErrorType);
#endif
```

If the preprocessor symbol `debug` is defined for compilation, then the WRITELN commands are compiled.

The `-D` option defines a preprocessor symbol and allows you to set its value. Using the command-line option may be easier than editing header files to produce customized source files.

Do not type a space after `-D`, but follow it directly with the symbol to define. Rememeber that, as in the C language, preprocessor symbols are case sensitive. For example `ABC` and `abc` are different symbols.

If you then follow `ABC` or `abc` by an equal sign (`=`) and a value, the preprocessor symbol is set to that value. If you do not specify a value in this way, the symbol takes the value one (1). The value of a preprocessor symbol is sometimes irrelevant; a source file that uses `#ifdef` tests only whether the symbol is defined.

In the previous example, the WRITELN statement was conditional, based on whether the preprocessor symbol `debug` was defined. To compile a module and include such WRITELN statements, you can invoke `pc` by using one of these equivalent forms:

```
% pc -Ddebug sourcefile.p
```

or,

```
% pc -Ddebug=1 sourcefile.p
```

You can accomplish much of the same functionality with the conditional compilation syntax that is built into the compiler (with %IF), and the `-constant` option.

For More Information:

- On `cpp` (Section 5.1)

-E
Preprocesses only; send result to standard output.

-enumeration_size *keyword*
Controls the allocation of unpacked enumerated data types and Boolean data types, which are considered to be an enumerated type containing two elements for allocation purposes.

Table 2–5 lists the available values for keyword.

Table 2–5 -enumeration_size Option Keywords

Value	Action	Default Information
`byte`	Allocates unpacked enumerated data types with up to 255 elements in a single byte. Otherwise, enumerated data types are allocated in a 16-bit word.	
`long`	Allocates all unpacked enumerated data types in a 32-bit longword.	Default

-error_limit *count*
Stops compilation after *count* errors.

Terminates compilation after the occurrence of a specified number of error messages, excluding warning-level and information-level errors. The default is to stop the compilation after 30 errors.

-env *file*
Produces a file containing precompressed symbol table information for subsequent *Compaq Pascal* with the inherit attribute.

Produces an environment file in which declarations and definitions made at the outermost level of a compilation unit are saved.

The `-env` option on the command line overrides the ENVIRONMENT attribute in the source program or module. By default, the attributes of the source program or module determine whether an environment file is created; however, if the `-env` option is specified at compile time, an environment file will always be created.

-feedback *file*
Specifies the feedback file for the `-cord` option.

The `prof` command produces the feedback file with its `-feedback` option from an execution of the program produced by pixie.

For More Information:

- On profiling (Section 4.5)

-G*num*

Specify maximum size of data for global pointer area.

For More Information:

- On global pointer area (Section 4.6)

-g*[n]*

Specifies the amount of symbol table information to be produced for debugging.

Source-level debugging relates the run-time operation of your program to the source files. For this kind of debugging, your executable program must contain information that relates the run-time actions to elements from the source file, such as variable identifiers. The `-g` option on the `pc` command inserts this information into the executable program to assist in source-level debugging.

glevel	Action
`-g0`	The default. Do not produce symbol table information for symbolic debugging.
`-g1`	Produce additional symbol table information for accurate, but limited, symbolic debugging of partially optimized code.
`-g` or `-g2`	Produce additional symbol table information for full symbolic debugging and disable optimizations that limit full symbolic debugging. This option also sets the `-O0` option to disable optimization unless an explicit `-O` option is specified.
`-g3`	Produce additional symbol table information for full symbolic debugging for fully optimized code. This option produces additional debugging information describing the effects of optimizations, but debugging inaccuracices may occur as a result of the optimizations that have been performed.

If you do not use the `-g` option, you can still use any of the following `dbx` commands:

- `conti`
- `continue`
- `stepi`
- `stop in` *procedure*
- `tracei`

- *address/count mode*

-granularity *keyword*
Directs the compiler to generate additional code to preserve the indicated granularity. Granularity refers to the amount of storage that can be modified when updating a variable. You can specify the following values for keyword:

Table 2–6 -granularity Option Keywords

Value	Action	Default Information
`byte`	Generate additional code to provide byte granularity for data access. This option can impose a large run-time performance penalty.	
`long`	Generate additional code to provide longword granularity for data accesses. This option only imposes a slight run-time performance penalty.	
`quadword`	Generate code that assumes quadword granularity. This option yields the best code quality.	Default

To update a variable that is smaller than a longword, *Compaq Pascal* may issue multiple instructions to fetch the surrounding longword or quadword, update the memory inside to longword or quadword, and then write the longword or quadword back into memory. If multiple processes are writing into memory that is contained in the same longword or quadword, you might incur inaccurate results, unless `-granularity byte` or some other synchronization mechanism is used. For additional information, refer to the `-arch` option to enable instructions to access byte and word data.

-I
Omit `/usr/include` from include file search path for `#include` directives.

A Pascal source file may include the text of other source files by using the *Compaq Pascal* `#include` directive, which is processed by `cpp`, or the *Compaq Pascal* %INCLUDE directive. The `-I` option also affects the *Compaq Pascal* %INCLUDE directive and the [INHERIT] attribute.

If you use the `#include` directive, the `cpp` preprocessor copies the text of the named file to the point where the directive appears. For more information about the `#include` directive, see `cpp(1)`. The directive can take two forms:

```
#include <filespec>

#include "filespec"
```

The preprocessor uses different search paths depending on whether you use angle brackets or double quotes for delimiters. You can change these search paths with the `-I` option on the `pc` command.

If the file name begins with a slash, the preprocessor looks for the file only in the directory specified. The `-I` option does not affect such files.

-I*dir*
Adds a directory to include a search path for `cpp #include` directives, Pascal %INCLUDE directives, and [INHERIT] attributes.

If the specified file is not found in the current directory, the compiler searches as follows:

1. The current directory with a default extension of .pas for %INCLUDE directives and .pen for [INHERIT] attributes
2. Any directories specified with `-I` (in the order specified) with the appropriate default extension
3. `/usr/include` with the appropriate default extension

-inline *keyword*
Controls the algorithm that the optimizer chooses for inlining routines. The default is to choose routines to inline that will maximize performance.

The choices for keyword are:

Table 2–7 -inline Option Keywords

Keyword	Action
`none`	No inlining is done
`manual`	Only routines with the [OPTIMIZE(INLINE)] attribute are inlined
`size`	Perform manual inlining and inline other calls that are likely to improve performance without significantly increasing code size
`speed`	Perform manual inlining and inline any other calls that are likely to improve performance (default)
`all`	Inline every call that can be inlined

-instruction_set nofloating_point
Controls whether the compiler can use any floating point instructions.

-K
Does not remove temporary files created during compilation and linking.

-L
Determines library search path.

To find libraries, the linker searches the following directories in the order listed in Table 2–8.

Table 2–8 Library Search Path

Order	Directory
1st	`/lib`
2nd	`/usr/lib/cmplrs/pc`
3rd	`/usr/lib`
4th	`/usr/local/lib`

-L*string*
Searches *string* libraries for `ld`. This option should be placed at the end of the command line.

-M
Tells `cpp` to generate dependency lists for `#include` files for `make`.

-math_library *keyword*
Determines whether the compiler uses alternate math library routines that boost performance, but sacrifice accuracy. You can specify the following values for keyword:

- `accurate` (default)
- `fast`

-nocpp
Inhibits preprocessing by `cpp`.

Normally, the `cpp` preprocessor automatically applies to every source file (Pascal, C, or assembler) that you specify in the `pc` command. You can disable the preprocessor by specifying the `-nocpp` option on the `pc` command. This speeds the compilation unless you use `cpp` commands such as `#define` and `#include`, which are nonstandard features that would produce compilation errors.

-non_shared
The `-non_shared` option specifies that the linker searches only `.a` files. The object module created contains static references to external routines. The references are loaded into the executable image to link time not at run

time. The following example requests that `.a` files be searched instead of the corresponding `.so` files:

```
%  pc -non_shared main.p rest.o
```

For More Information:

- On shared library processing, see `-call-shared` and `-shared` sections

-nowarn
Suppresses all warning messages.

-nozero_heap
Specifies that heap memory should not be zeroed after allocation. By default, the Pascal RTL will return zero-filled memory for each call to the NEW builtin. Using the `-nozero_heap` option can increase runtime performance.

-O*[n]*
Enables or disables optimizations and sets the optimization level.

At optimization level zero, the compiler may produce additional instructions intended to result in code that is easier to debug and that provides more predictable program behavior when run-time exceptions occur. Compilers normally translate source statements to machine instructions, regardless of the context. However, optimization produces smaller and faster programs by merging nearby statements and using the results of previous statements.

The `pc` command can optimize at five levels, selectable through the `-O` option:

- `-O0` disables all optimizations by the compiler. Specify `-O0` when you want to use the `dbx` interactive debugger.
- `-O1` enables local optimizations and recognition of common subexpressions. The call graph determines the order of compilation for procedures. This level is the default when you compile without the `-O` option.
- `-O2` enables all `-O1` optimizations and additional global optimizations, including code motion, strength reduction and test replacement, split lifetime analysis, and code scheduling.
- `-O3` enables all `-O2` optimizations and performs global optimization across the entire body of compiled code, including optimizations that improve speed at the cost of extra code size, for example, integer multiplication and division expansion (using shifts), loop unrolling, and code replication to eliminate branches.
- `-O` or `-O4` enables all `-O3` optimizations and also enables inline expansion of procedure and functions. `-O4` is the default.

- `-O5` enables software pipelining and additional software dependency analysis, which in certain cases improves run-time performance.

Any optimization increases compilation time, so you may not want to specify it during development. However, when the application is complete, specifying optimization (despite the slower compilation) saves space and execution time when you run the application.

Optimization never affects program operation yet it is not advisable for all stages of program development.

Optimization reduces the file size and increases the execution speed of your program. However, you should not specify it when using the `dbx` interactive debugger. If you do, the resulting rearragement of coding will create incorrect debugger output.

For More Information:

- On program optimization and efficiency (Chapter 4)
- On the `dbx` debugger (Chapter 5)
- On how *Compaq Pascal* optimizes programs (Chapter 5)

-o *file-name*

Names the output file as *file-name*.

`pc` produces an executable file called, by default, `a.out`. You can use the `-o` option to specify a name for the output of the `pc` command. Follow `-o` with a space and then the desired file name. You can specify a pathname to place the output file someplace other than the current working directory. By default, *Compaq Pascal* searches for input files from, and writes any output files to, the current working directory.

You can use the options to request an intermediate Pascal object file or an output module. The object files take the same name as the respective input file, but the suffix of the input source file is replaced with `.o`. The `-o` option sets the output file name for the files generated by the `-c` and the `-o` options.

-om

Performs code optimization after linking, including nop (No Operation) removal, .lita removal, and reallocation of common symbols. This option also positions the global pointer register so the maximum addresses fall in the global pointer-accessible window. The `-om` option is supported only for programs compiled with the `-non_shared` option.

The following options can be passed directly to `-om` by using the `-WL` compiler option:

- `-WL,-om_compress_lita`

  Removes unused .lita entries after optimization, and then compresses the .lita section.

- `-WL,-om_dead_code`

  Removes dead code (unreachable instructions) generated after applying optimizations. The .lita section is not compressed by this option.

- `-WL,-om_no_inst_sched`

  Turns off instruction scheduling.

- `-WL,-om_no_align_labels`

  Turns off alignment of labels. Normally, the `-om` option will quadword align the targets of all branches to improve loop performance.

- `-WL,-om_Gcommon,num`

  Sets size threshold of "common" symbols. Every "common" symbol whose size is less than or equal to num will be allocated close to each other. This option can be used to improve the probability that the symbol can be accessed directory from the global pointer register. Normally, the om tries to collect all "common" symbols together.

-P

Preprocesses only; put the result in a `.i` file.

When debugging a Pascal program, you sometimes need to see the preprocessor output. For example, with conditional compilation, you may want to see which lines of code the preprocessor passed through to the compiler. The `-E` option runs the preprocessor on the specified source files, sends the output to the standard output, and exits without compiling anything.

The `-P` option also performs the preprocessing phase only, but writes the output to a file instead of to standard output. The name of the output file is the name of the source file with the suffix replaced by `.i`.

You can also invoke the preprocessing phase directly with the `cpp` command, but you must explicity define the symbol LANGUAGE_PASCAL (done automatically by the `pc` command). For example, the following commands are equivalent:

```
% pc -E module4.p > module4.i
% pc -P module4.p
% cpp -DLANGUAGE_PASCAL module4.p > module4.i
```

-p *[n]*
Disables or sets the levels of program-counter sampling for profiling.

For More Information:

- On program counter sampling (Section 4.5.1)

-platforms *keyword*
Displays informational messages about non-portable language features for the specified platform. Table 2–9 lists the supported platforms.

Table 2–9 -platforms Option Keywords

Keyword	Action
`common`	Displays informational messages for all platforms.
`OpenVMS_AXP`	Displays informational messages for the *OpenVMS Alpha* platform.
`OpenVMS_VAX`	Displays informational messages for the *OpenVMS VAX* platform.
`OSF1_AXP`	Displays informational messages for the *Tru64 UNIX* platform.

-pg
Sets up profiling for `gprof(1)`.

For More Information:

- On profiling (Section 4.5)

-shared
Creates position independent code (PIC) in the object module for inclusion in a shared library. If you specify the `-c` option to inhibit linking, `pc` creates a shareable object module `.o` file that can subsequently be processed by `ld` to create a shared library. If you omit the `-c` option, `pc` creates a shared library `.so` file. In either case, use the `-o` option to name the resulting object file or shared library with the correct file name and suffix.

External references found in an archive library result in that routine being included in the resulting executable program file at link time.

External references found in a shared object library result in a special link to that library being included in the resulting executable program file, and not the actual routine itself. When you run the program, this link gets resolved by either using the shared library in memory (if it already exists) or by loading it into memory from disk.

For More Information:

- About creating a shared library using either the `pc` command or the `ld` command (Section 2.4)
- On maintaining shared libraries, see your operating system documentation
- On shared library processing, see `-call_shared` and `-non_shared` sections

-show *keyword*
Specifies an item to be included in the listing file. A single identifier is expected.

Table 2–10 lists the available keywords, their corresponding actions, and their negations.

Table 2–10 -show Option Keywords

Keyword	Action	Negation
`all`	Enables listing of all options	`none`
`header`	Enables page headers	`noheader`
`include`	Enables listing of %INCLUDE files	`noinclude`
`machine_code`	Enables listing of machine code	`nomachine_code`
`source`	Enables listing of your program source code	`nosource`
`statistics`	Enables listing of compilation statistics	`nonstatistics`
`structure_layout`	Enables listing of the sizes, record, field offsets, and comments about non-optimal performance for variables and types in your program	`nostructure_layout`
`xref`	Enables listing of cross reference	`noxref`

The compiler ignores the `-show` option if you do not also specify the `-V` option on the same command line.

If the `-show` is not specified on the command line the listing will contain source, statistics and include file listings.

-std *keyword*
Issues warnings for violations of rules of the named standard.

Compaq Pascal extends standard Pascal in many areas. For example, *Compaq Pascal* provices schema types, separate compilation, and typesetting. The *Compaq Pascal Language Reference Manual* lists and describes *Compaq Pascal* extensions to standard Pascal.

The `-std` option makes the compiler print an informational message whenever you use a feature that is not in standard Pascal. Programs written for compatibility with ANSI-standard Pascal compilers should be compiled with `-std`. You can then correct the causes of any resulting informational messages. By confining nonstandard features to a single machine-specific source file, you simplify the job of adapting your program for other Pascal compilers.

Table 2–11 lists the possible values for `-std` option.

Table 2–11 -std Option Keywords

Keyword	Action
`ansi`	ANSI/IEEE703X3.97-1989
`ansi_validate`	Issue error level messages instead of default informational level messages.
`iso`	ISO 7185-1989
`iso_validate`	Issue error level messages instead of default informational level messages.
`extended`	ISO 10206-1989
`extended_validate`	Issue error level messages instead of default informational level messages.

-synchronous_exceptions
Specifies that the compiler should generate code to insure that exceptions are reported as near as possible to the instruction that generated the exception. This can avoid confusion in tracing the source of an exception, however, there is a performance penalty for using this option.

-tune *keyword*
Selects processor-specific instruction tuning for a specific implementation of the Alpha architecture. Tuning for a specific implementation can provide improvements in run-time performance.

It is important to note that regardless of the setting of the `-tune` flag, the generated code will run correctly on all implementations of the Alpha architecture, but performance may differ. For example, code that is optimally tuned for a specific target may run more slowly on another target (slower than generically-tuned code that is run on that target). Values for the processor type keyword are shown in Table 2–12.

Table 2–12 -tune Option Keywords

Keyword	Action
generic	Selects instruction tuning that is appropriate for all implementations of the Alpha architecture. This option is the default.
host	Selects instruction tuning that is appropriate for the machine on which the code is being compiled.
ev4	Selects instruction tuning for the 21064, 21064A, 21066, and 21068 implementations of the Alpha architecture.
ev5, ev56	Selects instruction tuning for the 21164 implementation of the Alpha architecture.
ev6	Selects instruction tuning for the 21264 implementation of the Alpha architecture.

-unroll *n*

Controls number of times loops are unrolled. The default is 4. `-unroll` 0 disables loop unrolling. Loop unrolling is only enabled above optimzation level 3.

See Section 2.1.1 for examples of using `-unroll`.

-usage *keyword*

Allows you to request information messages about unused or uninitialized variables, components of structures that result in unaligned accesses, incompatible alignment, and unsupported features on other platforms.

Table 2–13 lists available keywords for the `-usage` option.

Table 2–13 -usage Option Keywords

Keyword	Action	Negation
`all`	Enables the display of all usage information.	`noall`
`empty_records`	Detects usage of empty records that may not yield expected results.	`noempty_records`

(continued on next page)

Table 2–13 (Cont.) -usage Option Keywords

Keyword	Action	Negation
nongrnacc	Specifies whether the compiler should issue warning messages for code sequences when it cannot guarantee to match your granularity request from the `-granularity` option. When the compiler cannot guarantee that the generated code matches the granularity setting, it will issue a warning message. You should examine your program to make sure that the variable being accessed is quadword-aligned and is a multiple of quadwords in size. In that case, the resulting code will be correct even if the compiler could not determine that at compile time. Such cases involve pointer dereferences or VAR parameters. These messages are enabled by default by the compiler.	nonongrnacc
packed_actuals	Detects passing of fields of packed records or arrays to VAR parameters.	nopacked_actuals
performance	Detects variables, array components, and record fields that will generate less than optimal performance due to alignment or size considerations.	noperformance
uncallable	Specifies whether the compiler should issue informational messages for routines that are declared but never called. These messages are disabled by default.	
uncertain	Issues informational messages for components of structures that result in unaligned data accesses. Such accesses may cause poor run-time performance.	nouncertain
uninitialized	Detects variables that are uninitialized.	nouninitialized
unused	Detects variables that are unused.	nounused
volatile	Detects accesses to volatile data that cannot be protected as atomic operations.	novolatile

-V

Produces a source listing file with a file type of `.l`. See the `-show` option for more information on controlling the contents of the source listing file.

-v
Print the passes of the driver program during compilation and final resource usage.

The -v option displays the command for each pc phase. It also displays a resource usage message for each phase for calling time.

You might use the -v option when pc fails to produce expected output files. The messages from -v verify whether you correctly specified the compilation phases and object libraries.

2.3 Linking

The pc command automatically calls the ld linker unless you specify a partial operation, for example, with the -c option. The linker combines object modules to produce an executable program. The modules include:

- Object modules previously produced during the operation of pc
- Object modules in .o files that you name on the pc command line
- The Pascal run-time library
- Object modules from specified libraries

A pc command that has only object files as parameters does only a link. For example, such a command might appear in a make file that specifies a build procedure. This file separates the compile operations from the link so that make omits operations on unedited source files.

You should use the pc command instead of ld to link *Compaq Pascal* object files. The pc command automatically selects the correct run-time libraries and version of ld. However, if you need to specify options to ld, you must invoke ld directly. For more information, see ld(1) reference page.

2.3.1 Run-Time Libraries

The *Compaq Pascal* run-time library is libpas.a. The linker refers to this library whenever you invoke it through the pc command. If you are compiling (with the cc command) a C language file and wish to link with Pascal object files, the linker must use the Pascal run-time library. To specify this, use the -lpas option with the cc command.

To specify other libraries, use the -l option when you invoke the linker. For example, if your *Compaq Pascal* program calls routines from the library libc.a, specify:

```
% pc -lc ...
```

The administrator or installer determines the actual locations of run-time libraries.

The `-L` option to the linker, when followed by a directory name, adds that directory to the search path for run-time libraries. You must not type any separators between `-L` and the directory name, but you can use this option several times. The linker searches the directories in the order specified on the command line before it searches the standard directories listed previously. The search path definition must precede the specification of libraries on the command line (using the `-L` option.)

2.3.2 Specifying Shared Object Libraries

When you link your program with a shared library, all symbols must be referenced before `ld` searches the shared library. You must specify libraries at the end of the `pc` command line after all file names. On *Tru64 UNIX* systems, unless you specify the `-non_shared` option, shared libraries will be searched before the corresponding archive libraries.

For example, the following command generates an error if the file `rest.o` references routines in the library `libX`:

```
% pc -call_shared test.p -lX rest.o
```

The correct order is shown in this example:

```
% pc -call_shared test.p rest.o -lX
```

Link errors can occur with doubly defined symbols, for example, when both an archive and shared object are specified on the same command line. In general, specify archive libraries after the last file name, and specify shared libraries at the end of the command line.

Before you reference a shared library at run time, it must be installed.

2.4 Creating Shared Libraries

To create a shared library from a Pascal source file, process the files using the `pc` command and the following options:

- You must specify the `-shared` option to generate position-independent code.
- You must specify the `-o` option to name the output file.
- If you omit the `-c` option, you will create a shared library (`.so file`) directly from the `pc` command line in a single step. You must specify certain additional options required for shared libraries, including `-o` to ensure the output file has a `.so` file name suffix, and possibly other options associated witih shared library creation.

- If you specify the -c option, you will create an object module (.o file) that you can name using the -o option. To create a shared library, process the .o file using ld, specifying certain options associated with shared library creation.

You can specify multiple source and object files when creating a shared library with the pc command.

2.4.1 Creating a Shared Library Using the pc Command

You can create a shared libary (.so) file using a single pc command. For example:

```
% pc -shared -o octagon.so octagon.p
```

A description of each pc option follows:

- The -shared option creates a shared library. It generates postiion-independent code and recognizes certain options that are specified to ld if you omit the -c option.
- The -o option names the shared library octagon.so (instead of a.out).
- The name of the source module is octagon.p. You can specify multiple source files.

2.4.2 Creating a Shared Library Using pc and ld Command

You first must create a .o file that contains position-independent code, such as octagon.o in the following example:

```
% pc -shared -O3 -c -o octagon.o octagon.p
```

The file octagon.o is then used as input to the ld command to create the shared library, named octagon.so, with a package named subs:

```
% ld -shared -no_archive -lFutil -o octagon.so \
             octagon.o -lpas -lots -lm -lc
```

A description of each ld option follows:

- The -shared option creates a shared library.
- The -no_archive option indicates that ld should not search archive libraries to resolve external names (only shared libraries).
- The -o option names the shared library octagon.so (instead of a.out).
- The name of the object module is octagon.o. You can specify multiple .o files.

- The `-lpas` and subsequent options are the standard list of libraries that the `pc` command would have otherwise passed to `ld`. When creating shared libraries, note that all symbols must be resolved when you use the `ld` command to create that shared library. For more information about the standard list of libraries used by *Compaq Pascal*, see Table 2–8.

Certain other `ld` command options may be useful. For example, to optimize shared library startup, use the `-update_registry` and `-check_registry` options, which preassign a starting address in virtual memory to a shared library using the file `/usr/shlib/so_locations`. For additional information on the relevant `ld` options, see your operating system documentation or the `ld(1)` reference page.

2.4.3 Choosing How to Create a Shared Library

Consider the following when deciding whether to create a shared library with a single `pc` command (`-c` ommited) or with both the `pc` (`-c` present) and `ld` commands:

- Certain `ld` options may not be available from the `pc` command line. If you need to use those options, use the two-command method (specify `pc -c` and subsequently use `ld`). Such options include `-check_registry` and `-update_registry`, which preassign a starting address in virtual memory to a shared library using the file `/usr/shlib/so_locations`.
- If you use a single `pc` command with `-shared` and omit `-c`, you do not need to specify the standard list of `pc` libraries using the `-l`*string* option.

For additional information on `ld` options, see your operating system documentation of the `ld(1)` reference page.

For More Information:

- On the standard list of libraries used by *Compaq Pascal* (Table 2–8)

2.5 Temporary Files

Temporary files created by the compiler or a preprocessor reside in the current directory and are usually deleted, unless the `-K` option was specified. You can set the environment variable TMPDIR to specify a directory to contain temporary files if your local directory is not acceptable. For performance reasons, use a local disk (rather than using a NFS mounted disk) to contain the temporary files.

To view the file name and directory where each temporary file is created, use the `-v` option.

If your program creates scratch files during program execution, you can set the TMPDIR environment variable to specify which directory will contain the scratch (temporary) files.

2.6 Using Multiple Input Files: Effect on Output Files

When you specify multiple source files, the following options control the production of output files:

- The `-c` option prevents linking. This preserves the temporary object files, which the linker usually deletes.
- The `-K` option keeps temporary files.

A description of the interaction of these options follows:

- If you omit both the `-c` and the `-K` options (default), the specified Pascal file are each compiled into separate object files and then linked together. The object files are then deleted.
- If you specify the `-c` option or if you specify the `-K` option, each source file is compiled into an object file, creating one object file for each input source file specified.
- If you specify the `-c` option, you must link the object file(s) later by using a separate `pc` command, perhaps by means of a makefile processed by the `make` command, such as for incremental compilation of a large application.

2.7 Interactions of File Name Suffix and Options

You can select from a variety of processing options and specify files other than *Compaq Pascal* source files. The combination of the processing options and the suffix of each file determines how the `pc` command handles the processing.

For example, you may want to call a utility routine written in C from a *Compaq Pascal* program. The *Compaq Pascal* program is contained in a file named `myprog.pas`. The C routine named utilityx is contained in a file named `utilityx.c`. Consider the following `pc` command:

```
% pc -o myprog myprog.pas utilityx.c
```

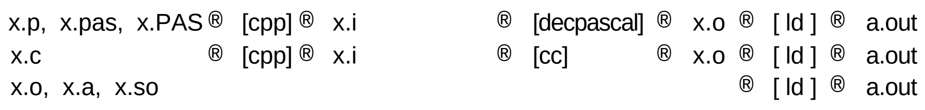
This command does the following:

1. Compiles `myprog.pas` with the *Compaq Pascal* compiler
2. Compiles `utilityx.c` with the C compiler
3. Links both object files together into the executable program named `myprog`

To insert the compiled programs into an object library instead of linking them directly into an executable program, use the `-c` option to keep the object (`.o`) file or files.

Figure 2–1 shows how the `pc` command processes the various types of file suffixes that it recognizes (assuming that a specified option does not negate some part of the processing).

Figure 2–1 pc Command Processing by File Suffix


x.p, x.pas, x.PAS ® [cpp] ® x.i ® [decpascal] ® x.o ® [ld] ® a.out
x.c ® [cpp] ® x.i ® [cc] ® x.o ® [ld] ® a.out
x.o, x.a, x.so ® [ld] ® a.out

ZK−7918A−GE

2.8 Using Listing Files

If you expect your program to have compilation errors, you should request a separate listing file (`-V` option).

For example, the following command compiles a *Compaq Pascal* source files and `ld` creates an executable file named `a.out`:

```
% pc -V aaa.p
```

The listing file assumes the name of the file. If the file was named `aaa.p`, the listing file is named `aaa.l`.

Using a listing file provides such information as the column pointer (1) that indicates the exact part of the line that caused the error. Especially for large files, consider obtaining a printed copy of the listing file you can reference while editing the source file.

3 Separate Compilation

Pascal allows you to divide your application into subprograms by creating procedures and functions. *Compaq Pascal* allows you further modularity by allowing you to create compilation units, called programs and modules, that can be compiled separately. This chapter discusses the following topics about separate compilation:

- Sharing data with the ENVIRONMENT and INHERIT attributes (Section 3.1)
- Isolating data with the HIDDEN attribute (Section 3.1)
- Interfaces and implementations (Section 3.2)
- Data models (Section 3.3)
- Examples (Section 3.4)

For More Information:

- On the ENVIRONMENT, HIDDEN, and INHERIT attributes (*Compaq Pascal Language Reference Manual*)
- On compiling and executing programs and modules (Chapter 2)

3.1 The ENVIRONMENT, HIDDEN, and INHERIT Attributes

To divide your program into a program and a series of modules, you need to decide, according to the needs of your application, which data types, constants, variables, and routines need to be shared either by other modules or by the program.

To share data, create an environment file by using the ENVIRONMENT attribute in a module. Consider the following example:

```
{
Source File: share_data.pas
This program initializes data to be shared with another compilation
unit.
}
[ENVIRONMENT( 'share_data' )]
Module Share_Data;
CONST
   Rate_For_Q1 = 0.1211;
   Rate_For_Q2 = 0.1156;
   Rate_For_Q3 = 0.1097;
   Rate_For_Q4 = 0.11243;
TYPE
   Initialized_Type = ARRAY[1..10] OF INTEGER VALUE
                      [1..5: 67; 6,9: 105; OTHERWISE 33];
END.
```

If you do not specify a file name with the [ENVIRONMENT] attribute, the file name of the source file is used with a ".pen" extension for the name of the environment file. For example,

```
  {
     Module share_data.pas
  }
  [ENVIRONMENT]
  Module Share_Data;
    CONST
      Rate_For_Q1 = 0.1211;
      Rate_For_Q2 = 0.1156;
  END.
```

The preceding module, when compiled, creates an environment file named "share_data.pen".

If a file name is specified with the [ENVIRONMENT] attribute, that file name is used (unchanged) in creating the environment file.

The [INHERIT] attribute causes the compiler to attempt to open a file with the exact file name that is specified. If this fails, an extension of .pen will be added to the file name and the compiler will try to open the file again. For example,

```
{
  Program inherit_example.pas
}
[INHERIT ('share_data')]

Program inherit_example(output);
  CONST
    My_Rate = Rate_For_Q1*2.0;
  BEGIN
    Writeln(My_Rate)
  END.
```

When the preceding program is compiled, the compiler attempts to open the file share_data as an environment file. If share_data is not found, the compiler attempts to open share_data.pen as an environment file. If share_data.pen is not found, an error message is issued and the compilation is stopped.

To build and run the application made up of the code in the previous examples, use the following commands:

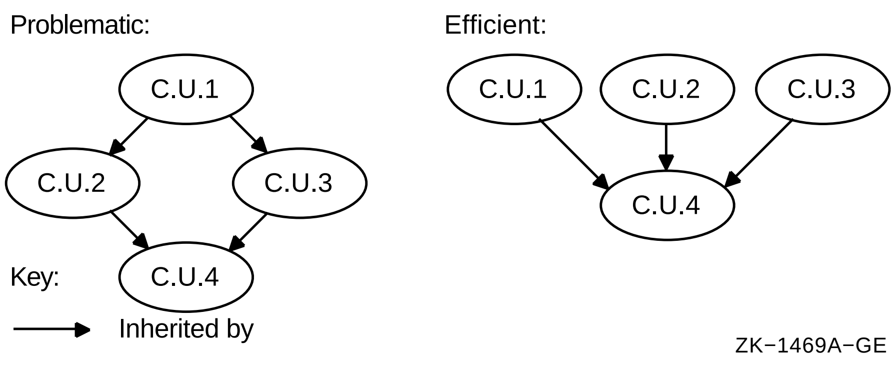
```
% pc -c share_data.pas
% pc -c program.pas
% pc share_data.o program.o -o program
% program
  33
```

If a module contains variable declarations, routine declarations, schema types, or module initialization or finalization sections, you must link the program with the module that created the environment file to resolve external references. To prevent errors, you may wish to link programs with modules of inherited environment files as standard programming practice. For example, if SHARE_DATA contained a variable declaration, you must enter the following to resolve the external reference:

```
% pc share_data.pas program.pas -o program
% program
    33
```

For many applications, it is a good idea to place all globally accessible data into one module, create a single environment file, and inherit that module in other compilation units that need to make use of that data. Using environment files in this way reduces the difficulties in maintaining the data (it is easier to maintain one file) and it eliminates problems that can occur when you **cascade** environment files. If compilation unit A inherits an environment file from compilation unit B, and if unit B inherits a file from unit C, then inheritance is cascading. Figure 3–1 shows a cascading inheritance path and a noncascading inheritance path.

Figure 3–1 Cascading Inheritance of Environment Files



Cascading is not always undesirable; it depends on your application and on the nature of the environment files. For example, if cascading occurs for a series of constant and type definitions that are not likely to change, cascading may require very little recompiling and relinking. However, if the constant and type definitions change often or if environment files contain routines and variables, you may find it easier to redesign the inheritance paths of environment files due to the recompiling and relinking involved.

Also, the inheritance path labeled **Efficient** in Figure 3–1 is not immune to misuse. That inheritance path, although it avoids the problems of cascading, may still involve multiply declared identifiers (identical identifiers contained in several of the compilation units whose environment files are inherited by compilation unit 4).

In many instances, *Compaq Pascal* does not allow multiply declared identifiers in one application. For example, a compilation unit cannot inherit two environment files that declare the same identifier; also, a compilation unit usually cannot inherit an environment file that contains an identifier that is identical to an identifier in the outermost level of the unit (one exception, for example, is the redeclaration of a redefinable reserved word or of an identifier predeclared by *Compaq Pascal*). Also, *Compaq Pascal* allows the following exceptions to the rules concerning multiply declared identifiers:

- A variable identifier can be multiply declared if all declarations of the variable have the same type and attributes, and if all but one declaration at most are external.

- A procedure identifier can be multiply declared if all declarations of the procedure have congruent parameter lists and if all but one declaration at most are external.
- A function identifier can be multiply declared if all declarations of the function have congruent parameter lists and identical result types, and if all but one declaration at most are external.

If a compilation unit creates an environment file and if it contains data that you do not want to share with other compilation units, you can use the HIDDEN attribute. Consider the following example:

```
[ENVIRONMENT]
MODULE Example;
TYPE
   Array_Template( Upper : INTEGER ) =
      [HIDDEN] ARRAY[1..Upper] OF INTEGER;
   Global_Type : Array_Template( 10 );
VAR
   i : [HIDDEN] INTEGER;   {Used for local incrementing}

PROCEDURE x;
   BEGIN
      i := i + 1;
   END;

PROCEDURE y;
   BEGIN
      FOR i := i + 1;
   END;
END.
```

The code in the previous example hides the schema type, preventing the schema type from being used in inheriting modules. (Whether to hide the type depends on the requirements of a given application.) Also, *Compaq Pascal* does not include the variable *i* in the environment file; this allows inheriting modules to declare the identifier variable *i* as an incrementing variable without being concerned about generating errors for a multiply defined identifier.

3.1.1 Environment File Dependency Checking

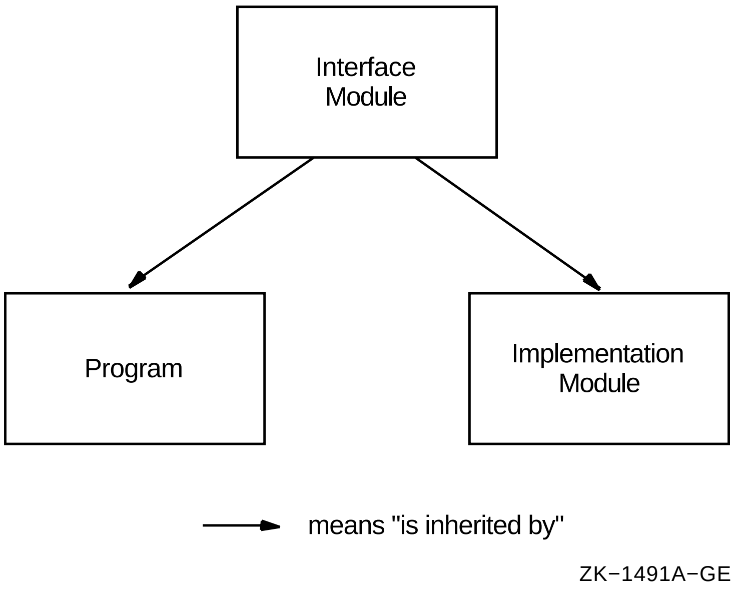
Compaq Pascal performs compile-time checks to ensure that all compilations that inherit environment files actually used the same environment file definition.

3.2 Interfaces and Implementations

If your application requires, you can use a method of creating and inheriting environment files that minimizes the number of times you have to recompile compilation units. This method involves the division of module declarations into two separate modules: an interface module and an implementation module. The **interface module** contains data that is not likely to change: constant definitions, variable declarations, and external routine declarations. The **implementation module** contains data that may change: bodies of the routines declared in the interface module, and private types, variables, routines, and so forth.

The interface module creates the environment file that is inherited by both the implementation module and by the program. Figure 3–2 shows the inheritance process.

Figure 3–2 Inheritance Path of an Interface, an Implementation, and a Program



Consider this code fragment from the interface module in Example 3–1 (see Section 3.4):

```
[ENVIRONMENT( 'interface' )]
MODULE Graphics_Interface( OUTPUT );

   {Globally accessible type}

{Provide routines that manipulate the shapes:}
PROCEDURE Draw( s : Shape ); EXTERNAL;
PROCEDURE Rotate( s : Shape ); EXTERNAL;
PROCEDURE Scale( s : Shape ); EXTERNAL;
PROCEDURE Delete( s : Shape ); EXTERNAL;

   {Module initialization section}

END.
```

The code contained in the interface is not likely to change often. The implementation code can change without requiring recompilation of the other modules in the application. Consider this code fragment from the implementation module in Example 3–2 (see Section 3.4):

```
[INHERIT( 'interface' )]   {Predeclared graphics types and routines}
MODULE Graphics_Implementation( OUTPUT );

[GLOBAL] PROCEDURE Rotate( s : Shape );
   BEGIN
   WRITELN( 'Rotating the shape :', s.t );
   END;
```

To compile, link, and run the code in Examples 3–1, 3–2, and 3–3 (the main program), use the following commands:

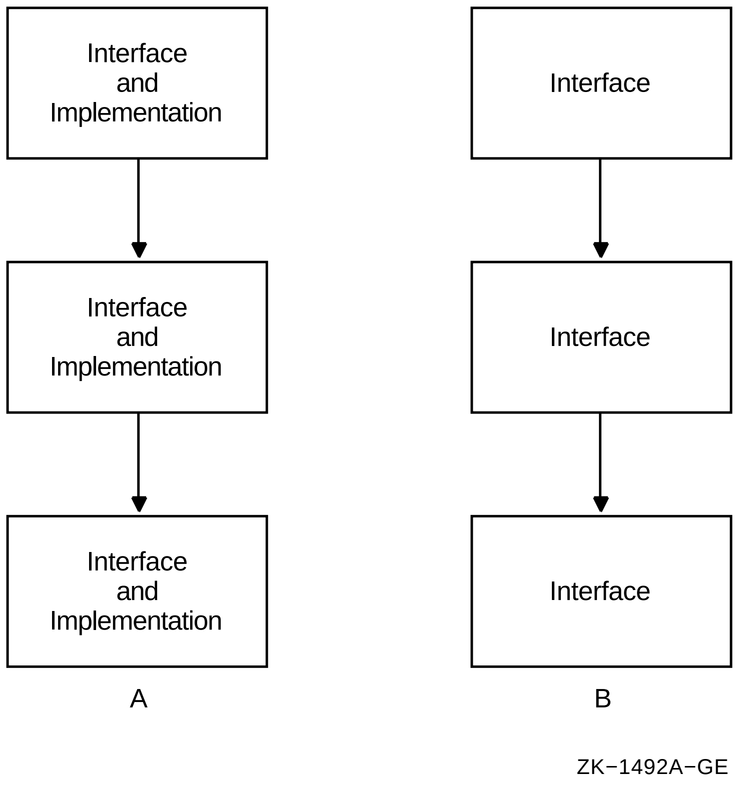
```
% pc -c graphics_interface.pas
% pc -c graphics_implementation.pas
% pc -c graphics_main_program.pas
% pc graphics_interface.o graphics_implementation.o \
graphics_main_program.o -o graphics_main_program
% graphics_main_program
```

If you need to change the code contained in any of the routine bodies declared in the implementation module, you do not have to recompile the program to reflect the changes. For example, if you have to edit the implementation module, you can regenerate the application with the following commands:

```
% vi graphics_implementation.pas
% pc -c graphics_implementation.pas
% pc graphics_interface.o graphics_implementation.o \
graphics_main_program.o -o graphics_main_program
% graphics_main_program
```

In this manner, interfaces and implementations can save you maintenance time and effort. In addition, the interface and implementation design allows you to better predict when cascading inheritance may provide maintenance problems. Figure 3–3 shows two forms of cascading.

Figure 3–3 Cascading Using the Interface and Implementation Design


Interface and Implementation
Interface and Implementation
Interface and Implementation
A
Interface
Interface
Interface
B
ZK−1492A−GE

If the compilation units creating environment files are designed to contain both interface and implementation declarations, the cascading in column A may lead to more recompiling, more relinking, and more multiply declared identifiers. The design shown in column B does not always provide easy maintenance, but it is more likely to do so. For example, if each interface provided a different kind of constant or type (as determined by your application) and if the constants and types are not derived from one another, the inheritance path in column B may be quite efficient and orderly, and may require little recompiling and relinking.

Do not place the following in an implementation module:

- Nonstatic types and variables at the module level
- A module initialization section (TO BEGIN DO)
- A module finalization section (TO END DO)

These restrictions are necessary because *Compaq Pascal* cannot determine the order of activation of initialization and finalization sections that do not directly follow an environment-file inheritance path. Since implementation modules do not create environment files, the initialization and finalization sections in those modules are effectively outside of any inheritance path. Also, if you use the previously listed objects in implementation modules, there may be attempts to access data that has not yet been declared. Consider the following example:

```
{In one file:}
[ENVIRONMENT( 'interface' )]
MODULE Interface;
PROCEDURE x; EXTERNAL;
END.

{In another file:}
[INHERIT( 'interface' )]
MODULE Implementation( OUTPUT );
VAR
   My_String : STRING( 10 );

[GLOBAL] PROCEDURE x;
   BEGIN
   WRITELN( My_String );
   END;

TO BEGIN DO
   My_String := 'Okay';
END.
```

In the previous example, it is possible for you to call procedure *x* (in some other module that also inherits `interface`) before the creation and initialization of the variable `My_String`. You can circumvent this problem by using a routine call to initialize the variable and by moving the code to the interface module, as shown in the next example:

```
{In one file:}
[ENVIRONMENT( 'interface' )]
MODULE Interface;
VAR
   My_String : STRING( 10 );

PROCEDURE x; EXTERNAL;
PROCEDURE Initialize; EXTERNAL;
```

```
TO BEGIN DO
   Initialize;
END.

{In another file:}
[INHERIT( 'interface' )]
MODULE Implementation( OUTPUT );


[GLOBAL] PROCEDURE x;
   BEGIN
   WRITELN( My_String );
   END;

[GLOBAL] PROCEDURE Initialize;
   BEGIN
   My_String := 'Okay';
   END;
END.
```

3.3 Data Models

Using separate compilation and a few other features of *Compaq Pascal* (including initial states, constructors, the HIDDEN attribute, and TO BEGIN DO and TO END DO sections), you can construct models for creating, distributing, isolating, and restricting data in an application.

Of course, the design of the data model depends on the needs of a particular application. However, to show some of the power of *Compaq Pascal* features used in conjunction, Examples 3–1, 3–2, and 3–3 in Section 3.4 create a generic graphics application. Consider the following code fragment from Example 3–1:

```
TYPE
   Shape_Types = ( Rectangle, Circle ); {Types of graphics objects}

   Shape( t : Shape_Types ) = RECORD
                                    {Starting coordinate points}
      Coordinate_X, Coordinate_Y : REAL VALUE 50.0;
      CASE t OF                           {Shape-specific values}
         Rectangle : ( Height, Width : REAL VALUE 10.0 );
         Circle    : ( Radius        : REAL VALUE 5.0 );
      END;

{Provide routines that manipulate the shapes:}
PROCEDURE Draw( s : Shape ); EXTERNAL;
PROCEDURE Rotate( s : Shape ); EXTERNAL;
PROCEDURE Scale( s : Shape ); EXTERNAL;
PROCEDURE Delete( s : Shape ); EXTERNAL;
```

The interface module provides an interface to the rest of the application. This module contains types and external procedure declarations that the data model chooses to make available to other compilation units in the application; other units can access these types and routines by inheriting the generated environment file.

The type Shape_Types defines two legal graphical objects for this application: a circle and a rectangle. The type Shape can be used by other units to create circles and rectangles of specified dimensions. This code uses a variant record to specify the different kinds of data needed for a circle (a radius value) and a rectangle (height and width values).

Since the type has initial-state values, any variable declared to be of this type receives these values upon declaration. Providing initial states for types that are included in environment files can prevent errors when other compilation units try to access uninitialized data.

The initial states in this code are specified for the individual record values. You can also provide an initial state for this type using a constructor, as follows:

```
   Shape( t : Shape_Types ) = RECORD
      Coordinate_X, Coordinate_Y : REAL;
      CASE t OF
         Square : ( Height, Width : REAL );
         Circle : ( Radius        : REAL );
      END VALUE [ Coordinate_X : 50.0; Coordinate_Y : 50.0;
                  CASE Circle OF [ Radius : 5.0 ] ];
```

If you use constructors for variant records, you can only specify an initial state for one of the variant values. If you need to specify initial states for all variant values, you must specify the initial states on the individual variants, as shown in Example 3–1.

The interface module also declares routines that can draw, rotate, scale, and delete an object of type Shape. The bodies of these routines are located in the implementation module. The interface module also contains a TO BEGIN DO section, as shown in the following code fragment:

```
[HIDDEN] PROCEDURE Draw_Logo; EXTERNAL;

{
Before program execution, display a logo to which the main
program has no access.
}
TO BEGIN DO
   Draw_Logo;
```

As with the other routines, the body of Draw_Logo is located in the implementation module. The HIDDEN attribute prevents compilation units that inherit the interface environment file from calling the Draw_Logo routine. This ensures that the application only calls Draw_Logo once at the beginning of the application.

Using this design, the interface module can provide graphical data and tools to be used by other compilation units without the other units having to worry about implementation details. The actual details are contained in one implementation module. For example, the routine bodies are contained in the implementation module. Consider the following code fragment from Example 3–2:

```
{Declare routine bodies for declarations in the interface}
[GLOBAL] PROCEDURE Draw( s : Shape );
   BEGIN
   CASE s.t OF
      Circle    : WRITELN( 'Code that draws a circle' );
      Rectangle : WRITELN( 'Code that draws a rectangle' );
      END;
   END; {Procedure Draw}
```

The routine bodies of the external routines declared in the interface module are located in the implementation module. The code in each of the routines uses the actual discriminant of parameter *s* to determine if the shape is a circle or a rectangle and draws the shape. If this code needs to change, it does not require that you recompile the code in Examples 3–1 or 3–3 in Section 3.4.

Example 3–2 also contains code that is isolated and hidden from other compilation units that inherit the interface environment file. Consider the following code fragment from the interface module:

```
[GLOBAL] PROCEDURE Draw_Logo;
   VAR
      Initial_Shape  : Shape( Circle ) {Declare object}
         VALUE [ Coordinate_X : 50.0;
                 Coordinate_Y : 50.0;
                 CASE Circle OF
                 [Radius      : 15.75;]];
   BEGIN
   WRITELN( 'Drawing a company logo' );
   Draw( Initial_Shape );
   {Code pauses for 30 seconds as the user looks at the logo.}
   Delete( Initial_Shape );
   WRITELN;
   {Ready for the rest of the graphics program to begin.}
   END;
```

In the graphical data model, you may wish to define a company logo, and you may wish to display that logo on the screen before any other graphical data is drawn or displayed. This code declares the variable Initial_Shape. Since this variable is declared locally to Draw_Logo and since Draw_Logo is contained in a module that does not produce an environment file, other modules that may have access to the interface environment file do not have access to this variable. In this application, you may not wish to give other compilation units the power to alter the company logo.

The code in the interface's TO BEGIN DO section, which executes before any program code, displays the company logo and deletes it to begin the application. Consider again the compilation process for interfaces, implementations, and programs:

```
% pc -c graphics_interface.pas
% pc -c graphics_implementation.pas
% pc -c graphics_main_program.pas
% pc graphics_interface.o graphics_implementation.o \
graphics_main_program.o -o graphics_main program
% graphics_main_program
```

Compaq Pascal executes the TO BEGIN DO section according to the inheritance order of environment files. Remember that *Compaq Pascal* cannot determine the order of execution for TO BEGIN DO sections contained in implementation modules, so do not use them there.

Using this design, you can allow different sites that run the graphics application to access global data through the interface module. One location can maintain and control the contents of the implementation module, shipping the implementation's object module for use at other sites. You can use this method for other types of sensitive data or data that needs to be maintained locally.

3.4 Separate Compilation Examples

Example 3–1 shows an interface module that creates the environment file interface. This environment file is inherited in Examples 3–2 and in 3–3.

Example 3–1 An Interface Module for Graphics Objects and Routines

```
{
Source File: graphics_interface.pas
This module creates an interface to graphical data and routines.
}
[ENVIRONMENT( 'interface' )]
MODULE Graphics_Interface;
TYPE
   Shape_Types = ( Rectangle, Circle ); {Types of graphics objects}

   Shape( t : Shape_Types ) = RECORD
                                    {Starting coordinate points:}
      Coordinate_X, Coordinate_Y : REAL VALUE 50.0;
      CASE t OF                           {Shape-specific values}
         Rectangle : ( Height, Width : REAL VALUE 10.0 );
         Circle    : ( Radius        : REAL VALUE 5.0 );
      END;

{Provide routines that manipulate the shapes:}
PROCEDURE Draw( s : Shape ); EXTERNAL;
PROCEDURE Rotate( s : Shape ); EXTERNAL;
PROCEDURE Scale( s : Shape ); EXTERNAL;
PROCEDURE Delete( s : Shape ); EXTERNAL;
[HIDDEN] PROCEDURE Draw_Logo; EXTERNAL;

{
Before program execution, display a logo to which the main
program has no access.
}
TO BEGIN DO
   Draw_Logo;
END.
```

Example 3–2 shows the implementation of the routines declared in Example 3–1.

Example 3–2 An Implementation Module for Graphics Objects and Routines

```
{
Source File: graphics_implementation.pas
This module implements the graphics routines and data declarations
made global by the interface module.
}
[INHERIT( 'interface' )]   {Predeclared graphics types and routines}
MODULE Graphics_Implementation( OUTPUT );
```

(continued on next page)

Example 3–2 (Cont.) An Implementation Module for Graphics Objects and Routines

```
{Declare routine bodies for declarations in the interface:}
[GLOBAL] PROCEDURE Draw( s : Shape );
   BEGIN
   CASE s.t OF
      Circle    : WRITELN( 'Code that draws a circle' );
      Rectangle : WRITELN( 'Code that draws a rectangle' );
      END;
   END; {Procedure Draw}

[GLOBAL] PROCEDURE Rotate( s : Shape );
   BEGIN
   WRITELN( 'Rotating the shape :', s.t );
   END;

[GLOBAL] PROCEDURE Scale( s : Shape );
   BEGIN
   WRITELN( 'Scaling the shape :', s.t );
   END;

[GLOBAL] PROCEDURE Delete( s : Shape );
   BEGIN
   WRITELN( 'Deleting the shape :', s.t );
   END;

[GLOBAL] PROCEDURE Draw_Logo;
   VAR
      Initial_Shape  : Shape( Circle ) {Declare object}
         VALUE [ Coordinate_X : 50.0;
                 Coordinate_Y : 50.0;
                 CASE Circle OF
                 [Radius       : 15.75;]];
   BEGIN
   WRITELN( 'Drawing a company logo' );
   Draw( Initial_Shape );
   {Code pauses for 30 seconds as the user looks at the logo.}
   Delete( Initial_Shape );
   WRITELN;
   {Ready for the rest of the graphics program to begin.}
   END;
END.
```

Example 3–3 shows a main program and its use of the types and routines provided by the interface module.

Example 3–3 A Graphics Main Program

```
{
Source File: graphics_main_program.pas
This program inherits the interface environment file, which gives it
access to the implementation's declarations.
}
[INHERIT( 'interface' )]  {Types and routines in interface module}
PROGRAM Graphics_Main_Program( OUTPUT );

VAR
   My_Shape : Shape( Rectangle )
      VALUE [ Coordinate_X : 25.0;
              Coordinate_Y : 25.0;
              CASE Rectangle OF
              [Height : 12.50; Width : 25.63]];
BEGIN
{
You cannot access the variable Initial_Shape, because it is in the
implementation module, and that module does not create an environment
file.

You can work with My_Shape.  If you did not provide initial values in
this declaration section, the module Graphics_Interface provided
initial values for the schema type Shape.
}
Draw(   My_Shape );
Scale(  My_Shape );
Rotate( My_Shape );
Delete( My_Shape );
END.
```

To compile, link, and run the code in Examples 3–1, 3–2, and 3–3, enter the following:

```
% pc -c graphics_interface.pas
% pc -c graphics_implementation.pas
% pc -c graphics_main_program.pas
% pc graphics_interface.o graphics_implementation.o \
graphics_main_program.o -o graphics_main_program
Drawing a company logo
Code that draws a circle
Deleting the shape :    CIRCLE

Code that draws a rectangle
Scaling the shape : RECTANGLE
Rotating the shape : RECTANGLE
Deleting the shape : RECTANGLE
```

4
Optimizing Performance

The objective of **optimization** is to produce source and object programs that achieve the greatest amount of processing with the least amount of time and memory. Realizing this objective requires programs that are carefully designed and written, and compilation techniques, such as those used by *Compaq Pascal*, that take advantage of the operating system and machine architecture environment. (The benefits of portable code and program efficiency depend on the requirements of your application.)

This chapter discusses the following topics:

- Compiler optimizations (Section 4.1)
- Programming considerations (Section 4.2)
- Optimization considerations (Section 4.3)
- Profiling a program (Section 4.5)
- Controlling the size of global pointer data (Section 4.6)

4.1 Compiler Optimizations

By default, programs compiled with the *Compaq Pascal* compiler undergo optimization. An optimizing compiler automatically attempts to remove repetitious instructions and redundant computations by making assumptions about the values of certain variables. This, in turn, reduces the size of the object code, allowing a program written in a high-level language to execute at a speed comparable to that of a well-written assembly language program. Optimization can increase the amount of time required to compile a program, but the result is a program that may execute faster and more efficiently than a nonoptimized program.

The language elements you use in the source program directly affect the compiler's ability to optimize the object program. Therefore, you should be aware of the ways in which you can assist compiler optimization.

In addition, this awareness often makes it easier for you to track down the source of a problem when your program exhibits unexpected behavior.

The compiler performs the following optimizations:

- Compile-time evaluation of constant expressions
- Elimination of some common subexpressions
- Partial elimination of unreachable code
- Code hoisting from structured statements, including the removal of invariant computations from loops
- Inline code expansion for many predeclared functions
- Inline code expansion for user-declared routines
- Rearrangement of unary minus and NOT operations to eliminate unary negation and complement operations
- Partial evaluation of logical expressions
- Propagation of compile-time known values
- Partial unrolling of FOR, WHILE, and REPEAT loops
- Strength reduction
- Split lifetime analysis
- Code scheduling
- Loop unrolling
- Software pipelining

These optimizations are described in the following sections. In addition, the compiler performs the following optimizations, which can be detected only by a careful examination of the machine code produced by the compiler:

- Global assignment of variables to registers

  If possible, reduce the number of memory references needed by assigning frequently referenced variables to registers.
- Reordering the evaluation of expressions

  This minimizes the number of temporary values required.
- Peephole optimization of instruction sequences

  The compiler examines code a few instructions at a time to find operations that can be replaced by shorter and faster equivalent operations.

For More Information:

- On *Compaq Pascal* language elements (*Compaq Pascal Language Reference Manual*)

4.1.1 Compile-Time Evaluation of Constants

The compiler performs the following computations on constant expressions at compile time:

- Negation of constants

  The value of a constant preceded by unary minus signs is negated at compile time. For example:

  ```
  x := -10.0;
  ```

- Type conversion of constants

  The value of a lower-ranked constant is converted to its equivalent in the data type of the higher-ranked operand at compile time. For example:

  ```
  x := 10 * y;
  ```

  If x and y are both real numbers, then this operation is compiled as follows:

  ```
  x := 10.0 * y;
  ```

- Arithmetic on integer and real constants

  An expression that involves +, –, *, or / operators is evaluated at compile time. For example:

  ```
  CONST
     nn = 27;
  {In the executable section:}
  i := 2 * nn + j;
  ```

  This is compiled as follows:

  ```
  i := 54 + j;
  ```

- Array address calculations involving constant indexes

  These are simplified at compile time whenever possible. For example:

  ```
  VAR
     i : ARRAY[1..10, 1..10] OF INTEGER;
  {In the executable section:}
  i[1,2] := i[4,5];
  ```

- Evaluation of constant functions and operators

Arithmetic, ordinal, transfer, unsigned, allocation size, CARD, EXPO, and ODD functions involving constants, concatenation of string constants, and logical and relational operations on constants, are evaluated at compile time.

For More Information:

- On the complete list of compile-time operations and routines (*Compaq Pascal Language Reference Manual*)

4.1.2 Elimination of Common Subexpressions

The same subexpression often appears in more than one computation within a program. For example:

```
a := b * c + e * f;

h := a + g - b * c;

IF ((b * c) - h) <> 0 THEN ...
```

In this code sequence, the subexpression b * c appears three times. If the values of the operands b and c do not change between computations, the value b * c can be computed once and the result can be used in place of the subexpression. The previous sequence is compiled as follows:

```
t := b * c;

a := t + e * f;

h := a + g - t;

IF ((t) - h) <> 0 THEN ...
```

Two computations of b * c have been eliminated. In this case, you could have modified the source program itself for greater program optimization.

The following example shows a more significant application of this kind of compiler optimization, in which you could not reasonably modify the source code to achieve the same effect:

```
VAR
   a, b : ARRAY[1..25, 1..25] OF REAL;
{In the executable section:}
a[i,j] := b[i,j];
```

Without optimization, this source program would be compiled as follows:

```
t1 := (j - 1) * 25 + i;
t2 := (j - 1) * 25 + i;
a[t1] := b[t2];
```

Variables t1 and t2 represent equivalent expressions. The compiler eliminates this redundancy by producing the following optimization:

```
t = (j - 1) * 25 + i;
a[t] := b[t];
```

4.1.3 Elimination of Unreachable Code

The compiler can determine which lines of code, if any, are never executed and eliminates that code from the object module being produced. For example, consider the following lines from a program:

```
CONST
   Debug_Switch = FALSE;
{In the executable section:}
IF  Debug_Switch  THEN  WRITELN( 'Error found here' );
```

The IF statement is designed to write an error message if the value of the symbolic constant `Debug_Switch` is TRUE. Suppose that the error has been removed, and you change the definition of `Debug_Switch` to give it the value FALSE. When the program is recompiled, the compiler can determine that the THEN clause will never be executed because the IF condition is always FALSE; no machine code is generated for this clause. You need not remove the IF statement from the source program.

Code that is otherwise unreachable, but contains one or more labels, is not eliminated unless the GOTO statement and the label itself are located in the same block.

4.1.4 Code Hoisting from Structured Statements

The compiler can improve the execution speed and size of programs by removing invariant computations from structured statements. For example:

```
FOR j := 1 TO i + 23 DO
   BEGIN
   IF Selector THEN a[i + 23, j - 14] := 0
   ELSE b[i + 23, j - 14] := 1;
   END;
```

If the compiler detected this IF statement, it would recognize that, regardless of the Boolean value of `Selector`, a value is stored in the array component denoted by [i + 23, j—14]. The compiler would change the sequence to the following:

```
t := i + 23;
FOR j := 1 TO t DO
   BEGIN
   u := j - 14;
   IF Selector THEN a[t,u] := 0
   ELSE b[t,u] := 1;
   END;
```

This removes the calculation of $j-14$ from the IF statement, and the calculation of $i+23$ from both the IF statement and the loop.

4.1.5 Inline Code Expansion for Predeclared Functions

The compiler can often replace calls to predeclared routines with the actual algorithm for performing the calculation. For example:

```
Square := SQR( a );
```

The compiler replaces this function call with the following, and generates machine code based on the expanded call:

```
Square := a * a;
```

The program executes faster because the algorithm for the SQR function has already been included in the machine code.

4.1.6 Inline Code Expansion for User-Declared Routines

Inline code expansion for user-declared routines performs in the same manner as inline code expansion for predeclared functions: the compiler can often replace calls to user-declared routines with an inline expansion of the routine's executable code. Inline code expansion is useful on routines that are called only a few times. The overhead of an actual procedure call is avoided, which increases program execution. The size of the program, however, may increase due to the expansion of the routine.

To determine whether or not it is desirable to inline expand a routine, compilers use a complex algorithm.

4.1.7 Partial Evaluation of Logical Expressions

The Pascal language does not specify the order in which the components of an expression must be evaluated. If the value of an expression can be determined by partial evaluation, then some subexpressions may not be evaluated at all. This situation occurs most frequently in the evaluation of logical expressions. For example:

```
WHILE ( i < 10 ) AND ( a[i] <> 0 ) DO
   BEGIN
   a[i] := a[i] + 1;
   i := i + 1;
   END;
```

In this WHILE statement, the order in which the two subexpressions `( i < 10 )` and `( a[i] <> 0 )` are evaluated is not specified; in fact, the compiler may evaluate them simultaneously. Regardless of which subexpression is evaluated first, if its value is FALSE the condition being tested in the WHILE statement is also FALSE. The other subexpression need not be evaluated at all. In this case, the body of the loop is never executed.

To force the compiler to evaluate expressions in left-to-right order with short circuiting, you can use the AND_THEN operator, as shown in the following example:

```
WHILE ( i < 10 ) AND_THEN ( a[i] <> 0 ) DO
   BEGIN
   a[i] := a[i] + 1;
   i := i + 1;
   END;
```

4.1.8 Value Propagation

The compiler keeps track of the values assigned to variables and traces the values to most of the places that they are used. If it is more efficient to use the value rather than a reference to the variable, the compiler makes this change. This optimization is called **value propagation**. Value propagation causes the object code to be smaller, and may also improve run-time speed.

Value propagation performs the following actions:

- It allows run-time operations to be replaced with compile-time operations. For example:

  ```
  Pi := 3.14;
  Pi_Over_2 := Pi/2;
  ```

  In a program that includes these assignments, the compiler recognizes the fact that the value of `Pi` did not change between the time `Pi` was assigned and used. The compiler will use the value `Pi` instead of a reference to `Pi` and perform the division at compile time. The compiler treats the assignments as if they were as follows:

  ```
  Pi := 3.14;
  Pi_Over_2 := 1.57;
  ```

  This process is repeated, allowing for further constant propagation to occur.

- It allows comparisons and branches to be avoided at run time. For example:

  ```
  x := 3;
  IF x <> 3 THEN  y := 30
  ELSE y := 20;
  ```

  In a program that includes these operations, the compiler recognizes that the value of x is 3 and the THEN statement cannot be reached. The compiler will generate code as if the statements were written as follows:

  ```
  x := 3;
  y := 20;
  ```

4.1.9 Error Reduction Through Optimization

An optimized program produces results and run-time diagnostic messages identical to those produced by an equivalent unoptimized program. An optimized program may produce fewer run-time diagnostics, however, and the diagnostics may occur at different statements in the source program. For example:

Unoptimized Code	Optimized Code
a := x/y;	t := x/y;
b := x/y;	a := t;
FOR i := 1 TO 10 DO	b := t;
c[i] := c[i] * x/y;	FOR i := 1 TO 10 DO c[i] := c[i] * t;

If the value of y is `0.0`, the unoptimized program would produce 12 divide-by-zero errors at run time; the optimized program produces only one. (The variable t is a temporary variable created by the compiler.) Eliminating redundant calculations and removing invariant calculations from loops can affect the detection of such arithmetic errors. You should keep this in mind when you include error-detection routines in your program.

4.1.10 Strength Reduction

Strength reduction speeds computations by replacing a multiply operation with a more efficient add instruction when your program is computing array addresses on each iteration of a loop.

4.1.11 Split Lifetime Analysis

Split lifetime analysis improves register usage by determining if the lifetime of a variable can be broken into multiple independent sections. If so, the variable may be stored in different registers for each section. The registers can then be reused for other purposes between sections. In this situation, there may be times when the value of the variable does not exist anywhere in the registers. For example:

```
v:= 3.0 *q;
   .
   .
   .
x:= SIN(y)*v:
   .
   .
   .
v:= PI*x:
   .
   .
   .
y:= COS(y)*v;
```

This example shows that the variable v has two disjoint usage sections. The value of v in the first section does not affect the value of v in the second section. The compiler may use different registers for each section.

4.1.12 Code Scheduling

Code scheduling is a technique for reordering machine instructions to maximize the amount of overlap of the multiple execution units inside the CPU. The exact scheduling algorithms depend on the implementation of the target architecture.

4.1.13 Loop Unrolling

Loop unrolling is a technique for increasing the amount of code between branch instructions and labels by replicating the body of a loop. Increasing the code optimizes instruction scheduling. The following code shows such a transformation:

Original Code

```
FOR i:= 1 to 12 DO
   a[i]:= b[i] + c[i]
```

Unrolled Loop Code

```
i:= 1
WHILE i < 12 DO
   BEGIN
   a[i]:= b[i] + c[i];
   a[i+1]:= b[i+1] + c[i+1];
   a[i+2]:= b[i+2] + c[i+2];
   a[i+3]:= b[i+3] + c[i+3];
   i:= i+4;
   END;
```

In this example, the loop body was replicated four times, allowing the instruction scheduler to overlap the fetching of array elements with the addition of other array elements.

By default, loop unrolling makes 4 copies of an unrolled loop. You can change the number of copies from 1 to 16. This is controlled by:

`-unroll`*`number`*

Numbers larger than 4 may improve performance at a cost of additional code size. However, larger numbers may decrease performance due to cache requirements, register conflicts, and other factors.

4.1.14 Software Pipelining

Software pipelining and additional software dependency analysis are enabled using the `-O5` command flag, which in certain cases improves run-time performance. Note that `-O5` is the default.

Compared to regular loop unrolling (enabled at optimization level 3 or above), software pipelining uses instruction scheduling to eliminate instruction stalls within loops, rearranging instructions between different unrolled loop iterations to improve performance.

For instance, if the software dependency analysis of data flow reveals that certain calculations can be done before or after the iteration of the unrolled loop, software pipelining reschedules the instructions ahead or behind that loop iteration at places where their execution can prevent instruction stalls and therefor, improve performance.

Loops chosen for software pipelining have the following characteristics:

- Always the innermost loops (those executed the most)
- Do not contain branches or procedure calls

By modifying the unrolled loop and inserting instructions (as needed) before and after the unrolled loop, software pipelining generally improves run-time performance. The exception is cases where the loops contain a large number of instructions with many existing overlapped operations. In this situation, software pipelining may not have enough registers available to effectively improve execution performance, and using optimization level 5 instead of optimization level 4 may not improve run-time performance.

To determine whether using optimization level 5 benefits your particular program, you should time program execution for the same program compiled at level 4 and 5. For programs that contain loops that exhaust available registers, longer execution times may result with optimization level 5.

In cases where performance does not improve, you should consider compiling using `-O5 -unroll 1` to (possibly) improve the effects of software pipelining.

4.2 Programming Considerations

The language elements that you use in a source program directly affect the compiler's ability to optimize the resulting object program. Therefore, you should be aware of the following ways in which you can assist compiler optimization and obtain a more efficient program:

- Define constant identifiers to represent values that do not change during your program. The use of constant identifiers generally makes a program easier to read, understand, and later modify. In addition, the resulting object code is more efficient because symbolic constants are evaluated only once, at compile time, while variables must be reevaluated whenever they are assigned new values.
- Whenever possible, use the structured control statements CASE, FOR, IF-THEN-ELSE, REPEAT, WHILE, and WITH rather than the GOTO statement. You can use the GOTO statement to exit from a loop, but careless use of it interferes with both optimization and the straightforward analysis of program flow.
- Enclose in parentheses any subexpression that occurs frequently in your program. The compiler checks whether any assignments have affected the subexpression's value since its last occurrence. If the value has not changed, the compiler recognizes that a subexpression enclosed in parentheses has already been evaluated and does not repeat the evaluation. For example:

  ```
  x := SIN( u + (b - c) );
  y := COS( v + (b - c) );
  ```

  The compiler evaluates the subexpression $(b - c)$ as a result of performing the SIN function. When it is encountered again, the compiler checks to see whether new values have been assigned to either b or c since they were last used. If their values have not changed, the compiler does not reevaluate $(b - c)$.
- When a variable is accessed by a program block other than the one in which it was declared, the variable should have static rather than automatic allocation. An automatically allocated variable has a varying location in memory; accessing it in another block is time-consuming and less efficient than accessing a static variable.
- Avoid using the same temporary variable many times in the course of a program. Instead, use a new variable every time your program needs a temporary variable. Because variables stored in registers are the easiest to access, your program is most efficient when as many variables as possible can be allocated in registers. If you use several different temporary

variables, the lifetime of each one is greatly reduced; thus, there is a greater chance that storage for them can be allocated in registers rather than at memory locations.

- When creating schema records (or records with nonstatic fields), place the fields with run-time size at the end of the record. The generated code has to compute the offset of all record fields after a field with run-time size, and this change minimizes the overhead.

For More Information:

- On *Compaq Pascal* language elements and on attributes (*Compaq Pascal Language Reference Manual*)
- On compilation switches (Chapter 2)

4.3 Optimization Considerations

Because the compiler must make certain assumptions in order to optimize a program, unexpected results may occur if you do not utilize the optimizations discussed in the following sections. If your program does not execute correctly because of undesired optimizations, you can use the NOOPTIMIZE attribute or the appropriate compilation switch to prevent optimizations from occurring.

For More Information:

- On attributes, static and automatic variables (*Compaq Pascal Language Reference Manual*)
- On compilation switches (Chapter 2)

4.3.1 Compiling for Optimal Performance

The following command line will result in producing the fastest code from the compiler.

```
% pc -nozero_heap -math_library fast -O5
```

You can use the performance optionger (-usage performance), to identify datatypes that could be modified for additional performance.

4.3.2 Subexpression Evaluation

The compiler can evaluate subexpressions in any order and may even choose not to evaluate some of them. Consider the following subexpressions that involve a function with side effects:

```
IF f( a ) AND f( b ) THEN ...
```

This IF statement contains two designators for function `f` with the same parameter `a`. If `f` has side effects, the compiler does not guarantee the order in which the side effects will be produced. In fact, if one call to `f` returns FALSE, the other call to `f` might never be executed, and the side effects that result from that call would never be produced. For example:

```
q := f( a ) + f( a );
```

The Pascal standard allows a compiler to optimize the code as follows:

```
Q := 2 * f( a )
```

If the compiler does so, and function `f` has side effects, the side effects would occur only once because the compiler has generated code that evaluates `f( a )` only once.

If you wish to ensure left-to-right evaluation with short circuiting, use the AND_THEN and OR_ELSE Boolean operators.

For More Information:

- On the order of expression evaluation, see the description of the NOOPTIMIZE attribute (*Compaq Pascal Language Reference Manual*)

4.3.3 MAXINT and MAXINT64 Predeclared Constants

The smallest possible value of the INTEGER type is represented by the predeclared constant −MAXINT. However, the operating system architecture supports an additional integer value, which is (−MAXINT −1). If your program contains a subexpression with this value, the program's evaluation might result in an integer overflow trap. Therefore, a computation involving the value (−MAXINT −1) might not produce the expected result. To evaluate expressions that include (−MAXINT −1), you should disable either optimization or integer overflow checking.

The smallest possible value of the INTEGER64 type is presented by the expression -MAXINT64. However, the operating system architecture supports an additional integer value, -MAXINT64-1. To evaluate expressions that include the -MAXINT64-1 integer value, you should disable either optimization or integer overflow checking.

4.3.4 Pointer References

The compiler assumes that the value of a pointer variable is either the constant identifier NIL or a reference to a variable allocated in heap storage by the NEW procedure. A variable allocated in heap storage is not declared in a VAR section and has no identifier of its own; you can refer to it only by the name of a pointer variable followed by a circumflex (^). Consider the following example:

```
VAR
   x : INTEGER;
   p : ^INTEGER;
{In the executable section:}
NEW( p );
p^ := 0;
x  := 0;
IF p^ = x THEN  p^ := p^ + 1;
```

If a pointer variable in your program must refer to a variable with an explicit name, that variable must be declared VOLATILE or READONLY. The compiler makes no assumptions about the value of volatile variables and therefore performs no optimizations on them.

Use of the ADDRESS function, which creates a pointer to a variable, can result in a warning message because of optimization characteristics. By passing a nonread-only or nonvolatile static or automatic variable as the parameter to the ADDRESS function, you indicate to the compiler that the variable was not allocated by NEW but was declared with its own identifier. Because the compiler's assumptions are incorrect, a warning message occurs. You can also use IADDRESS, which functions similarly to the ADDRESS function except that IADDRESS returns an INTEGER_ADDRESS value and does not generate any warning messages. Use caution when using IADDRESS.

Similarly, when the parameter to ADDRESS is a formal VAR parameter or a component of a formal VAR parameter, the compiler issues a warning message that not all dynamic variables allocated by NEW may be passed to the function.

For More Information:

- On attributes and on predeclared routines (*Compaq Pascal Language Reference Manual*)

4.3.5 Variant Records

Because all the variants of a record variable are stored in the same memory location, a program can use several different field identifiers to refer to the same storage space. However, only one variant is valid at a given time; all other variants are undefined. You must store a value in a field of a particular variant before you attempt to use it. For example:

```
VAR
   x : INTEGER;
   a : RECORD
      CASE t : BOOLEAN OF
         TRUE   : ( b : INTEGER );
         FALSE  : ( c : REAL );
      END;
{In the executable section:}
x := a.b + 5;
a.c := 3.0;
x := a.b + 5;
```

Record a has two variants, `b` and `c`, which are located at the same storage address. When the assignment `a.c := 3.0` is executed, the value of a.b becomes undefined because TRUE is no longer the currently valid variant. When the statement `x := a.b + 5` is executed for the second time, the value of `a.b` is unknown. The compiler may choose not to evaluate `a.b` a second time because it has retained the field's previous value. To eliminate any misinterpretations caused by this assumption, variable a should be associated with the VOLATILE attribute. The compiler makes no assumptions about the value of VOLATILE objects.

For More Information:

- On variant records or on the VOLATILE attribute (*Compaq Pascal Language Reference Manual*)

4.3.6 Effects of Optimization on Debugging

Some of the effects of optimized programs on debugging are as follows:

- Use of registers

  When the compiler determines that the value of an expression does not change between two given occurrences, it may save the value in a register. In such a case, it does not recompute the value for the next occurrence, but assumes that the value saved in the register is valid. If, while debugging the program, you attempt to change the value of the variable in the expression, then the value of that variable is changed, but the corresponding value stored in the register is not. When execution continues, the value in the register may be used instead of the changed value in the expression, causing unexpected results.

When the value of a variable is being held in a register, its value in memory is generally invalid; therefore, a spurious value may be displayed if you try to examine a variable under these circumstances.

- Coding order

  Some of the compiler optimizations cause code to be generated in a order different from the way it appears in the source. Sometimes code is eliminated altogether. This causes unexpected behavior when you try to step by line, use source display features, or examine or deposit variables.

- Inline code expansion on user-declared routines

  There is no stack frame for an inline user-declared routine and no debugger symbol table information for the expanded routine. Debugging the execution of an inline user-declared routine is difficult and is not recommended.

To prevent conflicts between optimization and debugging, you should always compile your program with a compilation switch that deactivates optimization until it is thoroughly debugged. Then you can recompile the program (which by default is optimized) to produce efficient code.

For More Information:

- On debugging tools (Chapter 5)
- On compilation switches (Chapter 2)

4.4 Analyze Program Performance

This section describes how you can:

- Analyze program performance using shell commands like `time` (Section 4.4.1)
- Analyze program performance using profiling tools `prof`, `gprof`, and `pixie` (Section 4.5)
- Use feedback files and optionally `cord` to provide feedback for a subsequent compilation (Section 4.5.5)

Before you analyze program performance, make sure any errors you might have encountered during the early stages of program development have been corrected.

4.4.1 Use the time Command to Measure Performance

Use the `time` command to provide information about program performance.

Run program timings when other users are not active. Your timing results can be affected by one or more CPU-intensive processes also running while doing your timings.

Try to run the program under the same conditions each time to provide the most accurate results, especially when comparing execution times of a previous version of the same program. Use the same CPU system (model, amount of memory, version of the operating system, and so on) if possible.

If you do need to change systems, you should measure the time using the same version of the program on both systems, so you know each system's effect on your timings.

For programs that run for less than a few seconds, run several timings to ensure that the results are not misleading. Overhead functions like loading shared libraries might influence short timings considerably.

Using the form of the `time` command that specifies the name of the executable program provides the following:

- The elapsed or "wall clock" time, which will be greater than the total charged actual CPU time.
- Charged actual CPU time, shown for both *system* and *user* execution. The total actual CPU time is the sum of the actual user CPU time and actual system CPU time.

Using the Bourne shell, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

```
$ time a.out
Average of all the numbers is:    4368488960.000000
real    0m2.46s
user    0m0.61s
sys     0m0.58s
```

The sample program being timed displays the following line:

```
Average of all the numbers is:    4368488960.000000
```

Using the C shell, the following program timing reports 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use), about 4 seconds (0:04) of elapsed time, the use of 28% of available CPU time, and other information:

```
% time a.out
Average of all the numbers is:    4368488960.000000
0.61u 0.58s 0:04 28% 78+424k 9+5io 0pf+0w
```

Timings that show a large amount of system time may indicate a lot of time spent doing I/O, which might be worth investigating.

If your program displays a lot of text, you can redirect the output from the program on the `time` command line. Redirecting output from the program will change the times reported because of reduced screen I/O.

In addition to the `time` command, you might consider modifying the program to call routines within the program to measure execution time. For example:

- *Compaq Pascal* intrinsic procedures, such as SYSTEM_CLOCK, DATE_AND_TIME, and TIME
- Library routines, such as `itime` or `time` or `intro`(3f).

4.5 Profiling a Program

To generate profiling information, use the *Compaq Pascal* compiler and the `prof`, `gprof`, and `pixie` tools.

Profiling identifies areas of code where significant program execution time is spent. Along with the `pc`command, use the `prof` and `pixie` tools to generate the following profile information:

- The CPU time spent in the different routines of the program, or **program counter sampling**. This type of profiling uses `prof`.
- The manner in which routines are called by other routines, or **call graph information**. This type of profiling uses `gprof`.
- The execution of basic blocks, called **basic block counting**. A **basic block** is a sequence of instructions entered only at the beginning and exited only at the end (no branches). This provides statistics on individual lines of code and is influenced by such optimizations as loop unrolling. This type of profiling uses `prof` and `pixie`.
- The estimated number of CPU cycles spent for each source line in one or more procedures, or **source line CPU cycle use**. This type of profiling uses `prof` and `pixie`.

Once you have determined those sections of code where most of the program execution time is spent, examine these sections for coding efficiency.

4.5.1 Program Counter Sampling (prof)

To obtain program counter sampling data, perform the following steps:

1. Use the `pc` command option `-p` to compile and link the program:

   ```
   % pc -p -O3 -o profsample profsample.p
   ```

   If you specify the `-c` option to prevent linking, you must specify the `-p` option when you link the program:

   ```
   % pc -c -O3 profsample.p
   % pc -p -O3 -o profsample profsample.o
   ```

   Consider specifying optimization level `-O3` or `-inline manual` to minimize the inlining of procedures. Once inlined, procedures are not listed as separate routines but as part of the routine into which they have been inlined. Allowing full inlining would result in program counter sampling for a small number of (usually) large routines, which might not help you locate areas of the program where significant program execution time is spent.

2. Execute the profiled program:

   ```
   % profsample
   ```

   During program execution, profiling data is written to a profile data file, whose default name is `mon.out`. You can execute the program multiple times to generate multiple profile data files, which can be averaged. Use the PROFDIR environment variable to request a different profile data file name.

3. Run the `prof` command, which formats the profiling data and displays it in a readable format:

   ```
   % prof  profsample mon.out
   ```

You can limit the report created by `prof` by using `prof` command options, such as `-only`, `-exclude`, or `-quit`.

For example, if you only want reports on procedures calc_max and calc_min, you could use the following command line to read the profile data file named `mon.out`:

```
% prof  -only calc_max -only calc_min profsample
```

The time spent in particular areas of code is reported by `prof` in the form of a percentage of the total CPU time spent by the program. To reduce the size of the report, you can either:

- Request that only certain procedures be included (by using the `-only` option).
- Exclude certain procedures (by using the `-exclude` option).

When you use the `-only` or `-exclude` options, the percentages are still based on all procedures of the application. To obtain percentages calculated by `prof` that are based on only those procedures included in the report, use the `-Only` and `-Exclude` options (use an uppercase initial letter in the option name).

You can use the `-quit` option to reduce the amount of information reported. For example, the following command prints information on only the five most time-consuming procedures:

```
% prof -quit 5 profsample
```

The following command limits information only to those procedures using 10% or more of the total execution time:

```
% prof  -quit 10% profsample
```

4.5.2 Call Graph Sampling (gprof)

To obtain call graph information, use the `gprof` tool. Perform the following steps:

1. Use the `pc` command option `-pg` when you compile and link the program:

   ```
   % pc -pg -O3 -o profsample profsample.p
   ```

   If you specify the `-c` option to prevent linking, you must specify the `-pg` option when you link the program:

   ```
   % pc -c -O3 profsample.p
   % pc -pg -O3 -o profsample profsample.p
   ```

   The first `pc` command specifies the `-c` option to prevent linking.

2. Execute the profiled program:

   ```
   % profsample
   ```

   During execution, profiling data is saved to the file `gmon.out`, unless the environment variable PROFDIR is set.

3. Run the formatting program `gprof`:

   ```
   % gprof profsample gmon.out
   ```

The output produced by `gprof` includes:

- Call graph profile
- Timing profile (similar to that produced by `prof`)
- Index

4.5.3 Basic Block Counting (pixie and prof)

To obtain basic block counting information, perform the following steps:

1. Compile and link the program without the `-p` option:

   ```
   % pc -O3 -o profsample profsample.p
   ```

   Consider specifying optimization level `-O3` or `-inline manual` to minimize the inlining of procedures (once inlined, procedures are not listed as separate routines but as part of the routine into which they are inlined).

2. Run the profiling command `pixie`:

   ```
   % pixie profsample
   ```

   The `pixie` command creates:

   - A program named `profsample.pixie` that is equivalent to `profsample` but contains additional code for counting the execution of each basic block.
   - A file named `profsample.Addrs`, which contains the address of each basic block.

3. Execute the profiled program `profsample.pixie` generated by `pixie`:

   ```
   % profsample.pixie
   ```

   This program creates the file `profsample.Counts`, which contains the basic block counts.

4. Run `prof` with the `-pixie` option, to extract and display information from the `profsample.Addrs` and `profsample.Counts` files:

   ```
   % prof -pixie profsample
   ```

   When you specify the `-pixie` option, the `prof` command searches for files with a suffix of `.Addrs` and `.Counts` (in this case `profsample.Addrs` and `profsample.Counts`).

   You can reduce the amount of information in the report created by `prof` by using the `-only`, `-exclude`, `-quit`, and related options.

To create multiple profile data files, run the program multiple times.

4.5.4 Source Line CPU Cycle Use (prof and pixie)

You use the same files created by the `pixie` command (see Section 4.5.3) for basic block counting to estimate the number of CPU cycles used to execute each source file line.

To view a report of the number of CPU cycles estimated for each source file line, use the following options with the `prof` command:

- The `-pixie` option is required to obtain source line information.
- The `-heavy` option prints an entry for each source code line, including the number of CPU cycles used by that line. Entries are sorted in descending order of CPU cycles and should be limited by using the `prof` command options that limit the report size, such as `-quit`, `-only`, or `-exclude`.
- The `-lines` option requests source line information, but in the order in which the lines occur in the program (not sorted in descending order of CPU cycles).

Depending on the level of optimization chosen, certain source lines might be optimized away.

The CPU cycle use estimates are based primarily on the instruction type and its operands and do not include memory effects such as cache misses or translation buffer fills.

For example, the following command sequence uses:

- The `pc` and `pixie` commands to create the necessary files.
- The `prof` command to request source line CPU cycle use information for the procedure named calc_max (`-only` option), sorted in descending order of CPU cycles (`-heavy` option):

```
% pc -o profsample profsample.p
% pixie profsample
% profsample.pixie
% prof -pixie -heavy -only calc_max profsample
```

4.5.5 Creating and Using Feedback Files and Optionally cord

You can create a feedback file by using the `pc` command with the `-gen_feedback` option and the `pixie` command. Once created, specify a feedback file in a subsequent compilation with the `pc` command option `-feedback`. You can also request that `cord` use the feedback file to rearrange procedures, by specifying the `-cord` option on the `pc` command line.

To create the feedback file, create an executable program and use `pixie` to generate the additional code needed for profiling information:

1. Compile and link the program. Omit the `-p` option, but specify the `-gen_feedback` option:

   ```
   % pc -o profsample -gen_feedback -O3 profsample.p
   ```

   This command can be used with any optimization level up to `-O3` (to avoid inlining procedures). If you omit a `-On` option, the `-gen_feedback` option changes the default optimization level to `-O0`.

   To include libraries in the profiling output, specify `-non_shared`.

2. Execute the profiling command `pixie`:

   ```
   % pixie profsample
   ```

   The `pixie` command creates:

   - A program named `profsample.pixie` that is equivalent to `profsample` but contains additional code for counting the execution of each basic block.
   - A file named `profsample.Addrs`, which contains the address of each basic block.

3. Execute the profiled program `profsample.pixie` generated by `pixie`:

   ```
   % profsample.pixie
   ```

   This program creates the file `profsample.Counts`, which contains the basic block counts.

4. Run `prof` with the `-pixie` and `-feedback` options:

   ```
   % prof -pixie -feedback profsample.feedback
   profsample
   ```

   This `prof` command creates the feedback file `profsample.feedback`.

You can use the feedback file as input to the `pc` compiler:

```
% pc -feedback profsample.feedback -o profsample -O3
profssample.pc
```

The feedback file provides the compiler with actual execution information, which the compiler can use to perform such optimizations as inlining function calls.

Specify the same optimization level (`-On` option) for the `pc` command with the `-gen_feedback` option and the `pc` command with the `-feedback` *name* option (in this example `-O3`).

You can use the feedback file as input to the `pc` compiler and `cord`, as follows:

```
% pc -cord -feedback profsample.feedback -o profsample
-O3 profsample.p
```

The `-cord` option invokes `cord`, which reorders the procedures in an executable program to improve program execution, using the information in the specified feedback file. Use the same level of optimization with the `-cord` option as used to generate the feedback file.

4.6 Controlling the Size of Global Pointer Data

The compiler places constants and variables in the `.lit8` , `.lit4`, `.sdata` and `.sbss` sections of the data and bss segments of the global data area. These sections of the global data area are referred to as the global pointer area.

The `.rdata`, `.data`, `.lit8` , `.lit4`, and `.sdata` sections contain initialized data. The `.sbss` and `.bss` sections reserve space for uninitialized data that is created before execution by the kernel loader for the program and is filled with zeros.

The compiler produces code that uses a register as a global pointer, called `$gp`, to access address constants. The linker (`ld`) can, in some cases, optimize data accesses to use `$gp` directly rather than using an address constant.

To maximize the number of individual variables and constants that a program can access in the global pointer area, the compiler first places in it those variables and constants that take the fewest bytes of memory. By default, variables and constants occupying eight or fewer bytes are placed in the global pointer area, and those occupying more than eight bytes are placed in the `.data` and `.bss` sections.

You can control the maximum size of items that are placed in the global pointer area by using the `-G` option to the `pc` command. The `-G` option must be followed immediately by a number which specifies the maximum size (in bytes) of data items to be placed in the global pointer area. If not specified, a defualt of `-G8` is assumed.

5 Programming Tools

This chapter describes some Pascal-specific assistance provided in the set of Compaq CASE tools. For general information on each tool, see the documentation for the tool.

This chapter provides information on:

- The C Language Preprocessor (Section 5.1)
- Coding for the debugger (Section 5.2)
- Running the `dbx` Debugger (Section 5.3)
- Debugging tips (Section 5.4)

For More Information

- On the debugger (*Tru64 UNIX Programmer's Guide and* `dbx(1)`)
- On the C preprocessor (*Tru64 UNIX Programmer's Guide and* `cpp(1)`)

5.1 The C Language Preprocessor

The `pc` command submits Pascal source files to the `cpp` C language preprocessor. The `cpp` preprocessor reads directives from the source file to substitute strings, eliminate text, or include text from other files. The *Compaq Pascal* compiler operates on the resulting text.

You can use `cpp` to do the following:

- Eliminate repetition from source files
- Locate information in a central location for reference by other files in an application
- Declare global procedures and variables when you write an application consisting of many files
- Write source files with multiple uses, using symbolic names to enable or disable compilation of specified code

Capitalization rules for the preprocessor are different from those for the Pascal language itself, although the preprocessor is still case-sensitive. For example, you must type keywords such as `#define` by using lowercase only.

5.1.1 Including Headers and Other Files

A *Compaq Pascal* source file may include the text of other source files by using the `#include` directive, which is processed by `cpp`. (You can also use the *Compaq Pascal* %INCLUDE directive, which is processed by the *Compaq Pascal* compiler) and cannot contain `cpp` directives. The `#include` directive can take two forms:

```
#include "filename"
```

```
#include <filename>
```

The two forms specify a different search path for the specified file. (You can override this search path with the `-I` option.) `cpp` searches for the files in this way:

- If the file name begins with a slash, the preprocessor looks for the file only in the specified directory. The `-I` option does not affect these file names.
- If you use quotes (" ") for delimiters, the preprocessor looks for the file in the current working directory, then the files specified with the `-I` option, and lastly in `/usr/include`.
- If you use angle brackets (< >) for delimiters, the preprocessor does not search in the current working directory but starts in the directories specified with the `-I` option and lastly in `/usr/include`.
- If you specify the `-I` option without a search path, `cpp` does not search in `/usr/include`.

In place of `filename`, type the name of the file. The preprocessor passes the compiler the text of the named file in place of the `#include` line.

5.1.2 Conditional Compilation

The #if, #ifdef, #else, and #endif directives conditionally disable compilation of a range of lines from the source file based on a specific condition.

The #if directive can contain an expression that specifies some integer or Boolean computations. To do so, it uses C language syntax, instead of Pascal expression syntax. The expression cannot refer to constants and other data items declared in the Pascal program.

The preprocessor regards the symbol LANGUAGE_PASCAL as a defined symbol. However, the symbol LANGUAGE_C, LANGUAGE_FORTRAN, and LANGUAGE_ASSEMBLY are defined only when the preprocessor is used by the respective language product. You can use this fact to write a source file that can be submitted to several different language processors, with correct results on each.

Here is an example of conditional compilation:

```
#ifdef LANGUAGE_PASCAL
   TYPE Pair = RECORD
          HIGH, Low: INTEGER;
        END {RECORD};
#endif
#ifdef LANGUAGE_C
   typedef struct {
       int high, low;
       } pair;
#endif
```

This source file section is valid input to both *Compaq Pascal* and the C Language.

Compaq Pascal also provides a form of conditional compilation built into the compiler. See the *Compaq Pascal Language Reference Manual* for more information on the %IF directive.

5.1.3 String Substitution

The cpp preprocessor enables you to define macros to replace one string with another wherever it occurs in the source file. The #define directive specifies a macro, which can have parameters.

In *Compaq Pascal*, compile-time expressions in declarations can include any expression for which the values of all operands is known at compile time. This is an alternative to some uses of the #define directive.

5.1.4 C-Style Comments

The *Compaq Pascal Language Reference Manual* describes the Pascal-style { }, (* *), and ! rest of line comment delimiters. You can also use the C-style /* and */ delimiters in *Compaq Pascal* programs. When the symbol LANGUAGE_PASCAL is defined (that is, during *Compaq Pascal* compilations), `cpp` removes C-style comments. If you compile without `cpp`, the *Compaq Pascal* compiler disregards Pascal-style comments and reports syntax errors for C-style comments.

5.2 Coding for the Debugger

The compiler (`pc` command) detects syntax errors that prevent creation of an executable program. Debugging through `dbx` detects only errors of logic in your program. *Compaq Pascal* programs support source-level debugging; information from `dbx` relates directly to the source files rather than to machine registers, so you can access variables by their declared names rather than their memory addresses.

The `step` command of `dbx` enables you to execute specific source lines. To use it efficiently, follow these guidelines:

- Write each Pascal statement on a separate source line. This format enables `dbx` to execute each statement separately, so that you can check the effect of individual statements.
- Write each IF statement and the expression that it tests on a different line from the word THEN and the executable statement that follows. This format lets you use `dbx` to determine whether an error occurs in the evaluation of the expression or in executing the resulting statement.

5.2.1 Debugging Optimized Programs

You should use the optimizing phases described in Chapter 1 only after debugging the program. The `-g` option, which produces a program that can be debugged, overrides some optimizations. For example, the optimizer may move portions of code to gain efficiency or to share some sections. Such changes interfere with source-level debugging because the optimized sequence of actions no longer corresponds to the original source code.

The `-g` option is the same as specifying the `-g2` debugging level. If you do not use the `-g` option, *Compaq Pascal* uses the `-g0` debugging level. If you must debug optimized code, for example, if your program fails only when you enable optimization, you may need to use two other debugging levels invoked through the `-g1` and the `-g3` options:

- Level 1 `-g1` permits accurate, but limited, source-level debugging. If you have invoked the optimization phases, they continue to do most of their normal optimizations.
- Level 3 `-g3` directs the optimization phases to do all the normal optimizations, even if they rearrange code in a way that may produce confusing output from the debugger.

5.2.2 Debugging Preprocessed Programs

The `cpp` preprocessor is run at the start of every *Compaq Pascal* compilation (unless you use the `-nocpp` option). The `cpp` preprocessor substitutes strings as directed by your program. Preprocessing may add or remove text from the version of your source file seen by the compiler.

The debugger ignores any rearrangement of source lines from preprocessing. For example, when `dbx` traces machine code based on the statement on line 100 it shows and identifies line 100 despite possibilities such as the following:

- You used an `#include` preprocessor directive at line 1 of your file to include another 1000 lines of Pascal code by reference.
- You commented out or conditionalized out line 50 through 80.

The `dbx` debugger never refers to or displays statements inserted into a program by the `#include` preprocessor directive. All such statements appear in the trace as a single, indivisible statement. If you trace execution, you will see only the `#include` directive that refers to the file where these statements reside.

5.3 Running the dbx Debugger

You can invoke the debugger by entering the `dbx` command and any options to the shell, followed by the name of the executable file. The `dbx` debugger assumes a `a.out` or the name of a file containing a core dump.

You can exit the debugger by entering the `quit` command.

5.3.1 Debugger Data Types

Table 5–1 shows the `dbx` data types that correspond to the built-in data types of *Compaq Pascal*.

Table 5–1 The dbx Equivalents of Compaq Pascal Data Types

Compaq Pascal	dbx
BOOLEAN	$boolean
UNSIGNED or CARDINAL	$unsigned
CHAR	$char or $uchar
DOUBLE	$double or $real
INTEGER	$integer
POINTER	$address
REAL	$float (not $real)

For variables of other data types, use `dbx` type-coercion to view them as a `dbx` type, such as `$short`, a 16-bit integer.

5.3.2 Compaq Pascal Data Names

Pascal is not case sensitive; it defines all identifiers in lowercase. The `dbx` program is also not case sensitive.

Your program can have several variables with the same name. Each use of that name in a variable declaration has a specific scope. The particular variable you get when you use the name depends on the location of the reference. The *Compaq Pascal Language Reference Manual* discusses the rules for scope in *Compaq Pascal*.

When you are debugging, you may want to examine or modify a variable even if you are not currently in its scope. You can specify any declared program variable if you indicate its location. You do so by specifying a source file, then a procedure, then a variable name, using the period character as a separator. For example:

```
myfile.main.i
```

This example refers to a variable `i` in the routine `main` of the file `myfile` (or `myfile.p`).

The debugger uses this same notation when its output refers to a variable having an ambiguous name.

5.3.3 Activation Levels

You can examine activation levels or stack frames using the `where` command of `dbx`. You can move around on the stack with the `up`, `down`, and `func` commands.

In a *Compaq Pascal* executable program, each routine is an activation level. For example, the highest level on the activation stack is in the main program. The next highest level reflects the first active function or procedure call (called from the main program), which has not yet returned. At lower levels, you see other routines that were called. Level 0 reflects the most recent function or procedure call and shows the name of the current routine.

5.4 Debugging Tips

After using the debugger, you can edit the source file to correct the error or just note the error and continue tracing the program. However, you should always correct immediately any errors that cause additional errors during the debugging run. For example, if a program error assigns the wrong value to a variable, assign the correct value from `dbx`, note the error, and continue debugging.

Recompile all relevant source files and link-edit to produce a new executable file whenever you edit source files. If you correct a source file and continue debugging, `dbx` may produce confusing results because it may display lines from corrected source files that do not correspond to the program behavior.

If you stop `dbx` by typing Ctrl/Z, do not resume it to debug a corrected program. Instead, kill the process and reinvoke `dbx` so that it uses the new executable file.

6 Calling Conventions

This chapter describes how *Compaq Pascal* passes parameters and calls routines. It discusses the following topics:

- *Tru64 UNIX* Calling Standard (Section 6.1)
- Parameter-passing semantics (Section 6.2)
- Parameter-passing mechanisms (Section 6.3)

6.1 Tru64 UNIX Calling Standard

Programs compiled by the *Compaq Pascal* compiler conform to the *Tru64 UNIX* Calling Standard. This standard describes how parameters are passed, how function values are returned, and how routines receive and return control. Because *Compaq Pascal* conforms to the calling standard, you can call and pass parameters to routines written in other Compaq languages from *Compaq Pascal* programs.

For More Information:

- On the *Tru64 UNIX* Calling Standard (*Tru64 UNIX* Calling Standard and the *Assembly Language Programmer's Guide*)

6.1.1 Parameter Lists

Each time a routine is called, the *Compaq Pascal* compiler constructs a parameter list.

On *Tru64 UNIX* systems, the parameters are a sequence of quadword (8-byte) entries. The first 6 integer parameters are located in integer registers designated as R16 to R21; the first 6 floating-point parameters are located in floating-point registers designated as F16 to F21.

6.1.2 Function Return Values

In *Compaq Pascal*, a function returns to the calling block the value that was assigned to its identifier during execution. *Compaq Pascal* chooses one of three methods for returning this value. The method chosen depends on the amount of storage required for values of the type returned, as follows:

On Tru64 UNIX Systems:

- An nonfloating-point scalar type, a schematic subrange, an array, a record, or set with size less than 64 bits, is returned in the first integer register, designated as R0. If the value is less than 64 bits, R0 is sign-extended or zero-extended depending on the tape.
- A floating-point value that can be represented in 64 bits of storage is returned in the first floating-point register, designated as F0.
- If the value is too large to be represented in 64 bits, if its type is a string type (PACKED ARRAY OF CHAR, VARYING OF CHAR, or STRING), or if the type is nonstatic, the calling routine allocates the required storage. An extra parameter (a pointer to the location where the function result will be stored) is added to the beginning of the calling routine's actual parameter list.
- If the value can be represented in 32 bits of storage, it is returned in register R0. If the value is less than 32 bits, the upper bits of R0 are undefined.
- If the value requires from 33 to 64 bits, the low-order bits of the result are returned in register R0 and the high-order bits are returned in register R1.
- If the value is too large to be represented in 64 bits, if its type is a string type (PACKED ARRAY OF CHAR, VARYING OF CHAR, or STRING), or if the type is nonstatic, the calling routine allocates the required storage. An extra parameter (a pointer to the location where the function result will be stored) is added to the beginning of the calling routine's actual parameter list.

Note that functions that require the use of an extra parameter can have no more than 254 parameters; functions that store their results in registers can have 255 parameters.

6.1.3 Contents of Call Stack

The *Tru64 UNIX* system conventions define three types of procedures. The calling process does not need to know what type it is calling; the compiler chooses which type to generate based on the requirements of the procedure. The types of procedures are:

- Stack frame procedures, in which the calling context is placed on the stack
- Register frame procedures, in which the calling context is in registers
- No frame procedures, for which the compiler does not establish a context and which, therefore, execute in the context of the caller

If a stack frame is required, it consists of a fixed part (that is known at compile time) and an optional variable part.

The compiler determines the exact contents of the stack frame but all stack frames have common characteristics:

- Fixed temporary locations: an optional section that contains language-specific locations required by the procedure context of some languages
- Register save area: a set of consecutive quadwords for storing registers saved and restored by the current procedure
- Argument home area: if allocated, a region of memory used by the called process to assemble the arguments passed in registers adjacent to the arguments passed in memory. This allows all arguments to be addressed as a contiguous array. The argument home area is also used to store arguments passed in registers if an address for such an argument is required.
- Arguments passed in memory

6.1.4 Unbound Routines

The frame pointer of calling routines is stored in an implementation-defined register. If, however, you declare a routine with the UNBOUND attribute, the system does not assume that the frame pointer of the calling routine is stored in a register and there is no link between the calling and the called routines. As a result, an unbound routine has the following restrictions:

- It cannot access automatic variables declared in enclosing blocks.
- It cannot call bound routines declared in enclosing blocks.
- It cannot use a GOTO statement to transfer control to enclosing blocks other than the main program block.

By default, routines declared at program or module level and all other routines declared with the INITIALIZE, GLOBAL, or EXTERNAL attributes have the characteristics of unbound routines. Routines passed by the immediate value mechanism must be UNBOUND.

For More Information:

- On attributes (*Compaq Pascal Language Reference Manual*)
- On the immediate value mechanism (Section 6.3.1)

6.2 Parameter-Passing Semantics

Parameter-passing semantics describe how parameters behave when passed between the calling and called routine. *Compaq Pascal* passes parameter values by the following methods:

- Value passing semantics (Standard)
- Variable passing semantics (Standard)
- Foreign passing semantics (*Compaq Pascal* extension)

By default, *Compaq Pascal* passes arguments using value semantics.

For More Information:

- On value, variable, and foreign semantics (*Compaq Pascal Language Reference Manual*)

6.3 Parameter-Passing Mechanisms

The way in which an argument specifies how the actual data to be passed by the called routine is defined by the **parameter-passing mechanism**. In compliance with the *Tru64 UNIX* Calling Standard, *Compaq Pascal* supports the basic parameter-passing mechanisms, shown in Table 6–1.

Table 6–1 Parameter-Passing Mechanisms Systems

Mechanism	Description
By immediate value	The argument contains the value of the data item.
By reference	The argument contains the address of the data to be used by the routine.

(continued on next page)

Table 6–1 (Cont.) Parameter-Passing Mechanisms Systems

Mechanism	Description
By descriptor	The argument contains the address of a descriptor, which describes type of the data and its location.

By default, *Compaq Pascal* uses the by reference mechanism to pass all actual parameters except those that correspond to conformant parameters and undiscriminated schema parameters, in which case the by descriptor mechanism is used. Table 6–2 describes the method you use in *Compaq Pascal* to obtain the desired parameter-passing mechanism.

Table 6–2 Parameter-Passing Methods

Mechanism	Methods Used by Compaq Pascal
By immediate value	%IMMED or [IMMEDIATE]
By reference	Default for nonconformant and nonschema parameters or %REF
By descriptor	Default for conformant and schema parameters or %DESCR, %STDESCR, [CLASS_S], or [CLASS_NCA]

A mechanism specifier usually appears before the name of a formal parameter, or if a passing attribute is used it appears in the attribute list of the formal parameter. However, in *Compaq Pascal*, a mechanism specifier can also appear before the name of an actual parameter. In the latter case, the specifier overrides the type, passing semantics, passing mechanism, and the number of formal parameters specified in the formal parameter declaration.

For More Information:

- On passing mechanisms and passing semantics (Section 6.3)

6.3.1 By Immediate Value Passing Mechanism

The by immediate value passing mechanism passes a copy of a value instead of the address. *Compaq Pascal* provides the %IMMED foreign passing mechanism and the IMMEDIATE attribute in order to pass a parameter by immediate value. You cannot use variable semantics with the by immediate value passing mechanism.

On *Tru64 UNIX* systems, scalars, floating points, records, and sets that are less than or equal to 64 bits can be passed by immediate value.

6.3.2 By Reference Passing Mechanism

The by reference passing mechanism passes the address of the parameter to the called routine. This is default parameter-passing mechanism for non-conformant and non-schematic parameters.

When using the by reference mechanism, the type of passing semantics used depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used.

In addition to using the defaults, the *Compaq Pascal* compiler provides the %REF foreign passing mechanism and the REFERENCE attribute, which has more than one interpretation for the passing semantics depending on the data item represented by the actual parameter. This allows you to have the called routine use either variable semantics or true foreign semantics. The mechanism specifier appears before the name of a formal parameter. The parameter passing attribute appears in the attribute list of the formal parameter.

6.3.3 By Descriptor Passing Mechanism

The by descriptor passing mechanism passes the address of a data structure called a descriptor. Descriptors are used to describe parameters that could not be passed with just an address. Descriptors hold additional information such as lower and upper bounds, sizes of intermediate dimensions, etc. *Compaq Pascal* uses the by descriptor mechanism for conformant and schematic parameters.

Compaq Pascal provides the [CLASS_S] and [CLASS_NCA] attribute for the by descriptor mechanism. With this attribute, the type of passing semantics used for the by descriptor argument depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used. The [CLASS_S] or [CLASS_NCA] attribute appears in the attribute list of the formal parameters.

Sometimes you may want to choose either variable semantics or true foreign semantics. In these cases, the *Compaq Pascal* compiler provides the two foreign passing mechanism specifiers, %DESCR and %STDESCR. These specifiers have more than one interpretation for the passing semantics depending on the data type of the actual parameter. The mechanism specifier appears before the name of a formal parameter.

For conformant array parameters without the [CLASS_S] attribute, the descriptor looks like:

```
struct {
        long pointer-to-data
        long total-data-size
        long element-size
        long pointer-to-virtual-base (aka, A[0] pointer)
        long dim-1-lower-bound
        long dim-1-upper-bound
        long dim-1-stride
```

with the last 3 long fields repeated for any additional dimensions

For conformant array parameters with a [CLASS_S] attribute, the descriptor looks like:

```
struct {
        long pointer-to-data
        long data-size
        }
```

For conformant variable parameters, the descriptor looks like:

```
struct {
        long pointer-to-varying-data
        long maximum-length
        }
```

The descriptors used for schematic parameters are Pascal-specific and undocumented.

6.4 Passing Parameters Between Compaq Pascal and Other Languages

Passing parameters between *Compaq Pascal* and other languages on *Tru64 UNIX* systems requires some additional knowledge about the semantics and mechanisms used by the *Compaq Pascal* and the other compilers involved.

6.4.1 Parameter Mechanisms Versus Parameter Semantics

The Pascal language provides three parameter semantics, "VAR parameters", "value parameters", and "routine parameters". These models define what happens to the parameters, not how the compiler actually implements them. "VAR parameters" are parameters that represent the actual variable passed to the routine. Changes made to the VAR parameter are reflected back to the actual variable passed in to the routine. "Value parameters" are parameters that are local copies of the expression passed into the routine. Changes made to the value parameter are not reflected back to any actual parameter passed in to the routine. "Routine parameters" are parameters that represent entire routines that may be called from inside the called routine.

The *Compaq Pascal* compiler provides three parameter mechanisms, "by immediate value", "by reference", and "by descriptor". These forms represent the actual implementation used by the compiler for the parameter. These forms are denoted by the [IMMEDIATE], [CLASS_S], and [CLASS_NCA] attributes (note, the [REFERENCE] attribute doesn't just specify a parameter mechanism, but also specifies a parameter semantic model).

Compaq Pascal also provides a fourth parameter model called "foreign parameters". These parameters become either VAR or value parameters depending on the actual parameter. If the actual parameter is a variable, then the parameter is treated as a VAR parameter. If the actual parameter is an expression, then the parameter is treated as a value parameter. These parameters are denoted by the %REF, %DESCR, and %STDESCR foreign mechanism specifiers and the [REFERENCE] attribute (identical in behavior to the %REF foreign parameter specifier).

Be careful not to confuse the term "value parameter" with the "by immediate value" mechanism. The "value" in "value parameter" describes the semantics of the parameter where changes made to the parameter inside the called routine are not reflected back to the actual parameter. It is a common misconception that *Compaq Pascal* uses the "by immediate value" mechanism for "value parameters."

6.4.2 Passing Non-Routine Parameters Between Compaq Pascal and Other Languages

By default, *Compaq Pascal* will use the "by reference" mechanism for the following VAR and value parameter types: Ordinal (integer, unsigned, char, Boolean, pointers, subranges, and enumerated types), Real (real, double, quadruple), Record, Array, Set, Varying, and File.

If you want to pass a parameter with the "by immediate value" mechanism, you can place the [IMMEDIATE] attribute on the definition of the formal parameter's definition or use the %IMMED foreign mechanism specifier on the actual parameter to override the default mechanism of the formal parameter. Only ordinal and real types may be passed with the "by immediate value" mechanism. Only value parameters may use the "by immediate value" mechanism.

If you want to accept a value parameter with the "by immediate value", you can place the [IMMEDIATE] attribute on the definition of the formal parameter. Only ordinal and real types may be accepted with the "by immediate value" mechanism.

For example, to pass an integer with the "by immediate value" mechanisn to another routine,

```
[external] procedure rtn( p : [immediate] integer ); external;

begin
rtn(3);
rtn(some-integer-expression);
end;
```

If you want to pass a parameter with the "by descriptor" mechanism, you can place the [CLASS_S], or [CLASS_NCA] attributes on the formal parameter's definition. You can also use the %DESCR and %STDESCR foreign mechanism specifiers, but be aware that these also imply parameter semantics as well as the parameter-passing mechanism.

When passing values to a subrange parameter in a Pascal routine, the argument must be large enough to hold a value of the subrange's base-type even if the formal parameter contained a size attribute.

When passing Boolean or enumerated-type values to a VAR parameter in a Pascal routine, the calling routine must ensure that the sizes of the Boolean or enumerated-type matches the setting of the `-enumeration_size` option qualifier or [ENUMERATION_SIZE] attribute used in the Pascal routine. For value parameters, you can pass the address of a longword as that will work for either setting.

When passing arrays or records to a Pascal routine, the calling routine must ensure that the array and record has the same layout (including any alignment holes) as chosen by the *Compaq Pascal* compiler. You may want to use the -`show structure_layout` listing section to help you determine the layout chosen by the *Compaq Pascal* compiler.

By default, *Compaq Pascal* will use the "by descriptor" mechanism for VAR and value conformant parameters.

For More Information

- On by descriptor mechanism (Section 6.3.3)

Using a conformant-varying parameter or STRING schema parameter with a routine not written in Pascal is not very common since the called routine does not know how to deal with these strings. If you just are passing a string expression to the non-Pascal routine, using a conformant PACKED ARRAY OF CHAR is the right solution.

Since *Compaq Pascal* will use either a descriptor for the conformant PACKED ARRAY OF CHAR, but other languages will expect either the string "by reference", you will need to use the %REF foreign mechanism specifier.

To pass a string expression to C (which expects a "by-reference" parameter and a null-terminated string),

```
[external] procedure crtn(
                %ref p : packed array [l..u:integer] of char); external;

  begin
crtn('string'(0));
  crtn(some-string-expression+'0');
  end;
```

Compaq Pascal on *Tru64 UNIX* systems has additional support to deal with null-terminated strings.

For More Information

- *Compaq Pascal Language Reference Manual*

When passing strings to a *Compaq Pascal* routine from another language, you must use a descriptor if the Pascal formal parameter is a conformant parameter. *Compaq Pascal* cannot accept a conformant parameter with the "by reference" mechanism.

If you wish to use the "by reference" mechanism to pass strings into a Pascal routine, you must define a Pascal datatype that represents a fixed-length string (or varying-string with a maximum size) and use that datatype in the formal parameter definition.

The *Compaq Pascal* schema type STRING is passed by descriptor. Other *Compaq Pascal* schema types use private data structures when passed between routines and cannot be accessed from routines written in other languages.

6.4.3 Passing Routine Parameters Between Compaq Pascal and Other Languages

By default, *Compaq Pascal* on *Tru64 UNIX* passes the address of a procedure descriptor for PROCEDURE or FUNCTION parameters. The presence of the [UNBOUND] attribute or the %IMMED foreign mechanism specifier has no effect over the generated code since the procedure descriptors in the *Tru64 UNIX* Calling Standard allow any combination of bound and unbound routines to be passed around and invoked.

On *Tru64 UNIX*, *Compaq Pascal* expects the address of a procedure descriptor for routine parameters. In all Alpha languages, asking for the address of a routine, yields the address of its procedure descriptor since the actual address of the instructions is not useful by itself.

6.5 Calling C Routines from Compaq Pascal

One method you can use to pass strings from Pascal to C routines is the null-terminated string support routine described in Section 6.4.2. Another method is the existing Pascal mechanisms and decode the Pascal-specific descriptor passed to the C routines. A summary of the calling sequences used by Pascal for various parameters is shown in the following list. Note that for parameters passed with value semantics, the compiler assumes that any data pointed to by the argument list or descriptors is not modified by the called routine.

- For non-conformant parameters (both VAR and value semantics), a pointer to the variable (or expression) is passed. *Compaq Pascal* does not pass any arguments by "immediate value" by default. If you want this behavior, you can place the [IMMEDIATE] attribute on the formal parameter definition or use the %IMMED foreign mechanism specifier on the actual parameter.
- For conformant array parameters without a CLASS_S attribute (both VAR and value semantics), a pointer to a Pascal-specific descriptor is passed. The descriptor can be constructed as follows:

  ```
  struct {
   long pointer-to-data
   long total-data-size
   long element-size
   long pointer-to-virtual-base (aka, A[0] pointer)
   long dim-1-lower-bound
   long dim-1-upper-bound
   long dim-1-stride
   }
  ```

  The last three longs can be repeated for any additional dimensions.
- For conformant array parameters with a CLASS_S attribute (both VAR and value semantics), a pointer to a Pascal-specific descriptor is passed. The descriptor can be constructed as follows:

  ```
  struct {
   long pointer-to-data
   long data-size
   }
  ```

- For conformant varying parameters (both VAR and value semantics), a pointer to a Pascal-specific descriptor is passed. The descriptor can be constructed as follows:

```
struct {
 long pointer-to-varying-data
 long maximum-length
 }
```

7 Error Processing and Condition Handling

An exception condition is an event, usually an error, that occurs during program execution and is detected by system hardware or software or the logic in a user application program. A **condition handler** is a routine that is used to resolve exception conditions.

This chapter discusses the following topics:

- Condition handling terms (Section 7.1)
- Overview of condition handling (Section 7.2)
- Writing condition handlers (Section 7.3)
- Examples of condition handlers (Section 7.4)

7.1 Condition Handling Terms

The following terms are used in the discussion of condition handling:

- Condition value—An integer value that identifies a specific condition.
- Stack frame—A standard data structure built on the stack during a routine call, starting from the location addressed by the frame pointer (FP) and proceeding to both higher and lower addresses; it is popped off the stack during the return from a routine.

- Routine activation—The environment in which a routine executes. This environment includes a unique stack frame on the run-time stack; the stack frame contains the address of a condition handler for the routine activation. A new routine activation is created every time a routine is called and is deleted when control passes from the routine.
- Establish—The process of placing the address of a condition handler in the stack frame of the current routine activation. A condition handler established for a routine activation is automatically called when a condition occurs. In *Compaq Pascal*, condition handlers are established by means of the predeclared procedure ESTABLISH. A routine that establishes a condition handler is known as an establisher.
- Signal—The means by which the occurrence of an exception condition is made known. Signals are generated by the operating system in response to I/O events and hardware errors, by the system-supplied library routines, and by user routines. All signals are initiated by a call to the signaling facility, for which there are two entry points:
- Resignal—The means by which a condition handler indicates that the signaling facility is to continue searching for a condition handler to process a previously signaled error. To resignal, a condition handler returns the value `ExceptionContinueSearch` defined in `/usr/include/excpt.pas`.
- Unwind—The return of control to a particular routine activation, bypassing any intermediate routine activations. For example, if X calls Y, and Y calls Z, and Z detects an error, then a condition handler associated with X or Y can unwind to X, bypassing Y. Control returns to X immediately following the point at which X called Y.

7.2 Overview of Condition Handling

To support the ESTABLISH and REVERT builtins, non-local GOTOs, and to signal and print run-time error messages, the *Compaq Pascal* run-time library (libpas) uses the stack-based exception handling mechanism. See `man exception_intro` for more information.

At the beginning of any program that uses the Pascal run-time library, the library does the following:

1. Registers the EXC_Raise_Signal_Exception routine as a signal handler for the SIGTRAP and SIGFPE signals using sigaction( ).
2. Registers a private last-chance handler with EXC_Set_Last_Chance_ Handler( ).

If your program registers its own signal handlers for SIGTRAP or SIGFPE, or registers a different last-chance handler, then errors signaled at run time might not produce the expected run-time messages, and language handlers might not be called to handle the error. Other language features, such as non-local GOTOs, should continue to work as expected. In addition, the special last-chance handler is used only when the main program is not written in Pascal.

7.2.1 Condition Signals

A condition signal consists of a call to `exc_raise_exception`. This entry point and data structure can be included from `/usr/include/excpt.pas`.

If a condition occurs in a routine that is not prepared to handle it, a signal is issued to notify other active routines. If the nature of the condition does not allow the current routine to continue, you should set the EXCEPTION_NONCONTINUABLE flag in the exception record.

7.2.2 Handler Responses

A condition handler responds to an exception condition by taking action in three major areas:

- Condition correction
- Condition reporting
- Execution control

The handler first determines whether the condition can be corrected. If so, it takes the appropriate action and execution continues. If the handler cannot correct the condition, the condition may be resignaled; that is, the handler requests that another condition handler be sought to process the condition.

A handler's condition reporting can involve one or more of the following actions:

- Maintaining a count of exceptions encountered during program execution
- Resignaling the same condition to send the appropriate message to the output file

- Signaling a different condition, for example, the production of a message designed for a specific application

A handler can control execution in several ways:

- By doing a nonlocal GOTO operation (see Section 7.4, Example 5).
- By unwinding to the establisher at the point of the call that resulted in the exception. The handler can then determine the function value returned by the called routine.
- By unwinding to the establisher's caller (the routine that called the routine which established the handler). The handler can then determine the function value returned by the called routine.

7.3 Writing Condition Handlers

The following sections describe how to write and establish condition handlers and provide some simple examples.

7.3.1 Establishing and Removing Handlers

To use a condition handler, you must first declare the handler as a routine in the declaration section of your program; then, within the executable section, you must call the predeclared procedure ESTABLISH. The ESTABLISH procedure sets up a *Compaq Pascal* language-specific condition handler that in turn allows your handler to be called. User-written condition handlers set up by ESTABLISH must have the ASYNCHRONOUS attribute and two integer array formal parameters. Such routines can access only local, read-only, and volatile variables, and local, predeclared, and asynchronous routines.

Because condition handlers are asynchronous, any attempt to access a non-read-only or nonvolatile variable declared in an enclosing block will result in a warning message. The predeclared file variables INPUT and OUTPUT are such nonvolatile variables; therefore, simultaneous access to these files from both an ordinary program and from an asynchronous condition handler's activation may have undefined results. The following steps outline the recommended method for performing I/O operations from a condition handler:

1. Declare a file with the VOLATILE attribute at program level.
2. Open this file to refer to INPUT, OUTPUT, or another appropriate file.
3. Use this file in the condition handler.

External routines (including system services) that are called by a condition handler require the ASYNCHRONOUS attribute in their declaration.

The following example shows how to establish a condition handler using the *Compaq Pascal* procedure ESTABLISH:

```
[EXTERNAL,ASYNCHRONOUS] FUNCTION Handler
   (VAR ERecord  : System_Exrec_Type;
        EFrame   : INTEGER_ADDRESS;
    VAR EContext : Sigcontext;
    VAR DContest : Dispatcher_Context);
                                 INTEGER64;
   EXTERN;
   .
   .
   .
ESTABLISH (Handler);
```

To establish the handler, call the ESTABLISH procedure, as follows:

```
ESTABLISH(Handler);
```

To remove an established handler, call the predeclared procedure REVERT, as follows:

```
REVERT;
```

As a result of this call, the condition handler established in the current stack frame is removed. When control passes from a routine, any condition handler established during the routine's activation is automatically removed.

7.3.2 Declaring Parameters for Condition Handlers

A *Compaq Pascal* condition handler on *Tru64 UNIX* systems is a function returning an INTEGER64 value and has 4 parameters. The four parameters are:

- ExceptionRecord - the address of the primary exception record. (see /usr /include/excpt.h)
- EstablisherFrame - the virtual frame pointer of the establisher
- ContextRecord - the address of an invocation context block containing the saved original context at the point where the exception occurred. During an unwind, this is the address of the invocation context block for the establisher. (see /usr/include/signal.h)
- DispatcherContext - the address of a control record for the exception dispatcher. (see /usr/include/excpt.h)

For your convenience, an include file containing Pascal-equivalent portions of excpt.h, signal.h and psdc.h has been provided in /usr/include/excpt.pas. You can use this include file when writing condition handlers in *Compaq Pascal* on *Tru64 UNIX* Systems.

7.3.3 Handler Function Return Values

Condition handlers are functions that return values to control subsequent execution. These values and their effects are listed as follows:

Value	Effect
ExceptionContinueExecution	Continues execution from the signal. If the signal has the EXCEPTION_NOCONTINUABLE flag set, the program does not continue, but exits.
ExceptionContinueSearch	Resignals to continue the search for a condition handler to process the condition.

In addition, a condition handler can request a stack unwind by calling the EXC_VIRTUAL_UNWIND or EXC_UNWIND system service routines. For example:

7.4 Example of a Condition Handler

The following is an example of a condition handler on a *Tru64 UNIX* system.

```
program foo(input,output);
label 1;

var i : integer;
    s : interger;

CONST
    %include 'pasdef.pas'

%include 'excpt.pas'
[asynchronous]
function user_handler(
   VAR ExceptionRecord : System_Exrec_Type;
       EstablisherFrame  : INTEGER_ADDRESS;
   VAR ContextRecord : Sigcontext;
   VAR DispatcherContext : Dispatcher_Context) : interger64;
   var f : text;
       CtxRecord :Sigcontext;
       Callers_PC : integer64;

begin
if UAND(ExceptionRecord.ExceptionFlags,EXCEPTION_UNWINDING) <> 0
then
    { unwind in progress.  Do cleanup (if any) and continue search }
    return ExceptionContinueSearch;
open(f,'OUTPUT',history :=new);
rewrite(f);
writeln(f,'In HANDLER with ',ExceptionRecord.ExceptionCodeNumber:1);
case ExceptionRecord.ExceptionCodeNumber of
```

```
pas$_invsynint: { Invalid syntax for integer, continue }
user_handler := ExceptionContinueExecution;

  pas$_invsynint: {Invalid syntax for integer, continue }
user_handler := ExceptionContinueExecution;

  pas$subasgval: { Subrange assignment out of range, do a non-local GOTO }
goto 1;

  otherwise
user_handler :=ExceptionContinueSearch;
  end;
  end;

procedure process_data;
    var i : integer;
    begin
    establish(user_handler);
    readln(i);
    writeln(i);
    end;

begin
writeln('Calling process_data');
process_data;
1:
writeln('Return from process data');
end.
```

8 Migrating from OpenVMS to Tru64 UNIX Systems

The following sections describe issues that affect *Compaq Pascal* programs being moved from OpenVMS systems to *Tru64 UNIX* systems.

8.1 Sharing Environment Files Across Platforms

The compiler can only inherit environment files created from a compiler for the same target platform. For example, you cannot inherit environment files generated on an *OpenVMS VAX* system with the *Compaq Pascal* for *Tru64 UNIX* compiler.

8.2 Default Size for Enumerated Types and Booleans

On both *Tru64 UNIX* and *OpenVMS Alpha* systems, the default size for enumerations and Booleans in unpacked structures is longword. On *OpenVMS VAX* systems, the default is a byte for Booleans and small enumerations or word for larger enumerations.

If you need the *OpenVMS VAX* behavior on *Tru64 UNIX* systems, you can:

- Use the `-enumeration_size byte` command qualifier
- Use the [ENUMERATION_SIZE(BYTE)] attribute
- Place individual [BYTE] or [WORD] attributes on the affected fields or components.

8.3 Default Data Layout for Unpacked Arrays and Records

On both *Tru64 UNIX* and *OpenVMS Alpha*, the default data layout is natural alignment. This means that record fields and array components are aligned on boundaries based on their size. For example, INTEGER on longword boundaries, and INTEGER64 on quadword boundaries.

On *OpenVMS VAX* systems, the default alignment rule is to allocate such fields on the next byte boundary. If you need the *OpenVMS VAX* behavior on *Tru64 UNIX* systems, you can use one of the following:

- `-align vax` command option
- [ALIGN(VAX)] attribute

8.4 IADDRESS and VOLATILE

The IADDRESS built-in assumes that its parameter is one of the following:

- VOLATILE variable
- VOLATILE parameter
- Routine entry point

Unlike the ADDRESS built-in, the IADDRESS built-in does not issue a warning if the parameter does not have the VOLATILE attribute.

On both *Tru64 UNIX* and *OpenVMS Alpha*, the *Compaq Pascal* compiler often allocates variables so that they exist for the entire routine in which they were declared. In these situations, use the IADDRESS built-in to obtain the address of the variable. The address is passed to a system service through an item list or something similar.

On *Tru64 UNIX* systems, the *Compaq Pascal* compiler is more aggressive with optimizing data layout on the stack. In the absence of a VOLATILE attribute, the compiler allocates variables for the smallest possible duration. If the address is taken with IADDRESS, by the time the address is written into by a library routine, the variable may no longer exist and the memory store would corrupt another variable.

If the IADDRESS built-in is used on automatic variables or parameters, then the VOLATILE attribute must be used to ensure proper behavior.

8.5 Overflow Checking

When overflow checking is enabled on *Tru64 UNIX*, the INT built-in signals a run-time error if its actual parameter cannot be represented as an INTEGER32 value.

If you have a large unsigned value that you wish to convert to a negative integer, you must use a typecast to perform the operation.

8.6 Bound Procedure Values

On *OpenVMS VAX* systems, a Bound Procedure Value is a 2-longword data structure holding the address of the entry point and a frame-pointer to define the nested environment. *Compaq Pascal* expects one of these 2-longword structures for PROCEDURE or FUNCTION parameters.

A routine not written in Pascal needs different code depending on whether it will receive a Bound Procedure Value versus a simple routine address. When passing routines to %IMMED formal routine parameters, *Compaq Pascal* passes the address of the entry point; otherwise, it passes the address of a Bound Procedure Value.

On both *Tru64 UNIX* and *OpenVMS Alpha* systems, a Bound Procedure Value is a special type of procedure descriptor that invokes a hidden jacket routine that initializes the frame-pointer and calls the real routine. Given this structure, a routine that is calling another routine indirectly does not need to do anything special for Bound Procedure Values.

When passing routines to %IMMED formal routine parameters, (or asking for the IADDRESS of a routine) *Compaq Pascal* passes the address of a procedure descriptor as if the %IMMED was not present. There is no direct way in *Compaq Pascal* to obtain the actual code address of a routine since, it is not generally useful without the associated procedure descriptor.

8.7 Argument List Functions

The *Tru64 UNIX* calling standard does not have an architected method for determining an argument count. The following built-in routines are not supported on *Tru64 UNIX* systems:

- ARGUMENT
- ARGUMENT_LIST_LENGTH
- PRESENT

The [TRUNCATE] attribute is not supported, and the [LIST] attribute can only be used on external routine definitions.

8.8 %DICTIONARY Directive

The %DICTIONARY directive is not supported on *Tru64 UNIX* systems.

8.9 VAX Floating Datatypes

The following VAX floating datatypes are not supported on *Tru64 UNIX* systems:

- F_FLOAT
- G_FLOAT
- D_FLOAT
- H_FLOAT

The IEEE_FLOAT option to the [FLOAT()] attribute is supported.

8.10 Relative and Indexed Files

The following keywords on the OPEN and CLOSE statement are not supported on *Tru64 UNIX* systems:

- DEFAULT
- DISPOSITION := PRINT
- PRINT_DELETE
- SUBMIT
- SUBMIT_DELETE
- SHARING
- USER_ACTION
- RECORD_TYPE := STREAM or STEAM_CR

The following built-in routines are not supported:

- DELETE
- FIND
- FINDK
- LOCATE
- RESETK
- UPDATE
- UNLOCK

8.11 Data Layout and Conversion

On Alpha systems (and to a lesser extent VAX systems), the layout of data can severely impact performance. The Alpha architecture and the current Alpha systems have strong preferences about data alignment and size.

The *Compaq Pascal* compiler has several features to enable you to write Pascal code that will get the best performance on the target system.

The remainder of this section describes the different types of record layouts, *Compaq Pascal* features that support them, how to get the best performance with your data structures, and how to convert existing code for better performance.

This section focuses on records, but arrays also have similar properties. In almost all cases, where record fields are discussed, you can substitute array components.

If you are converting from *OpenVMS Alpha* to *Tru64 UNIX*, you may have already dealt with most of the alignment and datatype issues.

8.11.1 Natural Alignment, VAX Alignment, and Enumeration Sizes

The compiler has the ability to lay out records in two ways:

- *OpenVMS VAX* alignment

  Fields and components less than or equal to 32 bits are allocated on the next available bit; otherwise they are allocated on the next available byte.

- Natural alignment where an object is aligned based on its size

  Essentially fields and components are allocated on the next naturally aligned address for their data type. For example:

– 8-bit character strings should start on byte boundaries

– 16-bit integers should start at addresses that are a multiple of 2 bytes (word alignment)

– 32-bit integers and single-precision real numbers should start at addresses that are a multiple of 4 bytes (longword alignment)

– 64-bit integers and double-precision real numbers should start at addresses that are a multiple of 8 bytes (quadword alignment)

For aggregates such as arrays and records, the data type to be considered for purposes of alignment is not the aggregate itself, but rather the elements of which the aggregate is composed. Varying 8-bit character strings must, for example, start at addresses that are a multiple of 2 bytes (word alignment) because of the 16-bit count at the beginning of the string. For records, the size is rounded up to a multiple of their natural alignment (a record with natural alignment of longword has a size that is a multiple of longwords, for example).

The *OpenVMS VAX* and naturally aligned record formats are fully documented in the *OpenVMS Calling Standard*. The *Tru64 UNIX Calling Standard* also documents the naturally aligned record format.

The size as well as the alignment of record fields and array components can affect performance. For example, accessing a byte or word on an Alpha system requires more instructions than accessing a naturally aligned longword. On Alpha systems, *Compaq Pascal* uses larger allocation for unpacked Booleans and enumeration types to help performance, as shown in Table 8–1

Table 8–1 Unpacked Sizes of Fields and Components

Datatype	Unpacked Size on VAX	Unpacked Size on Alpha
Boolean	1 byte	4 bytes
Enumerated types	1 or 2 bytes	4 bytes

For compatibility reasons, the size of all data types in PACKED records and arrays are the same for both VAX and natural alignment formats.

8.11.2 Compaq Pascal Features Affecting Data Alignment and Size

The *Compaq Pascal* for *Tru64 UNIX* compiler has the following command line options:

- `-align keyword`, where `keyword` is either alpha_axp or vax
- `-enumeration_size keyword`, where `keyword` is either byte or long

The `-align` option controls the default record format used by the compiler. The `-enumeration_size` option controls whether the compiler allocates Boolean and enumeration types as longwords or as 1 or 2 bytes.

On VAX systems, the default alignment format is "VAX" and the default enumeration size is "BYTE". On Alpha systems, the default alignment format is "ALPHA_AXP" and the default enumeration size is "LONG".

A corresponding pair of attributes can be used at the PROGRAM/MODULE level and on VAR and TYPE sections to specify the desired alignment format and enumeration size:

- ALIGN(keyword), where keyword is either ALPHA_AXP or VAX
- ENUMERATION_SIZE(keyword), where keyword is either BYTE or LONG

By using these attributes at the MODULE level, you can extract the records into a separate module and create an environment file with the desired alignment format. By using these attributes on VAR or TYPE sections, you can isolate the records in the same source file.

8.11.3 Optimal Record Layout

The optimal record layout is one where all the record's fields are naturally sized on naturally aligned boundaries and the overall record is as small as possible (for example, the fewest number of padding bytes required for proper alignment).

On Alpha systems, the compiler automatically places all fields of unpacked records on naturally aligned boundaries. On VAX systems, you have to explicitly ask for natural alignment by using either a DCL qualifier or the corresponding attribute.

To allow the compiler to do this placement, you should refrain from using explicit positioning and alignment attributes on record fields unless required by your application. The keyword PACKED should be avoided in all cases except:

- PACKED ARRAY OF CHARs require the PACKED keyword to be manipulated as strings. Since chars are each 1 byte, using the PACKED keyword does not hurt their performance in any way.
- PACKED SETs may perform better than unpacked SETs. For PACKED SETs, the compiler can sometimes allocate fewer bits for the set field or variable. These smaller sets can often be manipulated directly with longword or quadword instructions, instead of using a generic run-time library routine for larger sets.

  Inside unpacked records, PACKED SET fields are no slower than unpacked SET fields. The same holds true for variables of PACKED SETs. PACKED SETs of size 32 or 64 bits are the best performing set types; otherwise a multiple of 8 bits improves performance to a lesser degree.

You may still need to use PACKED if you rely on the record for compatability with binary data files or when assuming that types like PACKED ARRAY OF BOOLEAN are implemented as bit strings.

While the compiler can position record fields at natural boundaries, it cannot minimize the alignment bytes that are required between fields. The calling standard requires the compiler to allocate record fields in the same lexical order that they appear in the source file. For example:

```
type t1 = record
          f1 : char;
          f2 : integer;
          f3 : char;
          f4 : integer;
          end;
```

The size of this record is 16 bytes:

- F1 is a byte field, followed by 3 padding bytes to position F2 at a longword boundary
- F2 is 4 bytes
- F3 is a single byte, followed by 3 more padding bytes to position F4 at a longword boundary
- F4 is 4 bytes

The optimal layout would be:

```
type t2 = record
          f1,f2 : integer;
          f3,f4 : char;
          end;
```

The size of this record is only 12 bytes:

- F1 and F2 are placed on adjacent longword boundaries
- F3 and F4 can immediately follow, since they can appear on any byte boundary, they in turn are followed by 2 padding bytes to round the size of the record up to a multiple of its natural alignment of longword.

To achieve the fewest alignment bytes, you should place larger fields at the beginning of the record and smaller fields at the end. If you have record fields of schema types that have run-time size, you should place those at the very end of the record, since their offset requires run-time computation.

You can get the optimal record layout by:

- Avoiding the PACKED keyword except for PACKED ARRAY OF CHARs and possibly PACKED SETs
- Avoiding explicit POS or ALIGNED attributes
- Placing larger fields before smaller fields
- Placing fixed-size fields before run-time sized fields

8.11.4 Optimal Data Size

Data items that are smaller than 32-bits may impose a performance penalty, due to the additional instructions required to access them. The compiler will attempt to reorder loads and stores that manip adjacent items smaller than 32-bits to minimize the number of memory references required.

For performance reasons, the compiler on Alpha systems will allocate Boolean and enumerated types as longwords in unpacked records or arrays. On VAX systems, you have to explicitly request this with a DCL qualifier or the corresponding attribute.

You should avoid any explicit size attributes on subrange types. While it is true that [BYTE] 0..255 is smaller than 0..255 (which would allocate 4 bytes, since it is a subrange of INTEGER), the additional overhead of accessing the byte-sized subrange might be than the extra 3 bytes of storage. Using the BIT attribute on subranges is even less effective in terms of the extra instructions required to manipulate a 13-bit integer subrange inside a record. Use these attributes only where needed.

8.11.5 Converting Existing Records

When moving code from an *OpenVMS VAX* system to an Alpha system, you probably want to make sure that you are getting the best performance from your Alpha system. To do that, you must use natural alignment on your record types.

8.11.6 Applications with No External Data Dependencies

If your application has no external data dependencies (such as no stored binary data files, no binary data transmitted to some external device), then the conversion is as simple as:

- Using the default natural alignment.
- Using the default enumeration size.
- Removing any uses of PACKED that are not needed.
- Removing any explicit positioning or size attributes that are not needed.
- Optionally reordering fields to place larger fields before smaller fields. This does not make the record faster, but does make it smaller.

Depending on your datatypes, the removal of any PACKED keywords or attributes may make little improvement in performance. For example, a PACKED ARRAY OF REAL is identical in size and performance to an unpacked ARRAY OF REAL.

The *Compaq Pascal* compiler has two features to help you identify poorly aligned records and how often they are used:

- The `-usage performance` command line option

  This option causes the compiler to generate messages for declarations and uses of record fields that are poorly aligned or poorly sized. For example:

  ```
  program a;

  type r = packed record
           f1 : boolean;
           f2 : integer;
           end;

  begin
  end.
  ```

  In this program the compiler can highlight the following:

  ```
  unix> pc -usage performance test.pas
  pascal: Info: test.pas, line 4: Component is not optimally size
           f1 : boolean;
  .........^
  pascal: Info: test.pas, line 5: Component is not optimally aligned
           f2 : integer;
  .........^
  pascal: Success: pascal completed with 2 diagnostics
  ```

  In this example, the size of the Boolean field in the PACKED ARRAY is only 1 bit. Single bit fields require additional instructions to process. The integer field is not aligned on a well-aligned boundary for the target system. The `-usage performance` option gives performance information customized to the target system. For example, on an *OpenVMS VAX* system, INTEGERs need only be aligned on a byte boundary for "good" performance; on an Alpha system, INTEGERs should be on a longword boundary.

- The `-show structure_layout` command line option.

  This option causes the compiler to generate a structure layout summary in the listing file.

  This summary gives size and offset information about variables, types, and fields. It also flags the same information as the `-usage performance` command line option.

  For example, compiling the above program with the following command produces the following in the listing file:

```
$ pc -V -show structure_layout test.pas

Comments      Offset        Size
-----------  -----------  -----------
                            5 Bytes     R {In PROGRAM A} = PACKED RECORD
Size            0 Bytes     1 Bit           F1 : BOOLEAN
Align           1 Bit       4 Bytes         F2 : INTEGER
                                            END
```

This output shows the size of the record "R" as well as the sizes and offsets of the records fields. It also highlights any components that were poorly sized or poorly aligned.

8.11.7 Applications with External Data Dependencies

If your application has external data dependencies, the process is more involved, since you have to isolate and understand the dependencies.

Possible steps when porting the code include:

- Using the `-align vax` option
- Using the `-enumeration_size` byte qualifier
- Leaving the code exactly as is

This should produce the same behavior on the Alpha system as you had on your *OpenVMS VAX* system, unless the external data contains floating point data.

You then have to identify which records in your program have external data dependencies. These include binary files (for example, FILE OF xxx), shared memory sections with other programs, and binary information passed to a library routine.

If the external data contains floating point data, you will have to explicity convert the data using the `cvt-ftof` library routine found for the math library.

You can immediately begin to convert records without external data dependencies into optimal format (for example, remove any unneeded PACKED keywords and attributes as described earlier).

You need to classify records with external dependencies further into:

- Records that cannot be naturally aligned due to a hard dependency that cannot be changed (like a record that maps onto an external piece of hardware, or a record that is passed to some software you cannot change).
- Records that can be changed after conversion of binary data or cooperating software.

Isolate records that you cannot change into their own environment file by using `-align vax`, `-enumeration_size byte`. You can also attach the ALIGN and ENUMERATION_SIZE attributes to the TYPE or VAR sections that define these records.

You do not need to isolate the record if it uses the PACKED keyword, since PACKED records are identical regardless of the `-align` or `-enumeration_size` options. Nevertheless, isolating the records with dependencies might useful in the future if you eventually intend to change the format.

For records that you might change, you need to decide whether it is worthwhile to convert the record and any external binary data. If the record is of low-use and you have a large quantity of external data, the cost of conversion is probably too high. If a record is of high-use but is mostly aligned, then the conversion also may not be worthwhile. However, a high-use record that is poorly aligned suggests conversion of external data regardless of the amount of effort required.

There are two types of poorly aligned records:

- Records that use the PACKED keyword

  PACKED records lay out the same with either setting of the `-align` or `-enumeration_size` options. To get natural alignment, you must remove the PACKED keyword. However, the keyword PACKED by itself does not guarantee poor alignment. For example:

  ```
  type t = packed record
          f1,f2 : integer;
          end;
  ```

  This record is well aligned with or without the PACKED keyword. It is also well aligned with `-align alpha_axp` and `-align vax`. You can remove the PACKED keyword for completeness, but nothing else needs to be done.

- Unpacked records that lay out differently with `-align alpha_axp` and `-align vax`

  These records automatically are well-aligned by the compiler when recompiled with `-align alpha_axp`. However, there are some unpacked records are already well-aligned with both alignment formats. For example:

  ```
  type t = record
          f1,f2 : integer;
          end;
  ```

  This unpacked record is well aligned with `-align alpha_axp` and `-align vax`. Nothing else needs to be done to this record.

The -usage performance and -show structure_layout options can be helpful for identifying poorly aligned records.

For PACKED keywords, you can compile with and without the PACKED keyword to see if the fields are positioned at the same offsets or not.

You have classified the records with external data dependencies into:

- Records that are well aligned with both alignment/enumeration formats
- Records that are poorly aligned, where conversion is not worthwhile
- Records that are poorly aligned, where conversion is worthwhile

For the well-aligned records, no additional work is needed now, but be aware that you still have an external data dependency that might cause problems if you add fields to the record in the future.

Isolate records that are not being converted into the same environment file or into the TYPE or VAR sections where you placed the records that you could not convert.

For records that are worth converting, you need to plan how to convert the external binary data or cooperating software. For cooperating software, you need to ensure that it gets modified so it views the record with the "natural" layout. You can determine the layout by using the -show structure_layout command line option described above. For binary data, you need to write a conversion program.

Converting existing binary data involves writing a program that reads the existing data into a poorly-aligned record, copies the data into a well aligned record, and then writes out the new record.

A simple conversion program would look like:

```
program convert_it(oldfile,newfile);

[align(vax),enumeration_size(byte)]
type oldtype = packed record
               { Existing record fields }
               end;

type newtype = record
               { Record fields reorganized for optimal alignment }
               end;
```

```
var oldfile = file of oldtype;
    newfile = file of newtype;
    oldvar : oldtype;
newvar : newtype;
begin
reset(oldfile);
rewrite(newfile);
while not eof(oldfile) do
   begin
   read(oldfile,oldvar);

   { For each field, sub-field, etc. move the data }
   newvar.field1 := oldvar.field1;
   newvar.field2 := oldvar.field2;

   write(newfile,newtype);
   end;
close(oldfile);
close(newfile);
end.
```

Notice the "type" keyword before the definition of the "newtype" type. Without this keyword, "newtype" would be in the same type definition part as "oldtype" and would be processed with the same ALIGN and ENUMERATION_SIZE settings.

9 Migrating from Pascal for RISC to Compaq Pascal

If you are migrating software that was developed specifically for Pascal for RISC on RISC/ULTRIX DECstations, there are differences in features on *Compaq Pascal* that must be considered. These differences include run-time sized types and variables, a true separate compilation mechanism, variable length strings, compiler attributes (also known as pragmas), and more. You should refer to the pertinent sections in the *Compaq Pascal* documentation set for information about specific issues, but significant differences are discussed in this chapter.

9.1 Pascal for RISC and Compaq Pascal Compile-Time Differences

Table 9–1 lists differences you can see when you take a Pascal program that was written for Pascal for RISC and compile it with *Compaq Pascal* for *Tru64 UNIX*.

Table 9–1 Pascal for RISC and Compaq Pascal Compile-Time Differences

Pascal for RISC	Compaq Pascal
Independent compilation	Independent compilation through environment files[1].
NULL statement	Not supported
Syntax for modifying the default alignment rules for record fields	Use POS and ALIGNED attributes[1].
`-apc` to change default integer size to 16 bits	Default integer size cannot be changed.

[1]See *Compaq Pascal Language Reference Manual* for more information.

(continued on next page)

Table 9–1 (Cont.) Pascal for RISC and Compaq Pascal Compile-Time Differences

Pascal for RISC	Compaq Pascal
Type conversion: ALFA, BOOLEAN, CHAR, CARDINAL, DOUBLE, INTEGER, INTEGER16, INTEGER32, and REAL	Not supported
Max length of identifiers = 32 bits	Max length of identifiers = 31 bits
Negative numbers allowed in SETs	Positive numbers only allowed in SETs
<= 512 elements in an INTEGER or CARDINAL set	<=255 elements in an INTEGER or CARDINAL set.
Value of enumerated types written in lower case.	Value of enumerated types written in upper case.
Type CHAR is subrange 0..127 when used as base-type.	Type CHAR is subrange 0..255 when used as base-type.

`-w`
`-stdansi`
`-stdiso`
	`-nowarn`
	`-std ansi`
	`-std iso`

`-casesense`	Not supported. Mixed-case names are provided on routine and variable names.
`-j`, `-EB`, `-EL`, `-H_c`, `-#`, `-W_c`, `-t`, `-B(string)`, `-h(path)`, `-ko (output)`, `-k`, `-S`, `-Olimit(num)`	Not supported

Further differences are listed as follows:

- Pascal for RISC has different default field widths than *Compaq Pascal*.
- Pascal for RISC and *Compaq Pascal* may flag different variables and routines as uninitialized and unused because different algorithms are used by the two compilers to ascertain this information.
- The Pascal for RISC definition and syntax for type-casting is different from the *Compaq Pascal* definition for type-casting. See the *Compaq Pascal Language Reference Manual* for more information in type-casting in *Compaq Pascal*.

9.2 Pascal for RISC and Compaq Pascal Run-Time Differences

There are behavioral and implementation differences between Pascal for RISC and *Compaq Pascal*. Table 9–2 lists the differences you can see at run time when running code compiled by *Compaq Pascal*.

Table 9–2 Pascal for RISC and Compaq Pascal Run-Time Differences

Pascal for RISC	Compaq Pascal
AND and OR are short-circuited	AND_THEN and OR_ELSE are short-circuited.
Determination of REAL / DOUBLE precision constants are based on their use.	By default, all floating-point constants are of type REAL. Mechanism is provided for specifying constants that are DOUBLE precision.[1]
"Non-zero" value for the ordinal value of TRUE	Stores a 1 in boolean variables for TRUE. Non-Pascal code passing boolean arguments to *Compaq Pascal* could find behavior differences if even numbers (2,4,6...) were placed in boolean variables to signify TRUE. Even values are treated as FALSE because the low order bit is not set.
STLIMIT	STLIMIT is recognized, but will not count executed statements or generate a run-time error when STLIMIT is called.
Trims trailing blanks from strings when printed with a field width of 0.	Follows the Pascal standard, which dictates that strings printed with a field width of 0 print no characters.
Output is left-justified when negative field widths are used.	Negative field widths are treated as runtime errors.

[1]See *Compaq Pascal Language Reference Manual* for more information.

A
Errors Returned by STATUS and STATUSV Functions

Table A–1 lists the error conditions detected by the STATUS and STATUSV functions, their symbolic names, and the corresponding values. The symbolic names and their values are defined in the file `/usr/include/passtatus.pas`, which you can include with a %INCLUDE directive in a CONST section of your program. To test for a specific condition, you compare the STATUS or STATUSV return values against the value of a symbolic name.

Note that the symbolic names correspond to some of the run-time errors listed in Appendix C; however, not all run-time errors can be detected by STATUS.

Table A–1 STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_ACCMETINC	5	Specified access method is not compatible with this file.
PAS$K_AMBVALENU	30	"String" is an ambiguous value for the enumerated type "type".
PAS$K_CURCOMUND	73	DELETE or UPDATE was attempted while the current component was undefined.
PAS$K_DELNOTALL	100	DELETE is not allowed for a file with sequential organization.
PAS$K_EOF	–1	File is at end-of-file.
PAS$K_ERRDURCLO	16	Error occurred while the file was being closed.
PAS$K_ERRDURDEL	101	Error occurred during execution of DELETE.
PAS$K_ERRDUREXT	127	Error occurred during execution of EXTEND.

(continued on next page)

Table A–1 (Cont.) STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_ERRDURFIN	102	Error occurred during execution of FIND or FINDK.
PAS$K_ERRDURGET	103	Error occurred during execution of GET.
PAS$K_ERRDUROPE	2	Error occurred during execution of OPEN.
PAS$K_ERRDURPRO	36	Error occurred during prompting.
PAS$K_ERRDURPUT	104	Error occurred during execution of PUT.
PAS$K_ERRDURRES	105	Error occurred during execution of RESET or RESETK.
PAS$K_ERRDURREW	106	Error occurred during execution of REWRITE.
PAS$K_ERRDURTRU	107	Error occurred during execution of TRUNCATE.
PAS$K_ERRDURUNL	108	Error occurred during execution of UNLOCK.
PAS$K_ERRDURUPD	109	Error occurred during execution of UPDATE.
PAS$K_ERRDURWRI	50	Error occurred during execution of WRITELN.
PAS$K_EXTNOTALL	128	EXTEND is not allowed for a shared file.
PAS$K_FAIGETLOC	74	GET failed to retrieve a locked component.
PAS$K_FILALRCLO	15	File is already closed.
PAS$K_FILALROPE	1	File is already open.
PAS$K_FILNAMREQ	14	File name must be specified in order to save, print, or submit an internal file.
PAS$K_FILNOTDIR	110	File is not open for direct access.
PAS$K_FILNOTFOU	3	File was not found.
PAS$K_FILNOTGEN	111	File is not in generation mode.
PAS$K_FILNOTINS	112	File is not in inspection mode.
PAS$K_FILNOTKEY	113	File is not open for keyed access.
PAS$K_FILNOTOPE	114	File is not open.
PAS$K_FILNOTSEQ	115	File does not have sequential organization.
PAS$K_FILNOTTEX	116	File is not a text file.
PAS$K_GENNOTALL	117	Generation mode is not allowed for a read-only file.

(continued on next page)

Table A–1 (Cont.) STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_GETAFTEOF	118	GET attempted after end-of-file has been reached.
PAS$K_INSNOTALL	119	Inspection mode is not allowed for a write-only file.
PAS$K_INSVIRMEM	120	Insufficient virtual memory.
PAS$K_INVARGPAS	121	Invalid argument passed to a *Compaq Pascal* Run-Time Library procedure.
PAS$K_INVFILSYN	4	Invalid syntax for file name.
PAS$K_INVKEYDEF	9	Key definition is invalid.
PAS$K_INVRECLEN	12	Record length nnn is invalid.
PAS$K_INVSYNBIN	37	“String” is invalid syntax for a binary value.
PAS$K_INVSYNENU	31	“String” is invalid syntax for a value of an enumerated type.
PAS$K_INVSYNHEX	38	“String” is invalid syntax for a hexadecimal value.
PAS$K_INVSYNINT	32	“String” is invalid syntax for an integer.
PAS$K_INVSYNOCT	39	“String” is invalid syntax for an octal value.
PAS$K_INVSYNREA	33	“String” is invalid syntax for a real number.
PAS$K_INVSYNUNS	34	“String” is invalid syntax for an unsigned integer.
PAS$K_KEYCHANOT	72	Changing the key field is not allowed.
PAS$K_KEYDEFINC	10	KEY(nnn) definition is inconsistent with this file.
PAS$K_KEYDUPNOT	71	Duplication of key field is not allowed.
PAS$K_KEYNOTDEF	11	KEY(nnn) is not defined in this file.
PAS$K_KEYVALINC	70	Key value is incompatible with file’s key nnn.
PAS$K_LINTOOLON	52	Line is too long; exceeds record length by nnn characters.
PAS$K_LINVALEXC	122	LINELIMIT value exceeded.
PAS$K_NEGWIDDIG	53	Negative value in width or digits (of a field width specification) is invalid.
PAS$K_NOTVALTYP	35	“String” is not a value of type “type”.

(continued on next page)

Table A–1 (Cont.) STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_ORGSPEINC	8	Specified organization is inconsistent with this file.
PAS$K_RECLENINC	6	Specified record length is inconsistent with this file.
PAS$K_RECTYPINC	7	Specified record type is inconsistent with this file.
PAS$K_RESNOTALL	124	RESET is not allowed for an internal file that has not been opened.
PAS$K_REWNOTALL	123	REWRITE is not allowed for a file opened for sharing.
PAS$K_SUCCESS	0	Last file operation completed successfully.
PAS$K_TEXREQSEQ	13	Text files must have sequential organization and sequential access.
PAS$K_TRUNOTALL	125	TRUNCATE is not allowed for a file opened for sharing.
PAS$K_UPDNOTALL	126	UPDATE is not allowed for a file that has sequential organization.
PAS$K_WRIINVENU	54	WRITE operation attempted on an invalid enumerated value

B
Entry Points to Compaq Pascal Run-Time Library

This appendix describes the entry point to utility routines in the *Tru64 UNIX* Run-Time Library that can be called as external routines by a *Compaq Pascal* program. These utilities allow you to access *Compaq Pascal* Run-Time Library features that are not directly provided by the language.

The following routine has been added to libpas.a/libpas.so to enable a program to determine the C file variable that is used to implement a Pascal file variable. The routine is:

```
function
pas$c_file_variable( var f : text ) : pointer; external;
```

This routine will return the address of the C file variable used to implement the specified Pascal file variable. The file must already be opened before the routine can be called.

C
Diagnostic Messages

This appendix summarizes the error messages that can be generated by a *Compaq Pascal* program at compile time and at run time.

C.1 Compiler Diagnostics

The *Compaq Pascal* compiler reports compile-time diagnostics in the source listing (if one is being generated) and summarizes them on the terminal (in interactive mode) or in the batch log file (in batch mode). Compile-time diagnostics have the following format:

```
pascal: Error: err.pas, line n: error-text
```

Where:

- Error indicates the category of the diagnostic message. There are four categories of compile-time diagnostic messages:
  - Information

    Indicates an informational message that flags extensions to the Pascal standard, identifies unused or possibly uninitialized variables, or provides additional information about a more severe error.
  - Warning

    Indicates a warning that flags an error or construct that may cause unexpected results, but that does not prevent the program from linking and executing.
  - Error

    Indicates an error that prevents generation of machine code; instead, the compiler produces an empty object module indicating that E-level messages were detected in the source program.
  - Severe (same as Fatal) Indicates a fatal error.
- err.pas indicates the filename that is being compiled.
- line n indicates the line number that the error occured on.

- error-text contains the text of the compiler diagnostic.

If the source program contains either E- or F-level messages, the errors must be corrected before the program can be linked and executed.

All diagnostic messages contain a brief explanation of the event that caused the error.

C.2 Diagnostic Messages

This section lists compile-time diagnostic messages in alphabetical order, including their severity codes and explanatory message text. Where the message text is not self-explanatory, additional explanation follows. Portions of the message text enclosed in quotation marks are items that the compiler substitutes with the name of a data object when it generates the message.

64BITBASTYP, 64-bit pointer base types cannot contain file variables or schema types

ERROR: File types and schema types may not be allocated in 64-bit S2 address space, because their implementation currently assumes 32-bit pointers in internal data structures.

64BITNOTALL, 64-bit pointers are not allowed in this context

ERROR: File types and schema types may not be allocated in 64-bit S2 address space, because their implementation currently assumes 32-bit pointers in internal data structures.

ABSALIGNCON, Absolute address / alignment conflict

Error: The address specified by the AT attribute does not have the number of low-order bits implied by the specified alignment attribute.

ACCMETHCON, Specified ACCESS_METHOD conflicts with file's record organization

Warning: You cannot specify ACCESS_METHOD:=DIRECT for a file that has indexed organization or sequential organization and variable-length records. You cannot specify ACCESS_METHOD:=KEYED for a file with sequential or relative organization.

ACTHASNOFRML, Actual parameter has no corresponding formal parameter

Error: The number of actual parameters specified in a routine call exceeds the number of formal parameters in the routine's declaration, and the last formal parameter does not have the LIST attribute.

ACTMULTPL, Actual parameter specified more than once

Error: Each formal parameter (except one with the LIST attribute) can have only one corresponding actual parameter.

ACTPASCNVTMP, Conversion: actual passed is resulting temporary
ACTPASRDTMP, Formal requires read access: actual parameter is resulting temporary
ACTPASSIZTMP, Size mismatch: actual passed is resulting temporary
ACTPASWRTMP, Formal requires write access: actual parameter is resulting temporary

Warning: A temporary variable is created if an actual parameter does not have the size, type, and accessibility properties required by the corresponding foreign formal parameter.

ACTPRMORD, Actual parameter must be ordinal

Error: The actual parameter that specifies the starting index of an array for the PACK or UNPACK procedure must have an ordinal type.

ADDIWRDALIGN, ADD_INTERLOCKED requires variable with at least word alignment
ADDIWRDSIZE, ADD_INTERLOCKED requires 16-bit variable

Error: These restrictions are imposed by the instruction sequence that is used on the target architecture.

ADDRESSVAR, "parameter name" is a VAR parameter, ADDRESS is illegal

Warning: You should not use the ADDRESS function on a nonvolatile variable or component or on a formal VAR parameter.

ADISCABSENT, Formal discriminant "discriminant name" has no corresponding actual discriminant

Error: An actual discriminant must be specified for every formal discriminant in a schema type definition.

ADISCHASNOFRML, Actual discriminant has no corresponding formal discriminant

Error: The number of actual discriminants specified is greater than the number of formal discriminants defined in the schema type definition.

AGGNOTALL, Aggregate variable access of this type not allowed, must be indexed

Error.

ALIATRTYPCON, Alignment attribute / type conflict

ALIGNAUTO, Alignment greater than n conflicts with automatic allocation

Error: The value n has the value 3 on *OpenVMS Alpha* systems or 2 on *OpenVMS VAX* systems; *OpenVMS Alpha* hardware aligns the stack on a quadword boundary and *OpenVMS VAX* hardware aligns it on a longword boundary. You cannot specify a greater alignment for automatically allocated variables.

ALIDOWN, Alignment down-graded from default of ALIGNED(n)

Info: The value of n is based on the size of the object that is being downgraded.

ALIGNFNCRES, Alignment greater than n not allowed on function result

Error: The value n has the value 3 on *OpenVMS Alpha* systems or 2 on *OpenVMS VAX* systems. The use of an attribute on a routine conflicts with the requirements of the object's type.

ALIGNINT, ALIGNED expression must be integer value in range 0..n; defaulting to m

Error: The value n has the value of the largest argument to the ALIGNED attribute allowed on the platform.

ALIGNVALPRM, Alignment greater than n not allowed on value parameter

Error: The value n has the value 3 on *OpenVMS Alpha* systems or 2 on *OpenVMS VAX* systems. The use of an attribute on a parameter conflicts with the requirements of the object's type.

ALLPRMSAM, All parameters to 'MIN' or 'MAX' must have the same type

Error.

APARMACTDEF, Anonymous parameter "parameter number" has neither actual nor default

Error: If the declaration of a routine failed to specify a name for a formal parameter, a call to the routine will result in this error message. The routine declaration will also cause an error to be reported.

ARITHOPNDREQ, Arithmetic operand(s) required

Error.

ARRCNTPCK, Array cannot be PACKED

Error: At least one parameter to the PACK or UNPACK procedure must be unpacked.

ARRHAVSIZ, “routine name” requires that ARRAY component have compile-time known size

Error: You cannot use the PACK and UNPACK procedures to pack or unpack one multidimensional conformant array into another. The component type of the dimension being copied must have a compile-time known size; that is, it must have some type other than a conformant schema.

ARRMSTPCK, Array must be PACKED

Error: At least one parameter to the PACK or UNPACK procedure must be of type PACKED.

ARRNOTSTR, Array type is not a string type

Error: You cannot write a value to a text file (using WRITE or WRITELN) or to a VARYING string (using WRITEV) if there is no textual representation for the type. Similarly, you cannot read a value from a text file (using READ or READLN) or from a VARYING string (using READV) if there is no textual representation for the type. The only legal array, therefore, is PACKED ARRAY [1..n] OF CHAR.

ASYREQASY, ASYNCHRONOUS “calling routine” requires that “called routine” also be ASYNCHRONOUS

Warning.

ASYREQVOL, ASYNCHRONOUS “routine name” requires that “variable name” be VOLATILE

Warning: A variable referred to in a nested asynchronous routine must have the VOLATILE attribute.

ATINTUNS, AT address must be an integer value

Error.

ATREXTERN, “attribute name” attribute allowed only on external routines

Error: The LIST and CLASS_S attributes can be specified only with the declarations of external routines.

ATTRCONCMDLNE, Attribute contradicts command line qualifier

Error: The double-precision attribute specified contradicts the /FLOAT, /G_FLOATING, or /NOG_FLOATING qualifier specified on the compile command line.

ATTRCONFLICT, Attribute conflict: “attribute name”

Information: This message can appear as additional information on other error messages.

ATTRONTYP, Descriptor class attribute not allowed on this type

Error: The use of the descriptor class attribute on the variable, parameter, or routine conflicts with the requirements of the object’s type.

AUTOGTRMAXINT, Allocation of “variable name” causes automatic storage to exceed MAXINT bits

Error: The *Compaq Pascal* implementation restricts automatic storage to a size of 2,147,483,647 bits.

AUTOMAX, Unable to quadword align automatic variables, using long alignment

Info.

BADANAORG, Analysis data file “file name” is not on a random access device

Fatal.

BADENVORG, Environment file “file name” is not on a random access device

Fatal.

BADSETCMP, < and > not permitted in set comparisons

Error.

BINOCTHEX, Expecting BIN, OCT, or HEX

Error: You must supply BIN, OCT, or HEX as a variable modifier when reading the variable on a nondecimal basis.

BLKNOTFND, “routine” block “routine name” declared FORWARD in “block name” is missing

Error.

BLKTOODEEP, Routine blocks nested too deeply

Error: You cannot nest more than 31 routine blocks.

BNDACTDIFF, Actual's array bounds differ from those of other parameters in same section

Error: All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible.

BNDCNFRUN, Bounds of conformant ARRAY "array name" not known until run-time

Error: You cannot use the UPPER and LOWER functions on a dynamic array parameter in a compile-time constant expression.

BNDSUBORD, Bound expressions in a subrange type must be ordinal

Error: The expressions that designate the upper and lower limits of a subrange must be of an ordinal type.

BOOLOPREQ, BOOLEAN operand(s) required

Error: The operation being performed requires operands of type BOOLEAN. Such operations include the AND, OR, and NOT operators and the SET_INTERLOCKED and CLEAR_INTERLOCKED functions.

BOOSETREQ, BOOLEAN or SET operand(s) required

Error.

BYTEALIGN, Type larger than 32 bits can be positioned only on a byte boundary

Error: See the *Compaq Pascal Language Reference Manual* for information on the types that are allocated more than 32 bits.

CALLFUNC, Function "function name" called as procedure, function value discarded

Warning.

CARCONMNGLS, CARRIAGE_CONTROL parameter is meaningless given file's type

Warning: The carriage-control parameter is usually meaningful only for files of type TEXT and VARYING OF CHAR.

CASLABEXPR, Case label and case selector expressions are not compatible

Error: All case labels in a CASE statement must be compatible with the expression specified as the case selector.

CASORDRELPTR, Compile-time cast allowed only between ordinal, real, and pointer types

CASSELORD, Case selector expression must be an ordinal type

Error.

CASSRCSIZ, Source type of a cast must have a size known at compile-time

CASTARSIZ, Target type of a cast must have a size known at compile-time

Error: A variable being cast by the type cast operator cannot be a conformant array or a conformant VARYING parameter. An expression being cast cannot be a conformant array parameter, a conformant VARYING parameter, or a VARYING OF CHAR expression. The target type of the cast cannot be VARYING OF CHAR.

CDDABORT, %DICTIONARY processing of CDD record definition aborted

Error: The *Compaq Pascal* compiler is unable to process the CDD record description. See the accompanying CDD messages for more information.

CDDBADDIR, %DICTIONARY directive not allowed in deepest %INCLUDE, ignored

Error: A program cannot use the %DICTIONARY directive in the fifth nested %INCLUDE level. The compiler ignores all %DICTIONARY directives in the fifth nested %INCLUDE level.

CDDBADPTR, invalid pointer was specified in CDD record description

Warning: The CDD pointer data type refers to a CDD path name that cannot be extracted, and is replaced by ^INTEGER.

CDDBIT, Ignoring bit field in CDD record description

Information: The *Compaq Pascal* compiler cannot translate a CDD bit data type that is not aligned on a byte boundary and whose size is greater than 32 bits.

CDDBLNKZERO, Ignoring blank when zero attribute specified in CDD record description

Information: The *Compaq Pascal* compiler does not support the CDD BLANK WHEN ZERO clause.

CDDCOLMAJOR, CDD description specifies a column-major array

Error: The *Compaq Pascal* compiler supports only row-major arrays. Change the CDD description to specify a row-major array.

CDDDEPITEM, Ignoring depends item attribute specified in CDD record description

Information: The *Compaq Pascal* compiler does not support the CDD DEPENDING ON ITEM attribute.

CDDDFLOAT, D_floating CDD datatype was specified when compiling with G_FLOATING

Warning: The CDD record description contains a D_floating data type while compiling with G_floating enabled. It is replaced with [BYTE(8)] RECORD END.

CDDFLDVAR, CDD record description contains field(s) after CDD variant clause

Error: The CDD record description contains fields after the CDD variant clause. Because *Compaq Pascal* translates a CDD variant clause into a Pascal variant clause, and a Pascal variant clause must be the last field in a record type definition, the fields following the CDD variant clause are illegal.

CDDGFLOAT, G_floating CDD datatype was specified when compiling with NOG_FLOATING

Warning: The CDD record description contains a G_floating data type while compiling with D_floating enabled. It is replaced with [BYTE(8)] RECORD END.

CDDILLARR, Aligned array elements can not be represented, replacing with [BIT(n)] RECORD END

Information: The *Compaq Pascal* compiler does not support CDD record descriptions that specify an array whose array elements are aligned on a boundary greater than the size needed to represent the data type. It is replaced with [BIT(n)] RECORD END, where n is the appropriate length in bits.

CDDINITVAL, Ignoring specified initial value specified in CDD record description

Information: The *Compaq Pascal* compiler does not support the CDD INITIAL VALUE clause.

CDDMINOCC, Ignoring minimum occurs attribute specified in CDD record description

Information: The *Compaq Pascal* compiler does not support the CDD MINIMUM OCCURS attribute.

CDDONLYTYP, %DICTIONARY may only appear in a TYPE definition part

Error: The %DICTIONARY directive is allowed only in the TYPE section of a program.

CDDRGHTJUST, Ignoring right justified attribute specified in CDD record description

Information: The *Compaq Pascal* compiler does not support the CDD JUSTIFIED RIGHT clause.

CDDSCALE, Ignoring scaled attribute specified in CDD record description

Information: The *Compaq Pascal* compiler does not support the CDD scaled data types.

CDDSRCTYPE, Ignoring source type attribute specified in CDD record description

Information: The *Compaq Pascal* compiler does not support the CDD source type attribute.

CDDTAGDEEP, CDD description nested variants too deep

Error: A CDD record description may not include more than 15 levels of CDD variants. The compiler ignores variants beyond the fifteenth level.

CDDTAGVAR, Ignoring tag variable and any tag values specified in CDD record description

Information: The *Compaq Pascal* compiler does not fully support the CDD VARIANTS OF field description statement. The specified tag variable and any tag values are ignored.

CDDTOODEEP, CDD description nested too deep

Error: Attributes for the CDD record description exceed the implementation's limit for record complexity. Modify the CDD description to reduce the level of nesting in the record description.

CDDTRUNCREF, Reference string which exceeds 255 characters has been truncated

Information: The *Compaq Pascal* compiler does not support reference strings greater than 255 characters.

CDDUNSTYP, Unsupported CDD datatype "standard data type name"

Information: The CDD record description for an item has attempted to use a data type that is not supported by *Compaq Pascal*. The *Compaq Pascal* compiler makes the data type accessible by declaring it as [BYTE(n)] RECORD END where n is the appropriate length in bytes. Change the data type to one that is supported by *Compaq Pascal* or manipulate the contents of the field by passing it to external routines as variables or by using the *Compaq Pascal* type casting capabilities to perform an assignment.

CLSCNFVAL, CLASS_S is only valid with conformant strings

Error: When the CLASS_S attribute is used in the declaration of an internal routine, it can be used only on a conformant PACKED ARRAY OF CHAR. The conformant variable must also be passed by value semantics.

CLSNOTALLW, "descriptor class name" not allowed on a parameter of this type

Error: Descriptor class attributes are not allowed on formal parameters defined with either an immediate or a reference passing mechanism.

CMTBEFEOF, Comment not terminated before end of input

Error.

CNFCANTCNF, Component of PACKED conformant parameter cannot be conformant

Error.

CNFREQNCA, Conformants of this parameter type require CLASS_NCA

Error: The conformant parameter cannot be described with the default CLASS_A descriptor. Add the CLASS_NCA attribute to the parameter declaration.

CNSTRNOTALL, Nonstandard constructors are not allowed on nonstatic types

Error: You can write constructors for nonstatic types using the standard style of constructor.

CNSTRONZERO, Record constructors only allow OTHERWISE ZERO

Error.

CNTBEARRCMP, Not allowed on an array component
CNTBEARRIDX, Not allowed on an array index
CNTBECAST, Not allowed on a cast
CNTBECNFCMP, Not allowed on a conformant array component
CNTBECNFIDX, Not allowed on a conformant array index
CNTBECNFVRY, Not allowed on a conformant varying component
CNTBECOMP, Not allowed on a compilation unit
CNTBECONST, Not allowed on a CONST definition part

CNTBEDEFDECL, Not allowed on any definition or declaration part
CNTBEDESPARM, Not allowed on a %DESCR foreign mechanism parameter
CNTBEEXESEC, Not allowed on an executable section
CNTBEFILCMP, Not allowed on a file component
CNTBEFORMAL, Not allowed on a formal discriminant
CNTBEFUNC, Not allowed on a function result
CNTBEIMMPARM, Not allowed on a parameter passed by an immediate passing mechanism
CNTBELABEL, Not allowed on a LABEL declaration part
CNTBEPCKCNF, Not allowed on a PACKED conformant array component

CNTBEPTRBAS, Not allowed on a pointer base
CNTBERECFLD, Not allowed on a record field
CNTBEREFPARM, Not allowed on a parameter passed by a reference passing mechanism
CNTBERTNDECL, Not allowed on a routine declaration
CNTBERTNPARM, Not allowed on a routine parameter
CNTBESCHEMA, Not allowed on a nonstatic type
CNTBESETRNG, Not allowed on a set range
CNTBESTDPARM, Not allowed on a %STDESCR foreign mechanism parameter
CNTBETAGFLD, Not allowed on a variant tag field

CNTBETAGTYP, Not allowed on a variant tag type
CNTBETO, Not allowed on TO BEGIN/END DO
CNTBETYPDEF, Not allowed on a type definition
CNTBETYPE, Not allowed on a TYPE definition part
CNTBEVALPARM, Not allowed on a value parameter
CNTBEVALUE, Not allowed on a VALUE initialization part
CNTBEVALVAR, Not allowed on a VALUE variable
CNTBEVAR, Not allowed on a VAR declaration part
CNTBEVARBLE, Not allowed on a variable

CNTBEVARPARM, Not allowed on a VAR parameter
CNTBEVRYCMP, Not allowed on a varying component

Information: These messages can appear as additional information on other error messages.

COMCONFLICT, COMMON “block name” conflicts with another COMMON or PSECT of same name

Error: You can allocate only one variable in a particular common block, and the name of the common block cannot be the same as the names of other common blocks or program sections used by your program.

COMNOTALN, Component is not optimally aligned

Info.

COMNOTSIZ, Component is not optimally sized

Info.

COMNOTALNSIZ, Component is not optimally aligned and sized

Info.

COMNOTPOS, Fixed size field positioned after a run-time sized field is not optimal

Info.

CONTXTIGN, Text following constant definition ignored

Warning: When defining constants with the /CONSTANT DCL qualifier, any text appearing after a valid constant definition is ignored.

CPPFILERR, Preprocessor record: error oprning specified file

Error.

CRETIMMOD, Creation time for module “module name” in environment “environment file name” differs from creation time in previous environments

Warning: Two or more PEN files referred to a module, but the PEN files did not agree on the creation date/time for the module. This can occur if you recompile a module but do not recompile all the modules that inherited its PEN file.

CSTRBADTYP, Constructor: only ARRAY, RECORD, or SET type
CSTRCOMISS, Constructor: component(s) missing
CSTRNOVRNT, Constructor: no matching variant
CSTRREFAARR, Repetition factor allowed only in ARRAY constructors
CSTRREFAINT, Repetition factor must be integer
CSTRREFALRG, Repetition factor too large
CSTRREFANEG, Repetition factor cannot be negative
CSTRTOOMANY, Constructor: too many components

Error: You can write constructors only for data items of an ARRAY type. You must specify one and only one value in the constructor for each component of the type. In an array constructor, you cannot use a negative integer value as a repetition factor to specify values for consecutive components.

CSTRREFAINT, Repetition factor must be an integer

Error.

CTESTRSIZ, Compile-time strings must be less than 8192 characters

Error.

CTGARRDESC, Contiguous array descriptor cannot describe size/alignment properties

Information: Conformant array parameters, dynamic array parameters, and %DESCR array parameters all use the contiguous array descriptor mechanism in the *Tru64 UNIX* Calling Standard. Size and alignment attributes are prohibited on such arrays, as these attributes can create noncontiguous allocation. This message can appear as additional information in other error messages.

DEBUGOPT, /-o0 is recommended with /-g

Information: Unexpected results may be seen when debugging an optimized program. To prevent conflicts between optimization and debugging, you should compile your program with /-o0 until it is thoroughly debugged. Then you can recompile the program with optimization enabled to produce more efficient code.

DECLORDER, Declarations are out of order

Error: The TO BEGIN DO and TO END DO declarations in a module must appear at the end of the module and may not be reordered.

DEFRTNPARM, Default parameter syntax not allowed on routine parameters
DEFVARPARM, Default parameter syntax not allowed on VAR parameters

Error.

DESCOMABORT, Further processing of /DESIGN=COMMENTS has been aborted

Error: An error has occurred that prohibits further comment processing.

DESCOMERR, An error has occurred while processing design information

Error.

DESCOMSEVERR, An internal error has occurred while processing /DESIGN=COMMENTS - please submit a problem report

Error: A fatal error has occurred during comment processing. Please submit a problem report including sufficient information to reproduce the program, including the version numbers of the DEC Language-Sensitive Editor/Source Code Analyzer and the *Compaq Pascal* compiler.

DESCTYPCON, Descriptor class / type conflict

Error: The descriptor class for parameter passing conflicts with the parameter's type. Refer to Section 6.3.3 of the *Compaq Pascal User Manual for OpenVMS Systems* for legal descriptor class/type combinations.

DESIGNTOOOLD, The comment processing routines are too old for the compiler

Error: The support routines for the /DESIGN=COMMENT qualifier are obsolete. Contact your system manager.

DIRCONVISIB, Directive contradicts visibility attribute

Error: The EXTERN, EXTERNAL, and FORTRAN directives conflict directly with the LOCAL and GLOBAL attributes.

DIREXPECT, No matching directive for the %IF directive

Error:: A %IF directive must contain a %THEN clause and be terminated by %ENDIF.

DIRUNEXP

Error:: Conditional compilation directives other than %IF are only valid after the parts of a %IF directive.

DISCLIMIT, Limit of 255 discriminants exceeded

Error.

DISNOTORD, Discriminant type must be an ordinal type

Error: The formal discriminant in a schema type definition must be an ordinal type.

DONTPACKVAR, "routine name" is illegal, variable can never appear in a packed context

Error: You cannot call the BITSIZE and BITNEXT functions for conformant parameters.

DUPLALIGN, Alignment already specified
DUPLALLOC, Allocation already specified
DUPLATTR, Attribute already specified
DUPLCLASS, Descriptor class already specified
DUPLDOUBLE, Double precision already specified

Error: Only one member of a particular attribute class can appear in the same attribute list.

DUPLFIN, TO END DO already specified
DUPLINIT, TO BEGIN DO already specified

Error: Only one TO BEGIN DO and one TO END DO section can appear in the same module.

DUPLGBLNAM, Duplicated global name

Warning: The GLOBAL attribute cannot appear on more than one variable or routine with the same name.

DUPLMECH, Passing mechanism already specified
DUPLOPT, Optimization already specified
DUPLSIZE, Size already specified
DUPLVISIB, Visibility already specified

Error: Only one member of a particular attribute class can appear in the same attribute list.

DUPTYPALI, Alignment already specified by type identifier “type name”
DUPTYPALL, Allocation already specified by type identifier “type name”
DUPTYPATR, Attribute already specified by type identifier “type name”
DUPTYPDES, Descriptor class already specified by type identifier “type name”
DUPTYPSIZ, Size already specified by type identifier “type name”
DUPTYPVIS, Visibility already specified by the type identifier “type name”

Error: An attribute specified for an object was already specified in the definition of the object’s type.

ELEOUTRNG, Element out of range

Error: A value specified in a set constructor used as a compile-time constant expression does not fall within the subrange defined as the set’s base type.

EMPTYCASE, Empty case body

Error: You failed to specify any case labels and corresponding statements in the body of a CASE statement.

ENVERROR, Environment resulted from a compilation with Errors

Error: When a program inherits an environment file that compiled with errors, unexpected results may occur during the program’s compilation. The environment file inherited by the program compiled with errors. Unexpected results may occur in the program now being compiled.

ENVFATAL, Environment resulted from a compilation with Fatal Errors

Error: The environment file inherited by the program compiled with fatal errors. Unexpected results may occur in the program now being compiled.

ENVOLDVER, Environment was created by a *Compaq Pascal* compiler, please recompile

Warning:

ENVWARN, Environment resulted from a compilation with Warnings

Warning: The environment file inherited by the program compiled with warnings. Unexpected results may occur in the program now being compiled.

ENVWRGCMP, Environment identifier was compiled by a *Compaq Pascal* for platform compiler

Fatal.

ERREALCNST, Error in real constant: digit expected

Error.

ERRNONPOS, ERROR parameter can be specified only with nonpositional syntax

Error.

ERROR, %ERROR

Error: This message is generated by the %ERROR directive.

ERRORLIMIT, Error Limit = “current error limit”, source analysis terminated

Fatal: The error limit specified for the program’s compilation was exceeded; the compiler was unable to continue processing the program. By default, the error limit is set at 30, but you can use the error limit switch at compile time to change it.

ESTBASYNCH, ESTABLISH requires that “routine name” be ASYNCHRONOUS

Warning.

EXPLCONVREQ, Explicit conversion to lower type required

Error: An expression of a higher-ranked type cannot be assigned to a variable of a lower-ranked type; you must first convert the higher-ranked expression by using DBLE, SNGL, TRUNC, ROUND, UTRUNC, or UROUND, as appropriate.

EXPNOTRES, Expression does not contribute to result

Information: The optimizer has determined that part of the expression does not affect the result of the expression and it will not evaluate that part of the expression.

EXPR2ONVAL, Expression is allowed only on real, integer, or unsigned values

Error: The second expression (and preceding colon) are allowed only if the value being written is of a real, integer, or unsigned type.

EXPRARITH, Expression must be arithmetic

Error: An expression whose type is not arithmetic cannot be assigned to a variable of a real type.

EXPRARRIDX, Expression is incompatible with unpacked array's index type

Error: The index type of the unpacked array is not compatible with the index type of either the PACK or UNPACK procedure it was passed to.

EXPRCOMTAG, Expression is not compatible with tag type

Error: A case label specified for a NEW, DISPOSE, or SIZE routine must be assignment compatible with the tag type of the variant.

EXPRINFUNC, Expression allowed only in FUNCTION

Error.

EXPRNOTSET, Expression is not a SET type

Error: The compiler encountered an expression of some type other than SET in a call to the CARD function.

EXTRNALLOC, Allocation attribute conflicts with EXTERNAL visibility

Error: The storage for an external variable or routine is not allocated by the current compilation; therefore, the specification of an allocation attribute is meaningless.

EXTRNAMDIFF, External names are different

Information: This message can appear as additional information on other error messages.

EXTRNCFLCT, "PSECT or FORWARD" conflicts with EXTERNAL visibility

Error: The storage for an external variable or routine is not allocated by the current compilation; therefore, the specification of an allocation attribute is meaningless.

FILEVALASS, FILE evaluation / assignment is not allowed

Error: You cannot attempt to evaluate a file variable or assign values to it.

FILHASSCH, FILE component may not contain nonstatic types or discriminant identifiers

Error: *Compaq Pascal* restricts components of files to those with compile-time size.

FILOPNDREQ, FILE operand required

Error: The EOF, EOLN, and UFB functions require parameters of file types.

FILVARFIL, FILE_VARIABLE parameter must be of a FILE type

Error: The file variable parameter to the OPEN and CLOSE procedures must denote a file variable.

FLDIVPOS, Field "field name" is illegally positioned

Error: A POS attribute attempted to position a record field before the end of the previous field in the declaration.

FLDNOTKNOWN, Unknown record field

Error.

FLDONLYTXT, Field width allowed only when writing to a TEXT file

Error.

FLDRADINT, Field width or radix expression must be of type INTEGER

Error: The field-width or radix expression in a WRITE, WRITELN, or WRITEV routine must be of type INTEGER.

FORACTORD, FOR loop control variable must be of an ordinal type
FORACTVAR, FOR loop control must be a true variable

Error: The control variable of a FOR statement must be a simple variable of an ordinal type and must be declared in a VAR section. For example, it cannot be a field in a record that was specified by a WITH statement, or a function identifier.

FLDWDTHINT, Field-width expression must be of type integer

Error.

FORCTLVAR, "variable name" is a FOR control variable

Warning: The control variable of a FOR statement cannot be assigned a value; used as a parameter to the ADDRESS function; passed as a writable VAR, %REF, %DESCR, or %STDESCR parameter; used as the control variable of a nested FOR statement; or written into by a READ, READLN, or READV procedure.

FORINEXPR, Expression is incompatible with FOR loop control variable

Error: The type of the initial or final value specified in a FOR statement is variable.

FRMLPRMDESC, Formal parameters use different descriptor formats
FRMLPRMINCMP, Formal routine parameters are not compatible
FRMLPRMNAM, Formal parameters have different names
FRMLPRMSIZ, Formal parameters have different size attributes
FRMLPRMTYP, Formal parameters have different types

Information: These messages can appear as additional information on other error messages.

FRSTPRMSTR, READV requires first parameter to be a string expression

Error: You must specify at least two parameters for the READV procedure—a character-string expression and a variable into which new values will be read.

FRSTPRMVARY, WRITEV requires first parameter to be a variable of type VARYING

Error.

FTRNOTHER, Feature not supported io this context

Error.

FTRNOTPOR, Feature not supported on platform(s)

Info.

FTRNOTSUP, Feature not supported on this platform

Error.

FUNCTRESTYP, Routine must be declared as FUNCTION to specify a result type

Error: You cannot specify a result type on a PROCEDURE declaration.

FUNRESTYP, Function result types are different

Information: This message can appear as additional information on other error messages.

FWDREPATRLST, Declared FORWARD; repetition of attribute list not allowed
FWDREPPRMLST, Declared FORWARD; repetition of formal parameter list not allowed
FWDREPRESTYP, Declared FORWARD; repetition of result type not allowed

Error: If the heading of a routine has the FORWARD directive, the declaration of the routine body cannot repeat the formal parameter list, the result type (applies only if the routine is a function), or any attribute lists that appeared in the heading.

FWDWASFUNC, FORWARD declaration was FUNCTION
FWDWASPROC, FORWARD declaration was PROCEDURE

Error.

GOTONOTALL, GOTO not allowed to jump into a structured statement

Warning.

GOTSZOVFL, GOT table overflow for module "name"

Error: The GOT (Global Offset Table) for the module is too large. Break up the module into multiple modules.

GTR32BITS, "routine name" cannot accept parameters larger than 32 bits

Error: DEC and UDEC cannot translate objects larger than 32 bits into their textual equivalent.

HIDATOUTER, HIDDEN legal only on definitions and declarations at outermost level

Error: When an environment file is being generated, it is possible to prevent information concerning a declaration from being included in the environment file by using the HIDDEN attribute. However, because an environment file consists only of declarations and definitions at the outermost level of a compilation unit, the HIDDEN attribute is legal only on these definitions and declarations.

IDENTGTR31, Identifier longer than 31 characters exceeds capacity of compiler

Warning.

IDESTRMOD, IDENT string for module identifier in environment identifier differs from IDENT string in previous environments

Warning.

IDNOTLAB, Identifier “symbol name” not declared as a label

Error.

IDXNOTCOMPAT, Index type is not compatible with declaration

Error: The type of an index expression is not assignment compatible with the index type specified in the array’s type definition.

IDXREQDKEY, Creating INDEXED organization requires dense keys

Warning: When you specify ORGANIZATION:=INDEXED when opening a file with HISTORY := NEW or UNKNOWN, the file’s alternate keys must be dense; that is, you may not omit any key numbers in the range from 0 through the highest key number specified for the file’s component type.

IDXREQKEY0, Creating INDEXED organization requires FILE OF RECORD with at least KEY(0)

Warning: When you specify ORGANIZATION:=INDEXED when opening a file with HISTORY := NEW or UNKNOWN, the file’s component type must be a record for which a primary key, designated by the [KEY(0)] attribute, is defined.

ILLINISCH, Illegal initialization of variable of nonstatic type

Error: Nonstatic variables, such as those created from schema types, cannot be initialized in the VALUE declaration part. To initialize these variables, you must use the initial state feature.

IMMEDBNDROU, Immediate passing mechanism may not be used on bound routine “routine name”

Warning: You cannot prefix a formal or an actual routine parameter with the immediate passing mechanism unless the routine was declared with the UNBOUND attribute.

IMMEDUNBND, Routines passed by immediate passing mechanism must be UNBOUND

Warning: A formal routine parameter that has the immediate passing mechanism must also have the UNBOUND attribute.

IMMGTR32, Immediate passing mechanism not allowed on values larger than 32 bits

Error: See the *Compaq Pascal Language Reference Manual* for more information on the types that are allocated more than 32 bits.

IMMHAVSIZ, Type passed by immediate passing mechanism must have compile-time known size

Error: You cannot specify an immediate passing mechanism for a conformant parameter or a formal parameter of type VARYING OF CHAR.

INCMPBASE, Incompatible with SET base type

Error: If no type identifier denotes the base type of a set constructor, the first element of the constructor determines the base type. The type of all subsequent elements specified in the constructor must be compatible with the type of the first.

INCMPFLDS, Record fields are not the same type

Error.

INCMPOPND, Incompatible operand(s)

Error: The types of one or more operands in an expression are not compatible with the operation being performed.

INCMPPARM, Incompatible "routine" parameter

Error: An actual routine parameter is incompatible with the corresponding formal parameter.

INCMPTAGTYP, Incompatible variant tag types

Error: This message can appear as additional information on other error messages.

INCTOODEEP, %INCLUDE directives nested too deeply, ignored

Error: A program cannot include more than five levels of files with the %INCLUDE directive. The compiler ignores %INCLUDE files beyond the fifth level.

INDNOTORD, Index type must be an ordinal type

Error: The index type of an array must be an ordinal type.

INFO, %INFO

Information: This message is generated by the %INFO directive.

INITNOEXT, INITIALIZE routine may not be EXTERNAL
INITNOFRML, INITIALIZE routine must have no formal parameter list

Error.

INITSYNVAR, Illegal initialization syntax—Use VALUE

Error.

INPNOTDECL, INPUT not declared in heading

Error: A call to EOF, EOLN, READLN, or WRITELN did not specify a file variable, and the default INPUT or OUTPUT was not listed in the program heading.

INSTNEWLSE, Please install a new version of LSE

Error: The version of the DEC Language-Sensitive Editor/Source Code Analyzer on your system is too old for the compiler. Contact your system manager.

INVCASERNG, Invalid range in case label list

Error.

INVEVAL, Array or Record evaluation not allowed

Error.

INVQUAVAL, Value for optimizer switch is out of range. Default value will be used.

Warning.

IVATTOPT, Unrecognized option for attribute

Explanation: Explanation: You attempted to specify an invalid option for one of the following attributes:

- CHECK (Warning)
- FLOAT (Warning)
- KEY (Error)
- OPTIMIZE (Warning)

IVATTR, Unrecognized attribute

Error.

IVAUTOMOD, AUTOMATIC variable is illegal at the outermost level of a MODULE

Error: You cannot specify the AUTOMATIC attribute for a variable declared at module level.

IVCHKOPT, Unrecognized CHECK option

Warning.

IVCOMBFLOAT, Illegal combination of D_floating and G_floating

Error: You cannot combine D_floating and G_floating numbers in a binary operation.

IVDIRECTIVE, Unrecognized directive

Error: The directive following a procedure or function heading is not one of those recognized by the *Compaq Pascal* compiler.

IVENVIRON, Environment "environment name" has illegal format, source analysis terminated

Fatal: The environment file inherited by the program has an illegal format; compilation is immediately aborted. However, a listing will still be produced if one was being generated.

IVFUNC, Invalid use of function "function name"
IVFUNCALL, Invalid use of function call
IVFUNCID, Invalid use of function identifier

Error: These messages result from illegal attempts to assign values or otherwise refer to the components of the function result (if its type is structured), use the type cast operator on a function identifier or its result, or deallocate the storage reserved for the function result (if its type is a pointer).

IVKEYOPT, Unrecognized KEY option

Error.

IVKEYVAL, FINDK KEY_VALUE cannot be an array (other than PACKED ARRAY [1..n] OF CHAR)

Error.

IVKEYWORD, Missing or unrecognized keyword

Error: The compiler failed to find an identifier where it expected one in a call to the OPEN or CLOSE procedure, or it found an identifier that was not legal in this position in the parameter list.

IVMATCHTYP, Invalid MATCH_TYPE parameter to FINDK

Error.

IVOPTMOPT, Unrecognized OPTIMIZE option

Warning:

IVOTHVRNT, Illegal use of OTHERWISE within CASE variant

Error: The *Compaq Pascal* extension of using OTHERWISE in a record constructor is only defined at the outer level of a record.

IVQUALFILE, Illegal switch “switch name” on file specification

Warning: Only the /LIST and /NOLIST qualifiers are allowed on the file specification of a %INCLUDE directive.

IVQUOCHAR, Illegal nonprinting character (ASCII “nnn”) within quotes

Warning: The only nonprinting characters allowed in a quoted string are the space and tab; the use of other nonprinting characters in a string causes this warning. To include nonprinting characters in a string, you should use the extended string syntax described in the *Compaq Pascal Language Reference Manual*.

IVRADIX, Invalid radix was specified in the extended number

Error.

IVRADIXDGIT, Illegal digit in binary, octal, or hexadecimal constant

Error.

IVREDECL, Illegal redeclaration gives “symbol name” multiple meanings in “scope name”
IVREDECLREC, Illegal redeclaration gives “symbol name” multiple meanings in this record
IVREDEF, Illegal redefinition gives “symbol name” multiple meanings in “scope name”

Warning: When an identifier is used in any given block, it must have the same meaning wherever it appears in the block.

IVUSEALIGN, Invalid use of alignment attribute
IVUSEALLOC, Invalid use of allocation attribute

Error.

IVUSEATTR, Invalid use of “attribute name” attribute

Error: The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object’s type.

IVUSEATTRLST, Invalid use of an attribute list

Error.

IVUSEBNDID, Illegal use of bound identifier “identifier name”

Error: An identifier that represents one bound of a conformant schema was used where a variable was expected, such as in an assignment statement or in a formal VAR parameter section. The restrictions on the use of a bound identifier are identical to those on a constant identifier.

IVUSEDES, Invalid use of descriptor class attribute

Error: The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object’s type.

IVUSEFNID, Illegal use of function identifier “identifier name”

Error: Two examples of illegal uses are the assignment of values to the components of the function result (if its type is structured) and the passing of the function identifier as a VAR parameter.

IVUSEPOI, Illegal use of type POINTER or UNIV_PTR

Error: Values of type POINTER and UNIV_PTR can not be dereferenced with the ^ operator or used with the built-in routines NEW and DISPOSE.

IVUSESIZ, Invalid use of size attribute

Error: The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object’s type.

IVUSEVISIB, invalid use of visibility attribute

Error: The use of a visibility attribute conflicts with the requirements of the object’s type.

KEYINTRNG, KEY number must be an integer value in range 0..254

Error: The key number specified by a KEY attribute must fall in the integer subrange 0..254.

KEYNOTALIGN, KEY “key number” field “field name” at bit position “bit position” is unaligned
KEYORDSTR, KEY allowed only on ordinal and fixed-length string fields
KEYPCKREC, KEY field in PACKED RECORD must have an alignment attribute
KEYREDECL, Key number “key number” is multiply defined
KEYSIZ1_2_4, Size of an ordinal key must be 1, 2 or 4 bytes
KEYSIZ2_4, Size of a signed integer key must be 2 or 4 bytes

KEYSIZSTR, Size of a string key cannot exceed 255 bytes
KEYUNALIGN, KEY field cannot be UNALIGNED

Error.

LABDECIMAL, Label number must be expressed in decimal radix

Error.

LABINCTAG, Variant case label's type is incompatible with tag type

Error: The type of a constant specified as a case label of a variant record is not assignment compatible with the type of the tag field.

LABNOTFND, No definition of label "label name" in statement part of "block name"

Error: A label that you declared in a LABEL section does not prefix a statement in the executable section.

LABREDECL, Redefinition of label "label name" in "block name"

Error: A label cannot prefix more than one statement in the same block.

LABRNGTAG, Variant case label does not fall within range of tag type

Error: A constant specified as a case label of a variant record is not within the range defined for the type of the tag field.

LABTOOBIG, Label "label number" is greater than MAXINT

Error.

LABUNDECL, Undeclared label "label name"

Error: *Compaq Pascal* requires that you declare all labels in a LABEL declaration section before you use them in the executable section.

LABUNSATDECL, Unsatisfied declaration of label "label name" is not local to "block name"

Error: A label that prefixes a statement in a nested block was declared in an enclosing block.

LIBESTAB, LIB$ESTABLISH is incompatible with *Compaq Pascal*; use predeclared procedure ESTABLISH

Warning: *Compaq Pascal* establishes its own condition handler for processing Pascal-specific run-time signals. Calling LIB$ESTABLISH directly replaces the handler supplied by the compiler with a user-written handler; the probable result is improper handling of run-time signals. You should use Pascal's predeclared ESTABLISH procedure to establish user-written condition handlers.

LISTONEND, LIST attribute allowed only on final formal parameter

Error.

LISTUSEARG, Formal parameter has LIST attribute, use predeclared function ARGUMENT

Error: A formal parameter with the LIST attribute cannot be directly referenced. You should use the predeclared function ARGUMENT to reference the actual parameters corresponding to the formal parameter.

LNETOOLNG, Line too long, is truncated to 255 characters

Error: A source line cannot exceed 255 characters. If it does, the compiler disregards the remainder of the line.

LOWGTRHIGH, Low-bound exceeds high-bound

Error: The definition of the flagged subrange type is illegal because the value specified for the lower limit exceeds that for the upper limit.

MAXLENINT, Max-length must be a value of type integer

Error: The maximum length specified for type VARYING OF CHAR must be an integer in the range 1..65535; that is, the type definition must denote a legal character string.

MAXLENRNG, Max-length must be in range 1..65535

Error: The maximum length specified for type VARYING OF CHAR must be an integer in the range 1..65535; that is, the type definition must denote a legal character string.

MAXNUMENV, Maximum number of environments exceeded

Fatal: More than 512 environment files were used in the compilation.

MECHEXTERN, Foreign mechanism specifier allowed only on external routines

Error.

MISSINGEND, No matching END, expected near line “line number”

Information: The compiler expected an END statement at a location where none was found. Compilation proceeds as though the END statement were correctly located.

MODINIT26, Module name limited to 26 characters when initialization required

Error: When a module contains schema types, discriminated schema types, variables of discriminated schema types, or a TO BEGIN DO statement clause, the module name is limited to 26 characters.

MODOFNEGNUM, MOD of a negative modulus has no mathematical definition

Error: In the MOD operation A MOD B, the operand B must have a positive integer value. This message is issued only when the MOD operation occurs in a compile-time constant expression.

MSTBEARRAY, Type must be ARRAY

Error.

MSTBEARRVRY, Type must be ARRAY or VARYING

Error: You cannot use the syntax [index] to refer to an object that is not of type ARRAY or VARYING OF CHAR.

MSTBEBOOL, Control expression must be of type BOOLEAN

Error: The IF, REPEAT, and WHILE statements require a Boolean control expression.

MSTBEDEREF, Must be dereferenced

Information.

MSTBEDISCR, Schema type must be discriminated

Error: An undiscriminated schema type is not allowed everywhere that a regular type name is allowed.

MSTBEORDSETARR, Type must be ordinal, SET, or ARRAY

Error.

MSTBEREC, Type must be RECORD

Error.

MSTBERECVRY, Type must be RECORD or VARYING

Error: You cannot use the syntax "Variable.Identifier" to refer to an object that is not of type RECORD or VARYING OF CHAR.

MSTBESTAT, Cannot initialize non-STATIC variables

Error: You cannot initialize variables declared without the STATIC attribute in nested blocks, nor can you initialize program-level variables whose attributes give them some allocation other than static.

MSTBETEXT, "I/O routine" requires FILE_VARIABLE of type TEXT

Error: The READLN and WRITELN procedures operate only on text files.

MULTDECL, "symbol name" has multiple conflicting declarations, reason(s):

Error.

NCATOA, Cannot reformat content of actual's CLASS_NCA descriptor as CLASS_A

Error: This message can appear as additional information on other error messages.

NEWQUADAGN, "type name"'s base type is ALIGNED("nnn"); NEW handles at most ALIGNED(3)

Error: You cannot call the NEW procedure to allocate pointer variables whose base types specify alignment greater than a quadword. To allocate such variables, you must use external routines.

NOACTCOM, No actuals are compatible with schema formal parameter

Information: Undiscriminated schema formal parameters denoting subranges or sets cannot be used as value parameters. In these cases, no actual parameter can ever be compatible with the formal parameter.

NOASSTOFNC, Block does not contain an assignment to function result "function name"

Warning: The block of a function must include a statement that assigns a return value to the function identifier.

NOCONVAL, A constant value was not specified for field "field name"

Error.

NODECLVAR, “symbol name” is not declared in a VAR section of “block name”

Error: You cannot initialize a variable using the VALUE section if the variable was not declared in the same block in which the VALUE section appears.

NODSCREC, No descriptor class for RECORD type

Error: The *Tru64 UNIX* Calling Standard does not define a descriptor format for records; therefore, you cannot specify %DESCR for a parameter of type RECORD.

NODSCRSCH, No descriptor class for schematic types

Error.

NOFLDREC, No field “field name” in RECORD type “type name”

Error: The field specified does not exist in the specified record.

NOFRMINDECL, Declaration of “routine” parameter “routine name” supplied no formal parameter list

Information: You specified actual parameters in a call on a formal routine parameter that was declared with no formal parameters. Although such a call was legal in VAX Pascal Version 1.0, it does not follow the rules of the Pascal standard. You should edit your program to reflect this change.

NOINITEXT, Initialization not allowed on EXTERNAL variables
NOINITINH, Initialization not allowed on inherited variables

Error: You can initialize only those variables whose storage is allocated in this compilation.

NOINITVAR, Cannot initialize “symbol name”—it is not declared as a variable

Error: Variables are the only data items that can be initialized, and they can be initialized only once.

NOLISTATTR, Parameter to this predeclared function must have LIST attribute

Error: ARGUMENT and ARGUMENT_LIST_LENGTH require their first parameter to be a formal parameter with the LIST attribute.

NONATOMIC, Unable to generate code for atomic access

Warning: Due to poor alignment, the code generator is unable to generate an atomic code sequence to read or write the volatile object.

NONGRNACC, Unable to generate code for requested granularity

Warning: Due to poor alignment, the code generator is unable to generate a code sequence for the granularity requested.

NOREPRE, No textual representation for values of this type

Error: You cannot write a value to a text file (using WRITE or WRITELN) or to a VARYING string (using WRITEV) if there is no textual representation for the type. Similarly, you cannot read a value from a text file (using READ or READLN) or from a VARYING string (using READV) if there is no textual representation for the type. Such types are RECORD, ARRAY (other than PACKED ARRAY [1..n] OF CHAR), SET, and pointer.

NOTAFUNC, "symbol name" is not declared as a "routine."

Error: An identifier followed by a left parenthesis, a semicolon, or one of the reserved words END, UNTIL, and ELSE is interpreted as a call to a routine with no parameters. This message is issued if the identifier was not declared as a procedure or function identifier. Note that in the current version, functions can be called with the procedure call statement.

NOTASYNCH, "routine name" is not ASYNCHRONOUS

Information: This message can appear as additional information on other error messages.

NOTATAG, "identifier" is not a tag-identifier

Error: The identifier used with the CASE OF construct in a record constructor must be a tag identifier.

NOTATYPE, "symbol name" is not a type identifier

Error: An identifier that does not represent a type was used in a context where the compiler expected a type identifier.

NOTAVAR, "symbol name" is not declared as a variable

Error: You cannot assign a value to any object other than a variable.

NOTAVARFNID, "symbol name" is not declared as a variable or a function identifier

Error: You cannot assign a value to any object other than a variable or a function identifier.

NOTAVARPARM, "symbol name" is not declared as a variable or parameter

Error.

NOTBEADDR, May not be parameter to ADDRESS
NOTBEARGV, May not be used as a parameter to ARGV
NOTBEASSIGN, May not be assigned
NOTBECALL, May not be called as a FUNCTION
NOTBECAST, May not be type cast
NOTBEDEREF, May not be dereferenced
NOTBEDES, May not be passed by untyped %DESCR
NOTBEEVAL, May not be evaluated
NOTBEFILOP, May not be used in a file operation

NOTBEFLD, May not be field selected
NOTBEFNCPRM, May not be passed as a FUNCTION parameter
NOTBEFORCTL, May not be used as FOR loop variable
NOTBEFORDES, May not be passed as a descriptor foreign parameter
NOTBEFOREF, May not be passed as a reference foreign parameter
NOTBEIADDR, May not be parameter to IADDRESS
NOTBEIDX, May not be indexed
NOTBEIMMED, May not be passed by untyped immediate passing mechanism

NOTBENEW, May not be written into be NEW
NOTBENSTCTL, May not be control variable for an inner FOR loop
NOTBEREAD, May not be written into be READ
NOTBEREF, May not be passed by untyped reference passing mechanism
NOTBERODES, May not be passed as a READONLY descriptor foreign parameter
NOTBEROFOR, May not be passed as a READONLY reference foreign parameter
NOTBEROVAR, May not be passed as a READONLY VAR parameter
NOTBETOUCH, May not be read/modified/written

NOTBEVAR, May not be passed as a VAR parameter
NOTBEWODES, May not be passed as a WRITEONLY descriptor foreign parameter
NOTBEWOFOR, May not be passed as a WRITEONLY reference foreign parameter
NOTBEWOVAR, May not be passed as a WRITEONLY VAR parameter
NOTBEWRTV, May not be parameter to WRITEV

Information: These messages can appear as additional information on other error messages.

NOTBYTOFF, Field "field name" is not aligned on a byte boundary

Error.

NOTDECLROU, “symbol name” is not declared as a “routine.”
NOTINITIAL, “routine name” is not INITIALIZE

Information: These messages can appear as additional information on other error messages.

NOTINRNG, Value does not fall within range of the tag type

Error: The value specified as the case label of a variant record is not a legal value of the tag field’s type. This message is also issued if a case label in a call to NEW, DISPOSE, or SIZE falls outside the range of the tag type.

NOTLOOP, Loop control statement is not inside a loop

Error.

NOTNEWTYP, Schema must define a new type

Error: The type-denoter of a schema definition must define a new type; for example, a subrange, an array, or a record.

NOTXTLIB, No text library was specified at compile time

Error: The specified %INCLUDE module could not be accessed because a text library was not specified on the command line or in the PASCAL$LIBRARY logical name.

NOTSAMTYP, Not the same type
NOTUNBOUND, “routine name” is not UNBOUND

Information: These messages can appear as additional information on other error messages.

NOTSCHEMA, “symbol name” is not a schema type

Error.

NOTVARNAM, Parameter to this predeclared function must be simple variable name

Error: The parameter cannot be indexed, be dereferenced, have a field selected, or be an expression. It must be the name of the entire variable.

NOTVOLATILE, “variable name” is non-VOLATILE

Warning: You should not use the ADDRESS function on a nonvolatile variable or component or on a formal VAR parameter.

NOUNSATDECL, No unsatisfied declaration of label “label name” in “block name”

Error.

NUMFRMLPARM, Different numbers of formal parameters

Information: This message can appear as additional information on other error messages.

NXTACTDIFF, NEXT of actual’s component differs from that of other parameters in same section

Error: All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible. This message refers to the allocation size and alignment of the array’s inner dimensions.

OLDDECLSYN, Obsolete “routine” parameter declaration syntax

Information: The declaration of a formal routine parameter uses the obsolete VAX Pascal Version 1.0 syntax. You should edit your program to incorporate the current version syntax, which is mandated by the Pascal standard.

OPNDASSCOM, Operands are not assignment compatible
OPNOTINT, Operand(s) must be of type integer

Error.

OPNDNAMCOM, Operands are not name compatible

Error.

ORDOPNDREQ, Ordinal operand(s) required

Error or Warning: This message is at warning level if you try to use INT, ORD, or UINT on a pointer expression. It is at error level if you use PRED or SUCC on an expression whose type is not ordinal.

OUTNOTDECL, OUTPUT not declared in heading

Error: A call to EOF, EOLN, READLN, or WRITELN did not specify a file variable, and the default INPUT or OUTPUT was not listed in the program heading.

OVRDIVZERO, Overflow or division by zero in compile-time expression

Error.

PACKSTRUCT, “component name” of a PACKED structured type

Error or Warning: You cannot use the data items listed in a call to the ADDRESS function, nor can you pass them as writable VAR, %REF, %DESCR, or %STDESCR parameters. This message is at warning level if the variable or component has the UNALIGNED attribute, and at error level if the variable or component is actually unaligned.

PARMACTDEF, Formal parameter “parameter name” has neither actual nor default

Error: If a formal parameter is not declared with a default, you must pass an actual parameter to it when calling its routine.

PARMCLAMAT, Parameter section classes do not match

Information: This message can appear as additional information on other error messages.

PARMLIMIT, *Compaq Pascal* architectural limit of 255 parameters exceeded

Error: You cannot declare a procedure with more than 255 formal parameters. A function whose result type requires that the result be stored in more than 64 bits or whose result type is a character string cannot have more than 254 formal parameters. In a call to a routine declared with the LIST attribute, you also cannot pass more than 255 (or 254) actual parameters.

PARMSECTMAT, Division into parameter sections does not match

Information: This message can appear as additional information on other error messages.

PARSEFAIL, error parsing command line; use PASCAL command

Fatal: The *Compaq Pascal* compiler was invoked without using the PASCAL DCL command.

PARSEFAIL, error parsing command line; using an invalid CLD table

Fatal: The *Compaq Pascal* compiler was invoked with an incorrect or obsolete command-line definition in SYS$LIBRARY:DCLTABLES. Contact your system manager to reinstall SYS$LIBRARY: DCLTABLES.

PASPREILL, Passing predeclared "routine name" is illegal

Error: You cannot use the IADDRESS function on a predeclared routine for which there is no corresponding routine in the run-time library (such as the interlocked functions). In addition, you cannot pass a predeclared routine as a parameter if there is no way to write the predeclared routine's formal parameter list in *Compaq Pascal*. Examples of the latter case are the PRED and SUCC functions and many of the I/O routines.

PASSEXTERN, Passing mechanism allowed only on external routines

Error.

PASSNOTLEG, Passing mechanism not legal for this type

Error.

PCKARRBOO, PACKED ARRAY OF BOOLEAN parameter expected

Error.

PCKUNPCKCON, Packed/unpacked conflict

Information: This message can appear as additional information on other error messages.

PLACEBEFEOLN, Placeholder not terminated before end of line

Error.

PLACEIVCHAR, Illegal nonprinting character (ASCII "decimal representation of character") within placeholder

Warning.

PLACENODOT, Repetition of pseudocode placeholders not allowed

Error.

PLACESEEN, Placeholder encountered

Error.

PLACEUNMAT, Unmatched placeholder delimiter

Error.

POSAFTNONPOS, Positional parameter cannot follow a nonpositional parameter

Error.

POSALIGNCON, Position / alignment conflict

Error: The bit position specified by the POS attribute does not have the number of low-order bits implied by the specified alignment attribute.

POSINT, POS expression must be a positive integer value

Error.

PRENAMRED, Predeclared name cannot be redefined

Error: A predeclared name may not be redefined when defining constants with the /CONSTANT DCL qualifier.

PREREQPRMLST, Passing predeclared "routine name" requires formal to include parameter list

Error: To pass one of the predeclared routines EXPO, ROUND, TRUNC, UNDEFINED, UTRUNC, UROUND, DBLE, SNGL, QUAD, INT, ORD, and UINT as an actual parameter to a routine, you must specify a formal parameter list in the corresponding formal routine parameter.

PRMKWNSIZ, Parameter must have a size known at compile-time

Error: The BIN, HEX, OCT, DEC, and UDEC functions cannot be used on conformant parameters. The SIZE and NEXT functions cannot be used on conformant parameters in compile-time constant expressions.

PROCESSFILE, Compiling file "file name"

Information.

PROCESSRTN, Generating code for routine "routine name"

Information.

PROGSCHENV, PROGRAM with schema may not create environment

Error: A program that declares a schema type cannot have the [ENVIRONMENT] attribute. Schema declarations should be placed in a separate module and inherited by the program.

PROPRMEXT, Declaration of “program parameter name” is EXTERNAL—
program parameter files must be locally allocated
PROPRMFIL, A program parameter must be a variable of type FILE
PROPRMINH, Declaration of “program parameter name” is inherited—
program parameter files must be locally allocated
PROPRMLEV, Program parameter “program parameter name” is not declared
as a variable at the outermost level

Error: Any external file variable (other than INPUT and OUTPUT) that is listed in the program heading must also be declared as a file variable in a VAR section in the program block.

PSECTMAXINT, Allocation of “symbol name” causes PSECT “PSECT name” to exceed MAXINT bits

Error: The *Compaq Pascal* implementation restricts the size of a program section to 2,147,483,647 bits.

PTRCMPEQL, Pointer values may only be compared for equality

Error: The equality (=) and inequality (<>) operators are the only operators allowed for values of a pointer type; all other operators are illegal.

PTREXPRCOM, Pointer expressions are not compatible

Error: The base types of two pointer expressions being compared for equality (=) or inequality (<>) are not structurally compatible.

QUOBEFEOL, Quoted string not terminated before end of line

Error.

QUOSTRING, Quoted string expected

Error: The compiler expects the %DICTIONARY and %INCLUDE directives, and the radix notations for binary (%B), hexadecimal (%X), and octal constants (%O), to be followed by a quoted string of characters.

RADIXTEXT, Radix input requires FILE_VARIABLE of type TEXT

Error: The input radix specifiers (BIN, OCT, and HEX) operate only on text files.

READONLY, “variable name” is READONLY

Warning: You cannot use a read-only variable in any context that would store a new value in the variable. For example, a read-only variable cannot be used in a file operation.

REALCNSTRNG, Real constant out of range

Error: See the *Compaq Pascal Language Reference Manual* for details on the range of real numbers.

REALOPNDREQ, Real (SINGLE, DOUBLE or QUADRUPLE) operand(s) required

Error.

RECHASFILE, Record contains one or more FILE components, POS is illegal

Error.

RECHASTMSTMP, Record contains one or more TIMESTAMP components, POS is illegal

Error.

RECLENINT, RECORD_LENGTH expression must be of type integer

Error: The value of the record length parameter to the OPEN procedure must be an integer.

RECLENMNGLS, RECORD_LENGTH parameter is meaningless given file's type

Warning: The record length parameter is usually relevant only for files of type TEXT and VARYING OF CHAR.

RECMATCHTYP, MATCH_TYPE identifier "NXT or NXTEQL" is recommended instead of "GTR or GEQ"

Information.

REDECL, A declaration of "symbol name" already exists in "block name"

Error: You cannot redeclare an identifier or a label in the same block in which it was declared. Inheriting an environment is equivalent to including all of its declarations at program or module level.

REDECLATTR, "attribute name" already specified

Error: Only one member of a particular attribute class can appear in the same attribute list.

REDECLFLD, Record already contains a field "field name"

Error: The names of the fields in a record must be unique; they cannot be duplicated between variants.

REINITVAR, “variable name” has already been initialized

Error: Variables are the only data items that can be initialized, and they can be initialized only once.

REPCASLAB, Value has already appeared as a label in this CASE statement

Error: You cannot specify the same value more than once as a case label in a CASE statement.

REPFACZERO, Repetition factor cannot be the function ZERO
REQCLAORNCA, Arrays and conformants of this parameter type require either CLASS_A or CLASS_NCA
REQCLS, Scalars and strings of this parameter type require CLASS_S

Error.

REQNATAGN, Operand must be naturally aligned

Error.bold

REQNOCH, Primary key requires NOCHANGES option

Error.

REQPKDARR, The combination of CLASS_S and %STDESCR requires a PACKED ARRAY OF CHAR structure

Error.

REQREADVAR, READ or READV requires at least one variable to read into

Error: The READ and READV procedures require that you specify at least one variable to be read from a file.

REQWRITELEM, WRITE requires at least one write-list-element

Error: The WRITE procedure requires that you specify at least one item to be written to a file.

RESPTRTYP, Result must be a pointer type

Information.

REVRNTLAB, Value has already appeared as a label in this variant part

Error: You cannot specify the same value more than once as a case label in a variant part of a record.

RTNSTDESCR, Routines cannot be passed using %STDESCR

Error.

SCHCONST, Nonstatic constants are not allowed

Error: Constants cannot be made for nonstatic types since that would yield constants without compile-time size and value.

SCHFLDALN, Field in nonstatic type may not have greater than byte alignment

Error.

SCHOVERLAID, Use of schema types conflicts with OVERLAID attribute

Error: The OVERLAID attribute cannot be used on programs or modules that discriminate schema at the outermost level.

SENDSPR, Internal Compiler Error

Fatal: An error has occurred in the execution of the *Compaq Pascal* compiler. Along with this message, you will receive information that helps you find the location in the source program and the name of the compilation phase at which the error occurred. You may be able to rewrite the section of your program that caused the error and thus successfully compile the program. However, even if you are able to remedy the problem, please submit a problem report to Compaq and provide a machine-readable copy of the program.

SEQ11FORT, PDP–11 specific directive SEQ11 treated as equivalent to FORTRAN directive

Information.

SETBASCOM, SET base types are not compatible

Error: The base type of two sets used in a set operation are not compatible.

SETELEORD, SET element expression must be of an ordinal type

Error: The expressions used to denote the elements of a set constructor or the bounds of a set type definition must have an ordinal type.

SETNOTRNG, SET element is not in range 0..255

Error: In a set whose base type is a subrange of integers or unsigned integers, all set elements in the set's type definition or in a constructor for the set must be in the range 0..255.

SIZACTDIFF, SIZE of actual differs from that of other parameters in same section

Error: All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible. This message refers to the allocation size of the array's outermost dimension.

SIZARRNCA, Explicit size on ARRAY dimension makes CLASS_NCA mandatory

Error.

SIZATRTYPCON, Size attribute / type conflict

Error: For an ordinal type, the size specified must be at least as large as the packed size but no larger than 32 bits. Pointer types and type SINGLE must be allocated exactly 32 bits, type DOUBLE exactly 64 bits, and type QUADRUPLE exactly 128 bits. For types ARRAY, RECORD, SET, and VARYING OF CHAR, the size specified must be at least as large as their packed sizes. For the details of allocation sizes in *Compaq Pascal*, see the *Compaq Pascal Language Reference Manual*.

SIZCASTYP, Variable's size conflicts with cast's target type

Error: In a type cast operation, the size of the variable and the size of the type to which it is cast must be identical.

SIZEDIFF, Sizes are different

Information: This message can appear as additional information on other error messages.

SIZEINT, Size expression must be a positive integer value

Error.

SIZGTRMAX, Size exceeds MAXINT bits

Error: The size of a record or an array type or the size specified by a size attribute exceeds 2,147,483,647 bits. The *Compaq Pascal* implementation imposes this size restriction.

SIZMULTBYT, Size of component of array passed by descriptor is not a multiple of bytes

Error: When an array or a conformant parameter is passed using the %DESCR mechanism specifier, the descriptor built by the compiler must follow the *Tru64 UNIX* Calling Standard. Such a descriptor can describe only an array whose components fall on byte boundaries.

SPEOVRDECL, Foreign mechanism specifier required to override parameter declaration

Error: When you specify a default value for a formal VAR or routine parameter, you must also use a mechanism specifier to override the characteristics of the parameter section.

SPURIOUS, "error message" at "line number"—"column number"

Information: The compiler did not correctly note the location of this error in your program and later could not position and print the correct error message. You may be able to correct the section of your program that caused the error and thus avoid this error. Please submit a problem report and provide a machine-readable copy of the program if you receive this error.

SRCERRORS, Source errors inhibit continued compilation—correct and recompile

Fatal: A serious error previously detected in the source program has corrupted the compiler's symbol tables and inhibits further compilation. Correct the serious error and recompile the program.

SRCTXTIGNRD, Source text following end of compilation unit ignored

Warning: The compiler ignores any text following the END statement that terminates a compilation unit. This error probably resulted from an unmatched END statement in your program.

STDACTINCMP, Nonstandard: actual is not name compatible with other parameters in same section

Information: According to the Pascal standard, all actual parameters passed to a parameter section must have the same type identifier or the same type definition. This message is issued only if you have specified the standard switch on the compile command line.

STDATTRLST, Nonstandard: attribute list
STDBIGLABEL, Nonstandard: label number greater than 9999
STDBLANKPAD, Nonstandard: blank-padding used during string operation
STDBNDRMUSE, Nonstandard: usage of formal parameter for routine "routine name"
STDCALLFUNC, Nonstandard: function "function name" called as a procedure
STDCASLBLRNG, Nonstandard: label range in case selector

STDCAST, Nonstandard: type cast operator
STDCMPCOMPAT, Nonstandard: cannot "PACK or UNPACK", array component types are incompatible
STDCMPDIR, Nonstandard: compiler directive
STDCOMFUNACC, Nonstandard: component function access
STDCNFARR, Nonstandard: conformant array syntax

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDCNSTR, Nonstandard: array or record constructor

Information: A VAX Pascal Version 1.0 style constructor was used. You should convert this constructor to the new constructor syntax provided in the current version of *Compaq Pascal* to be compatible with the Extended Pascal standard.

STDCONCAT, Nonstandard: concatenation operator

Information: This message refers to extensions to Pascal and is issued only if you have specified the standard switch on the compile command line.

STDCONST, Nonstandard: "type name" constant

Information: Binary, hexadecimal, and octal constants and constants of type DOUBLE, QUADRUPLE, UNSIGNED, INTEGER64, and UNSIGNED64 are extensions to Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDCONSTACC, Nonstandard: structured constant access

Information: This message is issued if you have specified a standard option other than extended on the compile command line.

STDCTLDECL, Nonstandard: control variable "variable name" not declared in VAR section of "block name"

Information: The Pascal standard requires that the control variable of a FOR statement be declared in the same block in which the FOR statement appears.

STDDECLSEC, Nonstandard: declaration sections either out of order or duplicated in “block name”

Information: In the Pascal standard, the declaration sections must appear in the order LABEL, CONST, TYPE, VAR, PROCEDURE, and FUNCTION. The ability to specify the sections in any order is an extension. This message occurs only if you have specified the standard switch on the compile command line.

STDDEFPARM, Nonstandard: default parameter declaration

Information: This message refers to extensions to Pascal and is issued only if you have specified the standard switch on the compile command line.

STDDIRECT, Nonstandard: “directive name” directive

Information: The EXTERN, EXTERNAL, FORTRAN, and SEQ11 directives are extensions to Pascal. (FORWARD is the only directive specified by the Pascal standard.) This message is issued only if you have specified the standard switch on the compile command line.

STDDISCREF, Nonstandard: schema discriminant reference
STDDISCSCHEMA, Nonstandard: discriminated schema

Information: These messages are issued if you have specified a standard argument other than extended on the compile command line.

STDEMPCASLST, Nonstandard: empty case-list element

Information: This message is issued if you do not specify any case labels and executable statements between two semicolons or between OF and a semicolon in the CASE statement. You must also have specified the standard switch on the compile command line.

STDEMPPARM, Nonstandard: empty actual parameter position

Information: This message refers to extensions to Pascal and is issued only if you have specified the standard switch on the compile command line.

STDEMPREC, Nonstandard: empty record section

Information: The Pascal standard does not allow record type definitions of the form RECORD END. This message appears only if you have specified the standard switch on the compile command line.

STDEMPSTR, Nonstandard: empty string

Information: This message refers to extensions to Pascal and is issued only if you have specified the standard switch on the compile command line.

STDEMPVRNT, Nonstandard: empty variant

Information: This message occurs if you do not specify a variant between two semicolons or between OF and a semicolon. You must also have specified the standard switch on the compile command line.

STDEOLCOM, Nonstandard: end of line comment

Information: The message is issued if you use the exclamation point character to treat the remainder of the line as a comment. You must also have specified the standard switch on the compile command line.

STDERRPARM, Nonstandard: error-recovery parameter
STDEXPON, Nonstandard: exponentiation operator
STDEXTSTR, Nonstandard: extended string syntax

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDFLDHIDPTR, Nonstandard: record field identifier "field identifier name" hides type identifier "field identifier name"

Information.

STDFORIN, Nonstandard: SET-iteration in FOR statement

Information: This message is issued if you have specified a standard argument other than extended on the compile command line.

STDFORMECH, Nonstandard: foreign mechanism specifier

Information: This message refers to an extension to Pascal and is issued only if you have specified the standard switch on the compile command line.

STDFORWARD, Nonstandard: PROCEDURE/FUNCTION block "routine name" and its FORWARD heading are not in the same section

Information: The Extended Pascal standard requires that FORWARD declared routines must specify their corresponding blocks without intervening LABEL, CONST, TYPE, or VAR sections. This message is issued only if you have specified the extended argument to the standard switch on the compile command line.

STDFUNIDEVAR, Nonstandard: function identified variable

Information: This message is issued if you have specified a standard argument other than extended on the compile command line.

STDFUNCTRES, Nonstandard: FUNCTION returning a value of a "type name" type

Information: The ability of functions to have structured result types is an extension to Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDINCLUDE, Nonstandard: %INCLUDE directive
STDINITVAR, Nonstandard: initialization syntax in VAR section

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDKEYWRD, Nonstandard: "keyword name"

Information: This message is issued if you have specified a standard option other than extended on the compile command line.

STDMATCHVRNT, Nonstandard: no matching variant label

Information: This message is issued if you call the NEW or DISPOSE procedure, and one of the case labels specified in the call does not correspond to a case label in the record variable. You must also have specified the standard switch on the compile command line.

STDMODCTL, Nonstandard: potential uplevel modification of "variable name" prohibits use as control variable

Information: You cannot use as the control variable of a FOR statement any variable that might be modified in a nested block. This message is issued only if you have specified the standard switch on the compile command line.

STDMODULE, Nonstandard: MODULE declaration

Information: The item listed in this message is an extension to Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDNILCON, Nonstandard: use of reserved word NIL as a constant

Information: Only simple constants and quoted strings are allowed by the Pascal standard to appear as constants. Simple constants are integers, character strings, real constants, symbolic constants, and constants of BOOLEAN and enumerated types. This message is issued only if you have specified the standard switch on the compile command line.

STDNOFRML, Nonstandard: FUNCTION or PROCEDURE parameter declaration lacks formal parameter list

Information: This message is issued if you try to pass actual parameters to a formal routine parameter for which you declared no formal parameter list. You must also have specified the standard switch on the compile command line.

STDNONPOS, Nonstandard: nonpositional parameter syntax
STDOTHER, Nonstandard: OTHERWISE clause
STDPASSPRE, Nonstandard: passing predeclared "routine name"

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDNOTIN, Nonstandard: NOT IN operator

Information: This message refers to an extension in Pascal and is issued only if you have specified the standard switch on the compile command line.

STDPCKSET, Nonstandard: combination of packed and unpacked sets

Information: The Pascal standard does not allow packed and unpacked sets to be combined in set operations. This message is issued only if you have specified the standard switch on the compile command line.

STDPRECONST, Nonstandard: predeclared constant "constant name"

Information: The constants MAXCHAR, MAXINT64, MAXUNSIGNED, MAXUNSIGNED64, MAXREAL, MINREAL, EPSREAL, MAXDOUBLE, MINDOUBLE, EPSDOUBLE, MAXQUADRUPLE, MINQUADRUPLE, and EPSQUADRUPLE are extensions to Pascal. MAXCHAR, MAXREAL, MINREAL, and EPSREAL are contained in Extended Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDPREDECL, Nonstandard: predeclared “routine”

Information: Many predeclared procedures and functions are extensions to Pascal. The use of these routines causes this message to be issued if you have specified the standard switch on the compile command line.

STDPRESCH, Nonstandard: predefined schema “type name”

Information: This message is issued if you have specified a standard switch other than extended on the compile command line.

STDPRETYP, Nonstandard: predefined type “type name”

Information: The types SINGLE, DOUBLE, INTEGER64, QUADRUPLE, UNSIGNED, UNSIGNED64, and VARYING OF CHAR are extensions to Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDQUOSTR, Nonstandard: quotes enclosing radix constant

Information: This message is issued if you have specified the extended option on the compile command line.

STDRADFORMAT, Nonstandard: use format “radix”#nnn for radix constant

Information: This message refers to the use of an extension to Pascal. This message is issued only if you have specified the extended argument to the standard switch on the compile command line.

STDRADIX, Nonstandard: radix constant

Information: This message refers to the use of an extension to Pascal. This message is issued only if you have specified a standard switch other than extended on the compile command line.

STDRDBIN, Nonstandard: binary input from a TEXT file
STDRDENUM, Nonstandard: enumerated type input from a TEXT file
STDRDHEX, Nonstandard: hexadecimal input from a TEXT file
STDRDOCT, Nonstandard: octal input from a TEXT file
STDRDSTR, Nonstandard: string input from a TEXT file

Information: The Pascal standard allows only INTEGER, CHAR, and REAL values to be read from a text file. The ability to read values of other types is an extension to Pascal. These messages are issued only if you have specified the standard switch on the compile command line.

STDREDECLNIL, Nonstandard: redeclaration of reserved word NIL

Information: The Pascal standard considers NIL a reserved word, while *Compaq Pascal* considers it to be a predeclared identifier. Thus, if you have specified the standard switch on the compile command line, this message will be issued if you attempt to redefine NIL.

STDREM, Nonstandard: REM operator

Information: The item listed in this message is an extension to Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDSCHEMA, Nonstandard: schema type definition

Information: This message is issued if you have specified a standard argument other than extended on the compile command line.

STDSCHEMAUSE, Nonstandard: use of schema type

Information: This message is issued if you have specified a standard argument other than extended on the compile command line.

STDSIMCON, Nonstandard: only simple constant (optional sign) or quoted string

Information: Only simple constants and quoted strings are allowed by the Pascal standard to appear as constants. Simple constants are integers, character strings, real constants, symbolic constants, constants of type BOOLEAN, and enumerated types. This message is issued only if you have specified the standard switch on the compile command line.

STDSPECHAR, Nonstandard: “$” or “_” in identifier
STDSTRCOMPAT, Nonstandard: string compatibility

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDSTRUCT, Nonstandard: types do not have same name

Information: Because the Pascal standard does not recognize structural compatibility, two types must have the same type identifier or type definition to be compatible. This message is issued only if you have specified the standard switch on the compile command line.

STDSUBSTRING, Nonstandard: substring notation

Error.

STDSYMLABEL, Nonstandard: symbolic label

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDTAGFLD, Nonstandard: invalid use of tag field

Information: The tag field of a variant record cannot be a parameter to the ADDRESS function, nor can you pass it as a writable VAR, %REF, %DESCR, or %STDESCR formal parameter. This message is issued only if you have specified the standard switch on the compile command line.

STDTODECL, Nonstandard: TO BEGIN/END DO declaration

Information: This message is issued if you have specified a standard argument other than extended on the compile command line.

STDUNSAFE, Nonstandard: UNSAFE compatibility

Information: If you have used the UNSAFE attribute on an object that is later tested for compatibility, you will receive this message. You must also have specified the standard switch on the compile command line.

STDUSEDCNF, Nonstandard: conformant array used as a string
STDUSEDPCK, Nonstandard: PACKED ARRAY [1..1] OF CHAR used as a string

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDVALCNFPRM, Nonstandard: conformant array may not be passed to value conformant parameter

Information.

STDVALUE, Nonstandard: VALUE initialization section
STDVAXCDD, Nonstandard: %DICTIONARY directive

Information: These messages refere to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDVRNTCNSTR, Nonstandard: variant field outside constructor variant part

Information: This message refers to the use of an extension to Pascal. This message is issued only if you have specified the extended argument to the standard switch the compile command line.

STDVRNTPART, Nonstandard: empty variant part

Information: According to the Pascal standard, a variant part that declares no case labels and field lists between the words OF and END is illegal. This message occurs only if you have specified the standard switch on the compile command line.

STDVRNTRNG, Nonstandard: variant labels do not cover the range of the tag type

Information: According to the Pascal standard, you must specify one case label for each value in the tag type of a variant record. This message is issued only if you have specified the standard switch on the compile command line.

STDWRTBIN, Nonstandard: binary output to a TEXT file
STDWRTENUM, Nonstandard: user defined enumerated type output to a TEXT file
STDWRTHEX, Nonstandard: hexadecimal output to a TEXT file
STDWRTOCT, Nonstandard: octal output to a TEXT file

Information: The Pascal standard allows only INTEGER, BOOLEAN, CHAR, REAL, and PACKED ARRAY [1..n] OF CHAR values to be written to a text file. The ability to write values of other types is an extension to Pascal. These messages are issued only if you have specified the standard switch on the compile command line.

STDZERO, Nonstandard: ZERO function used in constructor

Information: This message refers to an extension in Pascal and is issued only if you have specified the standard switch on the compile command line.

STOREQEXC, Allocates to Psect "name" exceded growth bounds

Error: Too much data is allocated to the Psect. Either place variables into different Psects or break up the program into multiple modules

STREQLLEN, String values must be of equal length

Error: You cannot perform string comparisons on character strings that have different lengths.

STROPNDREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) operand required
STRPARMREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) parameter required
STRTYPREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) type required

Error: The file-name parameter to the OPEN procedure and the parameter to the LENGTH function must be character strings of the types listed.

SYNASCII, Illegal ASCII character
SYNASSERP, Syntax: ":=", ";" or ")" expected
SYNASSIGN, Syntax: ":=" expected
SYNASSIN, Syntax: ":=" or IN expected
SYNASSSEMI, Syntax: ":=" or ";" expected
SYNATRCAST, Syntax: attribute list not allowed on a type cast
SYNATTTYPE, Syntax: attribute-list or type specification
SYNBEGDECL, Syntax: BEGIN or declaration expected

SYNBEGEND, Syntax: BEGIN or END expected
SYNBEGIN, Syntax: BEGIN expected
SYNCOASSERP, Syntax: ",", ":=", ";" or ")" expected
SYNCOELRB, Syntax: ",", ".." or "]" expected
SYNCOLCOMRP, Syntax: ":", "," or ")" expected
SYNCOLON, Syntax: ":" expected
SYNCOMCOL, Syntax: "," or ":" expected
SYNCOMDO, Syntax: "," or DO expected
SYNCOMEQL, Syntax: "," or "=" expected

SYNCOMMA, Syntax: "," expected
SYNCOMRB, Syntax: "," or "]" expected
SYNCOMRP, Syntax: "," or ")" expected
SYNCOMSEM, Syntax: "," or ";" expected
SYNCONTMESS, Syntax: CONTINUE or MESSAGE expected
SYNCOSERP, Syntax: ",", ";" or ")" expected
SYNDIRBLK, Syntax: directive or block expected

Error: The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

SYNDIRMIS, Syntax: directive missing, EXTERNAL assumed

Error: In the absence of a directive where one is expected, the compiler assumes that EXTERNAL is the intended directive and proceeds with compilation based on that assumption.

SYNDO, Syntax: DO expected
SYNELIPSIS, Syntax: ".." expected
SYNELSESTMT, Syntax: ELSE or start of new statement expected
SYNEND, Syntax: END expected
SYNEQL, Syntax: "=" expected
SYNEQLLP, Syntax: "=" or "(" expected
SYNERRCTE, Error in compile-time expression
SYNEXPR, Syntax: expression expected
SYNEXSEOTEN, Syntax: expression, ";", OTHERWISE or END expected

SYNFUNPRO, Syntax: FUNCTION or PROCEDURE expected
SYNHEADTYP, Syntax: routine heading or type identifier expected
SYNIDCAEND, Syntax: identifier, CASE or END expected
SYNIDCARP, Syntax: identifier, CASE or ")" expected
SYNIDCASE, Syntax: identifier or CASE expected
SYNIDENT, Syntax: identifier expected
SYNILLEXPR, Syntax: ill-formed expression
SYNINT, Syntax: integer expected
SYNINTBOO, Syntax: integer, boolean, or string literal expected

SYNINVSEP, Syntax: invalid token separator
SYNIVATRLST, Syntax: illegal attribute list
SYNIVPARM, Syntax: illegal actual parameter
SYNIVPRMLST, Syntax: illegal actual parameter list
SYNIVSYM, Syntax: illegal symbol
SYNIVVAR, Syntax: illegal variable
SYNLABEL, Syntax: label expected
SYNLBRAC, Syntax: "[" expected

SYNLPAREN, Syntax: "(" expected
SYNLPASEM, Syntax: "(" or ";" expected
SYNLPCORB, Syntax: "(", "," or "]" expected
SYNLPSECO, Syntax: "(", ";" or ":" expected
SYNMECHEXPR, Syntax: mechanism specifier or expression expected
SYNNEWSTMT, Syntax: start of new statement expected
SYNOF, Syntax: OF expected
SYNPARMLST, Syntax: actual parameter list

SYNPARMSEC, Syntax: parameter section expected
SYNPERIOD Syntax: "." expected.
SYNPROMOD, Syntax: PROGRAM or MODULE expected
SYNQUOSTR, Syntax: quoted string expected
SYNRBRAC, Syntax: "]" expected
SYNRESWRD, Syntax: reserved word cannot be redefined
SYNRPAREN, Syntax: ")" expected
SYNRPASEM, Syntax: ";" or ")" expected
SYNRTNTYPCNF, Syntax: routine heading, type identifier or conformant parameter expected

SYNSEMI, Syntax: ";" expected
SYNSEMIEND, Syntax: ";" or END expected
SYNSEMMODI, Syntax: ";", "::", "^", or "[" expected
SYNSEMRB, Syntax: ";" or "]" expected
SYNSEOTEN, Syntax: ";", OTHERWISE or END expected
SYNTHEN, Syntax: THEN expected
SYNTODOWN, Syntax: TO or DOWNTO expected
SYNSEOTRP, Syntax: ";", OTHERWISE, or ")" expected
SYNTYPCNF, Syntax: type identifier or conformant parameter expected

SYNTYPID, Syntax: type identifier expected

Error: The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

SYNTYPPACK, Only ARRAY, FILE, RECORD or SET types can be PACKED

Warning: You cannot pack any type other than the structured types listed in the message.

SYNTYPSPEC, Syntax: type specification expected
SYNUNEXDECL, Syntax: declaration encountered in executable section
SYNUNTIL, Syntax: UNTIL expected
SYNXTRASEMI, Syntax: "; ELSE" is not valid Pascal, ELSE matched with IF on line "line number"

Error: The compiler either detected an error in a lexical or syntactical element in your program, or it failed to find such an element where one was expected.

TAGNOTORD, Tag type must be an ordinal type

Error: The type of a variant record's tag field must be one of the ordinal types.

TOOIDXEXPR, Too many index expressions; type has only “number of dimensions” dimensions

Error: A call to the UPPER or LOWER function specified an index value that exceeds the number of dimensions in the dynamic array.

TOOMANYIFS, Conditional compilation nesting level exceeds implementation limit

Error: %IF directives may only be nested 32 deep.

TOPROGRAM, TO BEGIN/END DO not allowed in PROGRAM

Error: TO BEGIN DO and TO END DO declarations are only allowed in modules.

TYPCNTDISCR, Type can not be discriminated in this context

Error.

TYPFILSIZ, Type contains one or more FILE components, size attribute is illegal

Error: The allocation size of a FILE type cannot be controlled by a size attribute; therefore, you cannot use a size attribute on any type that has a file component.

TYPHASFILE, Type contains one or more FILE components

Error: Many operations are illegal on objects of type FILE and objects of structured types with file components; for example, you cannot initialize them, use them as value parameters, or read them with input procedures.

TYPHASNOVRNT, Type contains no variant part

Error: You can only use the formats of the NEW, DISPOSE, and SIZE routines that allow case labels to be specified when their parameters have variants.

TYPPTRFIL, Type must be pointer or FILE

Error: You cannot use the syntax “Variable^” to refer to an object whose type is not pointer or FILE.

TYPREF, %REF not allowed for this type

Error: The %REF foreign mechanism specifier cannot be used with schematic variables.

TYPSTDESCR, %STDESCR not allowed for this type

Error: The %STDESCR mechanism specifier is allowed only on objects of type CHAR, PACKED ARRAY [1..n] OF CHAR, VARYING OF CHAR, and arrays of these types.

TYPVARYCHR, Component type of VARYING must be CHAR

Error.

UNALIGNED, “variable name” is UNALIGNED

Error or Warning: You cannot use the data items listed in a call to the ADDRESS function, nor can you pass them as writable VAR, %REF, %DESCR, or %STDESCR parameters. This message is at warning level if the variable or component has the UNALIGNED attribute, and at error level if the variable or component is actually unaligned.

UNAVOLACC, Volatile access appears unaligned, but must be aligned at run-time to ensure atomicity and byte granularity

Warning: The code generator was unable to determine if a volatile access was aligned or not. It generated two sequences; one sequence will perform the atomic access if it was aligned properly; the second sequence accesses the object, but may contain a timing window where incorrect results may occur.

UNBPNTRET, “routine name” is not UNBOUND—frame-pointer not returned

Warning: The IADDRESS function returns only the address of the procedure value (on *OpenVMS VAX* systems, the entry mask of the routine is called). This address may be sufficient information to successfully invoke an unbound routine, but not a bound routine. (Bound routines are represented as a pair of addresses: one pointing to the procedure value and the other to the frame pointer to the routine in which the routine was declared.)

UNCALLABLE, Routine "name" can never be called

Information.

UNCERTAIN, “Variable name” has not been initialized

Information.

UNDECLFRML, Undeclared formal parameter “symbol name”

Error: A formal parameter name listed in a nonpositional call to a routine does not match any of the formal parameters declared in the routine heading.

UNDECLID, Undeclared identifier “symbol name”

Error: In Pascal, an identifier must be declared before it is used. There are no default or implied declarations.

UNINIT, Variable "name" is fetched, not initialized

Explanation:

Information.

UNDSCHILL, Undiscriminated schema type is illegal

Error: An undiscriminated schema type does not have any actual discriminants. Without discriminants, the type size, any nested ARRAY bounds, and the offset of any nested RECORD fields are unknown.

UNINIT, “Variable name” is fetched, not initialized

Info.

UNPREDRES, Calling FUNCTION “function name” declared FORWARD may yield unpredictable results

Warning: By using FORWARD declared functions in actual discriminant expressions, you can cause infinite loops at run time or access violations.

UNREAD, Variable, “variable name” is assigned into, but never read

Information.

UNSCNFVRY, UNSAFE attribute not allowed on conformant VARYING parameter

Error.

UNSEXCRNG, UNSIGNED constant exceeds range

Error: The largest value allowed for an UNSIGNED integer is 4,294,967,295.

UNUSED, Variable, “variable name” is never referenced

Information.

UNWRITTEN, Variable “variable name” is read, but never assigned into

Warning.

UPLEVELACC, Unbound “routine name” precludes uplevel access to “variable name”

Error: A routine that was declared with the UNBOUND attribute cannot refer to automatic variables, routines, or labels declared in outer blocks.

UPLEVELGOTO, Unbound “routine name” precludes uplevel GOTO to “label name”

Error: A routine that was declared with the UNBOUND attribute cannot refer to automatic variables, routines, or labels declared in outer blocks.

USEDBFDECL, “symbol name” was used before being declared

Warning.

USEINISTA, Use initial-state (VALUE clause) on TYPE or VAR declaration

Information: Nonstatic variables, such as those created from schema types, cannot be initialized in the VALUE declaration part. To initialize these variables, you must use the initial state feature.

V1DYNARR, Decommitted Version 1 dynamic array type

Error: The type syntax used to define a dynamic array parameter has been decommitted for the current version of *Compaq Pascal*. You should edit your program to make the type definition conform to the current version conformant array syntax.

V1DYNARRASN, Decommitted Version 1 dynamic array assignment

Error: In VAX Pascal Version 1.0, dynamic arrays used in assignments could not be checked for compatibility until run time. This warning indicates that your program depends on an obsolete feature, which you should consider changing to reflect the current version syntax for conformant array parameters.

V1MISSPARM, Decommitted missing parameter syntax: correct by adding “number of commas” comma(s)

Error: An OPEN procedure called with the decommitted VAX Pascal Version 1.0 syntax fails to mark omitted parameters with commas. Your program depends on this obsolete feature, and you should insert the correct number of commas as listed in the message.

V1PARMSYN, Use of unsupported V1 omitted parameter syntax with new V2 feature(s)

Error: In a parameter list for the OPEN procedure, you cannot use both the Version 1.0 syntax for OPEN and the parameters that are new to subsequent versions of *Compaq Pascal*.

V1RADIX, Decommitted Version 1 radix output specification

Error: In VAX Pascal Version 1.0, octal and hexadecimal values could be written by placing the keywords OCT or HEX after a field width expression. Your program uses this obsolete feature; you should consider changing it to use the current versions OCT or HEX predeclared functions.

VALOUTBND, Value to be assigned is out of bounds

Error: A value specified in an array or record constructor exceeds the subrange defined as the type of the corresponding component.

VALUEINIT, VALUE variables must be initialized

Error: Variables with both the VALUE and GLOBAL attributes must be given an initial value in either the VAR section or in the VALUE section.

VALUETOOBIG, VALUE attribute not allowed on objects larger than 32 bits

Error: Variables with the VALUE attribute cannot be larger than 32 bits because they are expressed to the linker as global symbol references.

VALUETYP, VALUE allowed only on ordinal or real types

Error.

VALUEVISIB, GLOBAL or EXTERNAL visibility is required with the VALUE attribute

Error: Variables with the VALUE attribute must be given either external or global visibility. (If the variable is given global visibility, then it must also be given an initial value.)

VARCOMFRML, Variable is not compatible with formal parameter "formal parameter name"

Error: A variable being passed as an actual parameter is not compatible with the corresponding formal parameter indicated. Variable parameters must be structurally compatible. The reason for the incompatibility is provided in an informational message that the compiler prints along with this error message.

VARNOTEXT, Variable must be of type TEXT

Error: The EOLN function requires that its parameter be a file of type TEXT.

VARPRMRTN, Formal VAR parameter may not be a routine

Error: The reserved word VAR cannot precede the word PROCEDURE or FUNCTION in a formal parameter declaration.

VARPTRTYP, Variable must be of a pointer type

Error: The NEW and DISPOSE procedures operate only on pointer variables.

VARYFLDS, LENGTH and BODY are the only fields in a VARYING type

Error: You cannot use the syntax "Variable.Identifier" to specify any fields of a VARYING OF CHAR variable other than LENGTH and BODY.

VISAUTOCON, Visibility / AUTOMATIC allocation conflict

Error: The GLOBAL, EXTERNAL, WEAK_GLOBAL, and WEAK_ EXTERNAL attributes require static allocation and therefore conflict with the AUTOMATIC attribute.

VISGLOBEXT, Visibilities are not GLOBAL/EXTERNAL or EXTERNAL /EXTERNAL

Information: In repeated declarations of a variable or routine, only one declaration at most can be global; all others must be external. This message can appear as additional information for other error messages.

VRNTRNG, Variant labels do not cover the range of the tag type

Error: According to the Pascal standard, you must specify one case label for each value in the tag type of a variant record or include an OTHERWISE clause.

WDTHONREAL, Second field width is allowed only when value is of a real type

Error: The fraction value in a field-width specification is allowed only for real-number values.

WRITEONLY, "variable name" is WRITEONLY

Warning: You cannot use a write-only variable in any context that requires the variable to be evaluated. For example, a write-only variable cannot be used as the control variable of a FOR statement.

XTRAERRORS, Additional diagnostics occurred on this line

Information: The number of errors occurring on this line exceeds the implementation's limit for outputting errors. You should correct the errors given and recompile your program.

ZERNOTALL, ZERO is not allowed for type or types containing "type name"

Error: ZERO may not be used to initialize objects of type FILE, TEXT, or TIMESTAMP or objects containing these types.

C.3 Run-Time Diagnostics

During execution, an image can generate a fatal error called an exception condition. When the *Compaq Pascal* run-time system detects such a condition, the system displays an error message and aborts program execution.

Compaq Pascal run-time system diagnostics are preceded by the following:

```
pascal: Fatal error-text
```

The severity level of a run-time error is F, fatal error.

Some conditions, particularly I/O errors, may cause several messages to be generated. The first message is a diagnostic that specifies the file that was being accessed (if any) when the error occurred and the nature of the error.

All diagnostic messages contain a brief explanation of the event that caused the error. This section lists run-time diagnostic messages in alphabetical order, including explanatory message text. Where the message text is not self-explanatory, additional explanation follows. Portions of the message text enclosed in quotation marks are items that the compiler substitutes with the name of a data object when it generates the message.

ACCMETINC, ACCESS_METHOD specified is incompatible with this file

Explanation: The value of the ACCESS_METHOD parameter for a call to the OPEN procedure is not compatible with the file's organization or record type. You can use DIRECT access only with files that have relative organization or sequential organization and fixed-length records. You can use KEYED access only with indexed files.

User Action: Make sure that you are accessing the correct file.

AMBVALENU, “string” is an ambiguous value for enumerated type “type”

Explanation: While a value of an enumerated type was being read from a text file, not enough characters of the identifier were found to specify an unambiguous value.

User Action: Specify enough characters of the identifier so that it is not ambiguous.

ARRINDVAL, array index value is out of range

Explanation: You enabled bounds checking for a compilation unit and attempted to specify an index that is outside the array’s index bounds.

User Action: Correct the program or data so that all references to array indexes are within the declared bounds.

ARRNOTCOM, conformant array is not compatible

Explanation: You attempted to assign one dynamic array to another that did not have the same index bounds. This error occurs only when the arrays use the decommitted VAX Pascal Version 1.0 syntax for dynamic array parameters.

User Action: Correct the program so that the two dynamic arrays have the same index bounds. You could also change the arrays to conform to the current syntax for conformant arrays; most incompatibilities could then be detected at compile time rather than at run time. See the *Compaq Pascal Language Reference Manual* for more information on current conformant arrays.

ARRNOTSTR, conformant array is not a string

Explanation: In a string operation, you used a conformant PACKED ARRAY OF CHAR value whose index had a lower bound not equal to 1 or an upper bound greater than 65535.

User Action: Correct the array’s index so that the array is a character string.

ASSERTION, Pascal assertion failure

Explanation: The expression used in the Pascal ASSERT builtin routine evaluated to false.

User Action: Correct the problem that was being checked with the ASSERT builtin in the source program.

BUGCHECK, internal consistency failure “nnn” in Pascal Run-Time Library

Explanation: The run-time library has detected an internal error or inconsistency. This problem may be caused by an out-of-bounds array reference or a similar error in your program.

User Action: Rerun your program with all CHECK options enabled. If you are unable to find an error in your program, please submit a problem report, including a machine-readable copy of your program, data, and a sample execution illustrating the problem.

CANCNTERR, handler cannot continue from a nonfile error

Explanation: A user condition handler attempted to return SS$_CONTINUE for an error not involving file input/output. To recover from such an error, you must use either an uplevel GOTO statement or the SYS$UNWIND system service.

User Action: Modify the user handler to use one of the allowed recovery actions for nonfile errors, or to resignal the error if no recovery action is possible.

CASSELVAL, CASE selector value is out of range

Explanation: The value of the case selector in a CASE statement does not equal any of the specified case labels, and the statement has no OTHERWISE clause.

User Action: Either add an OTHERWISE clause to the CASE statement or change the value of the case selector so that it equals one of the case labels. See the *Compaq Pascal Language Reference Manual* for more information.

CONCATLEN, string concatenation has more than 65535 characters

Explanation: The result of a string concatenation operation would result in a string longer than 65,535 characters, which is the maximum length of a string.

User Action: Correct the program so that all concatenations result in strings no longer than 65,535 characters.

CSTRCOMISS, invalid constructor: component(s) missing

Explanation: The constructor did not specify sufficient component values to initialize a variable of the type.

User Action: Specify more components in the constructor, use the OTHERWISE clause in the constructor, or modify the type definition to specify fewer components.

CURCOMUND, current component is undefined for DELETE or UPDATE

Explanation: You attempted a DELETE or UPDATE procedure when no current component was defined. A current component is defined by a successful GET, FIND, FINDK, RESET, or RESETK that locks the component. Files opened with HISTORY:=READONLY never lock components.

User Action: Correct the program so that a current component is defined before executing DELETE or UPDATE.

DELNOTALL, DELETE is not allowed for a sequential organization file

Explanation: You attempted a DELETE procedure for a file with sequential organization, which is not allowed. DELETE is valid only on files with relative or indexed organization.

User Action: Make sure that the program is referencing the correct file.

ERRDURDIS, error during DISPOSE

Explanation: An error occurred during execution of a DISPOSE procedure. An additional message that further describes the error may also be displayed.

User Action: Make sure that the heap storage being freed was allocated by a successful call to the NEW procedure, and that it has not been already freed.

ERRDURNEW, error during NEW

Explanation: An error occurred during execution of the NEW procedure. An additional message is displayed that further describes the error.

ERRDUROPE, error during OPEN

Explanation: An unexpected error occurred during execution of the OPEN procedure, or during an implicit open caused by a RESET or REWRITE procedure. An additional message is displayed that further describes the error.

EXTNOTALL, EXTEND is not allowed for a shared file

Explanation: Your program attempted an EXTEND procedure for a file for which the program did not have exclusive access. EXTEND requires that no other users be allowed to access the file. Note that this message may also be issued if you do not have permission to extend to the file.

User Action: Correct the program so that the file is opened with SHARING:=NONE, which is the default, before performing an EXTEND procedure.

FAIGETLOC, failed to GET locked component

Explanation: Your program attempted to access a component of a file that was locked by another user. You can usually expect this condition to occur when more than one user is accessing the same relative or indexed file.

User Action: Determine whether this condition should be allowed to occur. If so, modify your program so that it detects the condition and retries the operation later.

FILALRACT, file "file name" is already active

Explanation: Your program attempted a file operation on a file for which another operation was still in progress. This error can occur if a file is used in AST or condition-handling routines.

User Action: Modify your program so that it does not try to use files that may currently be in use.

FILALRCLO, file is already closed

Explanation: Your program attempted to close a file that was already closed.

User Action: Modify your program so that it does not try to close files that are not open.

FILALROPE, file is already open

Explanation: Your program attempted to open a file that was already open.

User Action: Modify your program so that it does not try to open files that are already open.

FILNAMREQ, FILE_NAME required for this HISTORY or DISPOSITION

Explanation: Your program attempted to open a nonexternal file without specifying a file-name parameter to the OPEN procedure, but the HISTORY or DISPOSITION parameter specified requires a file name.

User Action: Add a file-name parameter to the OPEN procedure call, specifying an appropriate file name.

FILNOTDIR, file is not opened for direct access

Explanation: Your program attempted to execute a DELETE, FIND, LOCATE, or UPDATE procedure on a file that was not opened for direct access.

User Action: Modify the program to specify the ACCESS_ METHOD:=DIRECT parameter to the OPEN procedure when opening the file.

FILNOTFOU, file not found

Explanation: Your program attempted to open a file that does not exist. An additional RMS message is displayed that further describes the problem.

User Action: Make sure that you are specifying the correct file.

FILNOTGEN, file is not in Generation mode

Explanation: Your program attempted a file operation that required the file to be in generation mode (ready for writing).

User Action: Modify the program to use a REWRITE, TRUNCATE, or LOCATE procedure to place the file in generation mode as appropriate.

FILNOTINS, file is not in Inspection mode

Explanation: Your program attempted a file operation that required the file to be in inspection mode (ready for reading).

User Action: Modify the program to use a RESET, RESETK, FIND, or FINDK procedure to place the file in inspection mode as appropriate.

FILNOTKEY, file is not opened for keyed access

Explanation: Your program attempted to execute a FINDK, RESETK, DELETE, or UPDATE procedure on a file that was not opened for keyed access.

User Action: Modify the program to specify the ACCESS_METHOD:=KEYED parameter to the OPEN procedure when opening the file.

FILNOTOPE, file is not open

Explanation: Your program attempted to execute a file manipulation procedure on a file that was not open.

User Action: Correct the program to open the file using a RESET, REWRITE, or OPEN procedure as appropriate.

FILNOTSEQ, file is not sequential organization

Explanation: Your program attempted to execute the TRUNCATE procedure on a file that does not have sequential organization. TRUNCATE is valid only on sequential files.

User Action: Make sure that your program is accessing the correct file. Correct the program so that all TRUNCATE operations are performed on sequential files.

FILNOTTEX, file is not a textfile

Explanation: Your program performed a file operation that required a file of type TEXT on a nontext file. Note that the type FILE OF CHAR is not equivalent to TEXT unless you have compiled the program with the /OLD_VERSION qualifier.

User Action: Make sure that your program is accessing the correct file. Correct the program so that a text file is always used when required.

GENNOTALL, Generation mode is not allowed for a READONLY file

Explanation: Your program attempted to place a file declared with the READONLY attribute into generation mode, which is not allowed. Note that the READONLY file attribute is not equivalent to the HISTORY:=READONLY parameter to the OPEN procedure.

User Action: Correct the program so that the file either does not have the READONLY attribute or is not placed into generation mode.

GETAFTEOF, GET attempted after end-of-file

Explanation: Your program attempted a GET operation on a file while EOF(f) was TRUE. This situation occurs when a previous GET operation (possibly implicitly performed by a RESET, RESETK, or READ procedure) reads to the end of the file and causes the EOF(f) function to return TRUE. If another GET is then performed, this error is given.

User Action: Correct the program so that it either tests whether EOF(f) is TRUE, before attempting a GET operation, or repositions the file before the end-of-file marker.

GOTOFAILED, non-local GOTO failed

Explanation: An error occurred while a nonlocal GOTO statement was being executed. This error might occur because of an error in the user program, such as an out-of-bounds array reference.

User Action: Rerun your program, enabling all CHECK options. If you cannot locate an error in your program and the problem persists, please submit a problem report to Compaq, and include a machine-readable copy of your program, data, and results of a sample execution showing the problem.

HALT, HALT procedure called

Explanation: The program terminated its execution by executing the HALT procedure. This message is solely informational.

User Action: None.

ILLGOTO, illegal uplevel GOTO during routine activation

Explanation: An uplevel GOTO was made into the body of a routine before the declaration part of the routine was completely processed.

User Action: Correct the program to avoid the uplevel GOTO until the declaration part has been completely processed.

INSNOTALL, Inspection mode is not allowed for a WRITEONLY file

Explanation: Your program attempted to place a file declared with the WRITEONLY attribute into inspection mode, which is not allowed.

User Action: Correct the program so that the file variable either does not have the WRITEONLY attribute or is not placed into inspection mode.

INSVIRMEM, insufficient virtual memory

Explanation: The run-time library was unable to allocate enough heap storage to open the file.

User Action: Examine your program to see whether it is making excessive use of heap storage, which might be allocated using the NEW procedure or the run-time library procedure LIB$GET_VM. Modify your program to free any heap storage it does not need.

INVARGPAS, invalid argument to Pascal Run-Time Library

Explanation: An invalid argument or inconsistent data structure was passed to the run-time library by the compiled code, or a system service returned an unrecognized value to the run-time library.

User Action: Rerun your program with all CHECK options enabled. Make sure that the version of the current operating system is compatible with the version of the compiler. If you cannot locate an error in your program and the problem persists, please submit a problem report to Compaq, and include a machine-readable copy of your program, data, and results of a sample execution showing the problem.

INVFILSYN, invalid file name syntax

Explanation: Your program attempted to open a file with an invalid file name. The file name used can be derived from the file variable name, the value of the file-name parameter to the OPEN procedure, or the logical name translations (if any) of the file variable name and portions of the file-name parameter and your default device and directory. The displayed text may include the erroneous file name. This error can also occur if the value of the file-name parameter is longer than 255 characters. Additional RMS messages may be displayed that further describe the error.

User Action: Use the information provided in the displayed messages to determine which component of the file name is invalid. Verify that any logical names used are defined correctly. See the *Compaq Pascal Language Reference Manual* for information on file names.

INVFILVAR, invalid file variable at location "nnn"

Explanation: The file variable passed to a run-time library procedure was invalid or corrupted. This problem might be caused by an error in the user program, such as an out-of-bounds array access. It can also occur if a file variable is passed from a routine compiled with a version of VAX Pascal earlier than Version 2.0 to a routine compiled with a later version of the compiler, or if the new key options are used on OpenVMS systems earlier than Version 4.6.

User Action: Rerun your program with all CHECK options enabled, and recompile all modules using the same compiler. If the problem persists, please submit a problem report, and include a machine-readable copy of your program, data, and results of a sample execution showing the problem.

INVKEYDEF, invalid key definition

Explanation: Your program attempted to open a file of type RECORD whose component type contained a field with an invalid KEY attribute. One of the following errors occurred:

- A new file was being created and the key numbers were not dense.
- A key field was defined at an offset of more than 65,535 bytes from the beginning of the record.

User Action: If a new file is being created, make sure that the key fields are numbered consecutively, starting with 0 for the required primary key. If you are opening an existing file, you must explicitly specify HISTORY:=OLD or HISTORY:=READONLY as a parameter to the OPEN procedure. Make sure that the length of the record is within the maximum permitted for the file organization being used.

INVRADIX, specified radix must be in the range 2-36

Explanation: The specified radix for writing an ordinal value must be in the range of 2 through 36.

User Action: Modify the program to specify a radix in the proper range

INVRECLEN, invalid record length of “nnn”

Explanation: A file was being opened, and one of the following errors occurred:

- The length of the file components was greater than that allowed for the file organization and record format (for most operations, the largest length allowed is 32,765 bytes).
- The value of the RECORD_LENGTH parameter to the OPEN procedure was greater than that allowed for the file organization and record format (for most operations, the largest value allowed is 32,765 bytes).

User Action: Correct the program so that the record length used is within the permitted limits for the type of file being used.

INVSYNBIN, “string” is invalid syntax for a binary value

Explanation: While a READ or READV procedure was reading a binary value from a text file, the characters read did not conform to the syntax for a binary value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *Compaq Pascal Language Reference Manual* for more information.

INVSYNHEX, “string” is invalid syntax for a hexadecimal value

Explanation: While a READ or READV procedure was reading a hexadecimal value from a text file, the characters read did not conform to the syntax for an hexadecimal value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *Compaq Pascal Language Reference Manual* for more information.

INVSYNENU, “string” is invalid syntax for an enumerated value

Explanation: While a READ or READV procedure was reading an identifier of an enumerated type from a text file, the characters read did not conform to the syntax for an enumerated value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *Compaq Pascal Language Reference Manual* for more information.

INVSYNINT, “string” is invalid syntax for an integer value

Explanation: While a READ or READV procedure was reading a value for an integer identifier from a text file, the characters read did not conform to the syntax for an integer value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *Compaq Pascal Language Reference Manual* for more information.

INVSYNOCT, “string” is invalid syntax for an octal value

Explanation: While a READ or READV procedure was reading an octal value from a text file, the characters read did not conform to the syntax for an octal value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *Compaq Pascal Language Reference Manual* for more information.

INVSYNREA, “string” is invalid syntax for a real value

Explanation: While a READ or READV procedure was reading a value for a real identifier from a text file, the characters read did not conform to the syntax for a real value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *Compaq Pascal Language Reference Manual* for more information.

INVSYNUNS, “string” is invalid syntax for an unsigned value

Explanation: While a READ or READV procedure was reading a value for an unsigned identifier from a text file, the characters read did not conform to the syntax for an unsigned value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *Compaq Pascal Language Reference Manual* for more information.

KEYCHANOT, key field change is not allowed

Explanation: Your program attempted an UPDATE procedure for a record of an indexed file that would have changed the value of a key field, and this situation was disallowed when the file was created.

User Action: If the program needs to detect this situation when it occurs, specify the ERROR:=CONTINUE parameter for the UPDATE procedure, and use the STATUS function to determine which error, if any, occurred. If necessary, modify the program so that it does not improperly change a key field, or recreate the file specifying that the key field is permitted to change.

KEYDEFINC, KEY “nnn” definition is inconsistent with this file

Explanation: An indexed file of type RECORD was opened, and the component type contained fields whose KEY attributes did not match those of the existing file. The number of the key in error is displayed in the message.

User Action: Correct the RECORD definition so that it describes the correct KEY fields, or recreate the file so that it matches the declared keys.

KEYDUPNOT, key field duplication is not allowed

Explanation: Your program attempted an UPDATE or PUT procedure for a record of an indexed file that would have duplicated a key field value of an existing record, and this situation was disallowed when the file was created.

User Action: If the program needs to detect this situation when it occurs, specify the ERROR:=CONTINUE parameter for the PUT or UPDATE procedure, and use the STATUS function to determine which error, if any, occurred. If necessary, modify the program so that it does not improperly duplicate a key field, or recreate the file specifying that the key field is permitted to be duplicated.

KEYNOTDEF, KEY “nnn” is not defined for this file

Explanation: Your program attempted a FINDK or RESETK procedure on an indexed file, and the key number specified does not exist in the file.

User Action: Correct the program so that the correct key numbers are used when accessing the file.

KEYVALINC, key value is incompatible with the file's key "nnn"

Explanation: The key value specified for the FINDK procedure was incompatible in type or size with the key field of the file, or your program attempted an OPEN on an existing file and the key check failed.

User Action: Make sure that the correct key value is being specified for FINDK and OPEN. Correct the program so that the type of the key value is compatible with the key of the file.

LINTOOLON, line is too long, exceeded record length by "nnn" character(s)

Explanation: Your program attempted a WRITE, PUT, WRITEV, or other output procedure on a text file that would have placed more characters in the current line than the record length of the file would allow. The number of characters that did not fit is displayed in the message.

User Action: Correct the program so that it does not place too many characters in the current line. If appropriate, use the WRITELN procedure, or specify an increased record length parameter when opening the file with the OPEN procedure.

LINVALEXC, LINELIMIT value exceeded

Explanation: The number of lines written to the file exceeded the maximum specified as the line limit. The line limit value is determined by the translation of the logical name PAS$LINELIMIT, if any, or the value specified in a call to the LINELIMIT procedure for the file.

User Action: As appropriate, correct the program so that it does not write as many lines, or increase the line limit for the file. Note that if a line limit is specified for a nontext file, each PUT procedure called for the file is considered to be one line. See the *Compaq Pascal Language Reference Manual* for more information.

LOWGTRHIGH, low-bound exceeds high-bound

Explanation: The lower bound of a subrange definition is larger than the higher bound.

User Action: Modify the declaration so the lower bound is less than or equal to the higher bound.

MAXLENRNG, maximum length must be in range 1..65535

Explanation: The maximum length for a string type is 65,535.

User Action: Modify the declaration to specify a smaller amount.

MODNEGNUM, MOD of a negative modulus has no mathematical definition

Explanation: In the MOD operation A MOD B, the operand B must have a positive integer value.

User Action: Correct the program so that the operand B has a positive integer value.

NEGDIGARG, negative Digits argument to BIN, HEX or OCT is not allowed

Explanation: Your program attempted to specify a negative value for the Digits argument in a call to the BIN, HEX, or OCT procedure, which is not permitted.

User Action: Correct the program so that only nonnegative Digits arguments are used for calls to BIN, HEX, and OCT.

NEGWIDDIG, negative Width or Digits specification is not allowed

Explanation: A WRITE or WRITEV procedure on a text file contained a field width specification that included a negative Width or Digits value, which is not permitted.

User Action: Correct the program so that only nonnegative Width and Digits parameters are used.

NOTVALTYP, "string" is not a value of type "type"

Explanation: Your program attempted a READ or READV procedure on a text file, but the value read could not be expressed in the specified type. For example, this error results if a real value read is outside the range of the identifier's type, or if an enumerated value is read that does not match any of the valid constant identifiers in its type.

User Action: Correct the program or the input data so that the values read are compatible with the types of the identifiers receiving the data.

OPNDASSCOM, operands are not assignment compatible

Explanation: The operands do not have the same type.

User Action: Examine the declarations of the operands and make sure they have compatible types.

ORDVALOUT, ordinal value is out of range

Explanation: A value of an ordinal type is outside the range of values specified by the type. For example, this error results if you try to use the SUCC function on the last value in the type or the PRED function on the first value.

User Action: Correct the program so that all ordinal values are within the range of values specified by the ordinal type.

ORGSPEINC, ORGANIZATION specified is inconsistent with this file

Explanation: The value of the ORGANIZATION parameter for the OPEN procedure that opened an existing file was inconsistent with the actual organization of the file.

User Action: Correct the program so that the correct organization is specified.

PADLENERR, PAD length error

Explanation: The length of the character string to be padded by the PAD function is greater than the length specified as the finished size, or the finished size specified is greater than 65,535.

User Action: Correct the call to PAD so that the finished size specified describes a character string of the correct length. See the *Compaq Pascal Language Reference Manual* for the rules governing the PAD function.

PASSNOTLEG, Passing mechanism not legal for this type

Error.

PTRREFNIL, pointer reference to NIL

Explanation: Your program attempted to evaluate a pointer value while its value was NIL.

User Action: Make sure that the pointer has a value before you try to evaluate it. See the *Compaq Pascal Language Reference Manual* for more information on pointer values.

RECLENINC, RECORD_LENGTH specified is inconsistent with this file

Explanation: The record length obtained from the file component's length or from the value of the record length parameter specified for the OPEN procedure was inconsistent with the actual record length of an existing file.

User Action: Correct the program so that the record length specified, if any, is consistent with the file.

RECTYPINC, RECORD_TYPE specified is inconsistent with this file

Explanation: The value of the RECORD_LENGTH parameter specified for the OPEN procedure was inconsistent with the actual record type of an existing file.

User Action: Correct the program so that the record type specified, if any, is consistent with the file.

REFINAVAR, read or write of inactive variant

Explanation: A field of an inactive variant was read or written.

User Action: Correct the program so the variant is active or remove the reference to the inactive field.

REQNATAGN, Operand must be naturally aligned

Error.

RESNOTALL, RESET is not allowed on an unopened internal file

Explanation: Your program attempted a RESET procedure for a nonexternal file that was not open. This operation is not permitted because RESET must operate on an existing file, and there is no information associated with a nonexternal file that allows RESET to open it.

User Action: Correct the program so that nonexternal files are opened before using RESET. Either OPEN or REWRITE may be used to open a nonexternal file. See the *Compaq Pascal Language Reference Manual* for more information.

REWNOTALL, REWRITE is not allowed for a shared file

Explanation: Your program attempted a REWRITE procedure for a file for which the program did not have exclusive access. REWRITE requires that no other users be allowed to access the file while the file's data is deleted. Note that this message may also be issued if you do not have permission to write to the file.

User Action: Correct the program so that the file is opened with SHARING := NONE, which is the default, before performing a REWRITE procedure.

SETASGVAL, set assignment value has element out of range

Explanation: Your program attempted to assign to a set variable a value that is outside the range specified by the variable's component type.

User Action: Correct the assignment statement so that the value being assigned falls within the component type of the set variable. See the *Compaq Pascal Language Reference Manual* for more information on sets.

SETCONVAL, set constructor value out of range

Explanation: Your program attempted to include in a set constructor a value that is outside the range specified by the set's component type, or a value that is greater than 255 or less than 0.

User Action: Correct the constructor so that it includes only those values within the range of the set's component type. See the *Compaq Pascal Language Reference Manual* for more information on sets.

SETNOTRNG, set element is not in range 0..255

Explanation: Sets of INTEGER or UNSIGNED must be in the range of 0..255.

User Action: Modify the declaration to specify a smaller range.

STDEOLCOM, Nonstandard: End of line comment

Info.

STDSUBSTRING, Nonstandard: Substring notation

Info.

STOREQEXC, Allocations to Psect name exceeded growth bounds

Error.

STRASGLEN, string assignment length error

Explanation: Your program attempted to assign to a string variable a character string that is longer than the declared maximum length of the variable (if the variable's type is VARYING) or that is not of the same length as the variable (if the variable's type is PACKED ARRAY OF CHAR).

User Action: Correct the program so that the string is of a correct length for the variable to which it is being assigned.

STRCOMLEN, string comparison length error

Explanation: Your program attempted to compare two character strings that do not have the same current length.

User Action: Correct the program so that the two strings have the same length at the time of the comparison.

SUBASGVAL, subrange assignment value out of range

Explanation: Your program attempted to assign to a subrange variable a value that is not contained in the subrange type.

User Action: Correct the program so that all values assigned to a subrange variable fall within the variable's type.

SUBSTRSEL, SUBSTR selection error

Explanation: A SUBSTR function attempted to extract a substring that was not entirely contained in the original string.

User Action: Correct the call to SUBSTR so that it specifies a substring that can be extracted from the original string. See the *Compaq Pascal Language Reference Manual* for complete information on the SUBSTR function.

TEXREQSEQ, textfiles require sequential organization and access

Explanation: Your program attempted to open a file of type TEXT that either did not have sequential organization, or had an ACCESS_METHOD other than SEQUENTIAL (the default) when opened by the OPEN procedure.

User Action: Make sure that the program refers to the correct file. Correct the program so that only sequential organization and access are used for text files.

TRUNOTALL, TRUNCATE is not allowed for a shared file

Explanation: Your program attempted to call the TRUNCATE procedure for a file that was opened for shared access. You cannot truncate files that might be shared by other users. This message may also be issued if you do not have permission to write to the file.

User Action: Correct the program so that it does not try to truncate shared files. If the file is opened with the OPEN procedure, do not specify a value other than NONE (the default) for the SHARING parameter.

UNINIT, Variable name is fetched, not initalized

Info.

UPDNOTALL, UPDATE not allowed for a sequential organization file

Explanation: Your program attempted to call the UPDATE procedure for a sequential file. UPDATE is valid only on relative and indexed files.

User Action: Correct the program so that it does not try to use UPDATE for sequential files, or recreate the file with relative or indexed organization. If you are using direct access on a sequential file, individual records can be updated with the LOCATE and PUT procedures.

VARINDVAL, VARYING index value exceeds current length

Explanation: The index value specified for a VARYING OF CHAR string is greater than the string's current length.

User Action: Correct the index value so that it specifies a legal character in the string.

WIDTOOLRG, totalwidth too large

Explanation: The requested total-width for the floating point write operation overflowed an internal buffer.

User Action: Examine the source program to see if the specified total-width parameter is correct. If it is correct, please submit a problem report including a machine-readable copy of your program, data, and a sample execution illustrating the problem.

WRIINVENU, WRITE of an invalid enumerated value

Explanation: Your program attempted to write an enumerated value using a WRITE or WRITEV procedure, but the internal representation of that value was outside the possible range for the enumerated type.

User Action: Verify that your program is not improperly using PRED, SUCC, or type casting to assign an invalid value to a variable of enumerated type.

Index

-c option
  and object libraries, 2–31
  example, 2–2, 2–31
-cord option, 4–23
  with -feedback option, 4–23
-feedback option, 4–23
-std option, 2–22
-usage option, 2–24
-V option
  example, 2–31

A

a.out file, 1–3, 1–8
ADDRESS function
  effect on optimization, 4–15
ar command, 1–10
Archive library
  creating and maintaining, 1–10
  obtaining information about, 1–9
ASYNCHRONOUS attribute
  in condition handler, 7–4
Attribute
  CHECK, C–25
  ENVIRONMENT, 3–2
  FLOAT, C–25
  HIDDEN, 3–5
  KEY, C–25
  OPTIMIZE, C–25

B

Basic blocks, 4–19
Bound procedure values, 8–3
Bounds-checking, 2–12
Bourne shell (sh)
  process limits, 1–1

C

C language, 5–1
C shell (csh)
  process limits, 1–1
Call graph, 4–19
Call stack
  contents of, 6–3
Calling block
  function return value to, 6–2
Calling standard, 6–1
CASE statement
  effect on efficiency, 4–12
cc command
  using pc command instead of, 1–7
Character string
  as function results, 6–2
CHECK attribute, C–25
Code scheduling, 4–10
comments
  processing by cpp, 5–4
Compaq Pascal
  version number
    obtaining from object file, 1–9

Compilation
  conditional, 5–3
Compilation unit, 3–1
Compiler
  and linker, 1–7, 2–1
  diagnostics, C–1
  driver program, 1–6
  functions, 1–7
  process file descriptor limit, 1–2
  process stack size, 1–1
  progress messages, 2–26
  specifying directory for temporary files, 2–29
  usage, 2–6
Compiler options, 2–6
  -assume accuracy_sensitive, 2–9
  -C, 2–10
  -constant, 2–11
  -D, 2–11
  -E, 2–12
  -g, 2–14
  -G, 2–14
  -I, 2–15
  -L, 2–17
  -nocpp, 2–17
  -nowarn, 2–18
  -o, 2–19
  -O, 2–18
  -O0, 2–18
  -O1, 2–18
  -O2, 2–18
  -O3, 2–18
  -O5, 2–18
  -om, 2–19
  -p, 2–21
  -P, 2–20
  -std, 2–22
  -V, 2–25
Compiling C language programs
  examples, 2–3
  use with pc, 1–7
Condition handler
  controlling execution, 7–4
  declaring parameters for, 7–5
  definition of, 7–1
Condition handler (cont'd)
  establishing, 7–4
  overview of, 7–2
  performing I/O to and from, 7–4
  removing, 7–5
  reporting conditions, 7–3
  return value of, 7–6
  system-defined, 7–2
  writing of, 7–4
Condition signal, 7–3
Condition value
  definition of, 7–1
Constants
  compile-time evaluation of, 4–3
  effect on efficiency, 4–12
Constructor
  for schema variant record (example), 3–11
  using OTHERWISE (example), 3–2
Conversion
  of constants, 4–3
cord, 1–9
  related commands and pc options, 4–23
cpp preprocessor
  file name suffix, 2–31

D

Data
  conversion, 8–5
  layout, 8–5
Data type
  initial-state specifier for, 3–11
Data types
  and debugger, 5–5
dbx
  coding for, 5–4
dbx debugger
  running, 5–5
Debugging
  effects of optimization, 4–16
DEC FUSE
  profiler facility, 4–20
Declaration section
  multiply declared identifiers, 3–4

Driver program
  definition of, 1–6
  relationship to software components, 1–6

E

Entry points, B–1
Enumerated type
  used in graphical data model (example), 3–10
Environment file
  cascading inheritance of (figure), 3–4
  cascading interfaces (figure), 3–8
  creation of, 3–2
  dependency checking, 3–5
Environment variables
  directory for temporary compiler files, 2–29
  PROFDIR, 4–20
  setting in .login or shell files, 1–2
  TMPDIR, 2–29
Error conditions
  detected by STATUS and STATUSV, A–1 to A–4
Error messages
  compiler, C–1 to C–65
  run-time, C–83
ESTABLISH procedure, 7–4
Establisher routine
  definition of, 7–2
Examples
  of an implementation module, 3–14
  of an interface module, 3–14
  of separate compilation, 3–13
Exception condition, 7–1
ExceptionContinueExecution return value
  returned by condition handler, 7–6
ExceptionContinueSearch return value
  returned by condition handler, 7–6
exc_raise_exception
  signaling condition with, 7–3
EXC_UNWIND function
  called by condition handler, 7–6
EXC_VIRTUAL_UNWIND function
  called by condition handler, 7–6
Executable programs
  creating using pc command, 1–3
  installing using *Tru64 UNIX* tools, 1–9
EXTERNAL attribute
  effect on routine call, 6–4

F

Feedback files
  use with pc command, 4–23
Field
  initial-state specifier for, 3–10
file command, 1–9
File dependency checking
  environment, 3–5
File descriptor limit
  increasing number per process, 1–2
Files
  changing output file names (pc), 2–3
  compiling multiple input files pc, 2–2, 2–30
  effect of options on output files pc, 2–30
  input to pc, 2–30
  temporary pc, 2–29
FLOAT attribute, C–25
FOR statement
  effect on efficiency, 4–12
Frame pointer
  of unbound routine, 6–3
Function
  example use, 1–4
  methods of returning result, 6–2
  optimization of, 4–7

G

GLOBAL attribute
  effect on routine call, 6–4
Global pointer data
  controlling the size, 4–25
GOTO statement
  effect on efficiency, 4–12

gprof command
  for call graph information, 4–21
  use with pc, 4–21 to 4–22

H

HIDDEN attribute, 3–5

I

IADDRESS function
  effect on optimization, 4–15
Identifier
  multiply declared, 3–4
IF-THEN-ELSE statement
  effect on efficiency, 4–12
%IMMED mechanism specifier, 6–5
Implementation module, 3–6
  See also Separate compilation
  example of, 3–14
Infinite loop
  See Loop
Initial-state specifier
  for array variable (example), 3–2
  on a data type, 3–11
  on variant record fields (example), 3–10
  variant record constructor (example), 3–11
INITIALIZE attribute
  effect on routine call, 6–4
Integer overflow
  run-time checking of, 4–12
Interface module, 3–6
  See also Separate compilation
  example of, 3–14
Intrinsic procedures
  for timing program execution, 4–19

K

KEY attribute, C–25
Korn shell (ksh)
  process limits, 1–1

L

.l file suffix, 2–2
Language expression
  optimization of, 4–7, 4–13
  order of evaluation, 4–7
  reordering of, 4–2
Language-Sensitive Editor/Source Code Analyzer
  See LSE, LSE/SCA, 4–2
ld linker
  functions performed, 1–8
  relationship to pc command, 1–8
  sample use with pc command, 1–3, 1–5
ld linker
  relationship to pc command, 2–1
  sample use with pc command, 2–3
Library
  list passed by pc command to ld, 1–3
  obtaining information about, 1–9
Library routines
  for timing program execution, 4–19
LINK command, 3–3
  relinking implementation modules, 3–7
  TO BEGIN DO execution order, 3–13
Linking, 2–26
  inhibiting of, 2–10
Logical expression
  optimization of, 4–7
Loop
  See Infinite Loop
Loop unrolling, 4–10
Loops
  relationship to basic block size, 4–19

M

make command, 1–9
man command
  pc(1), 1–1
MAXINT predeclared constant, 4–14
MAXINT64 predeclared constant, 4–14

Messages
  compiler, C–1 to C–65
  run-time, C–83
Migrate
  bound procedure values, 8–3
  default data layout, 8–1
  IADDRESS, 8–2
  INT, 8–2
  VOLATILE, 8–2
Module, 3–1
  finalization section, 3–13
  implementations and interfaces, 3–6
    restrictions, 3–9
  initialization and finalization sections
    restrictions, 3–9
  initialization section, 3–13
  interface inheritance path (figure), 3–6
  multiply declared identifiers, 3–4
  relinking implementation modules, 3–7
  requirements for linking modules, 3–3
  TO BEGIN DO section, 3–10

N

nm command, 1–9
Nonstandard features
  information on, 2–22
Nonstatic type
  restriction in modules, 3–9
NOOPTIMIZE attribute
  effect on optimization, 4–13

O

Object file
  contents, 1–7
  directory used, 2–29
  linking, 2–1
  naming, 2–3
  obtaining information about, 1–9
  passing directly to ld (example), 2–2
  retaining, 2–3
odump command, 1–9
Optimization, 2–18, 4–1
  code scheduling, 4–10
  compiling for optimal performance, 4–13
  considerations, 4–13
  definition of, 4–1
  disabling for debugging, 4–13
  effect on debugging, 4–16
  kinds of, 4–2
  levels of, 2–18
  loop unrolling, 4–10
  reducing errors through, 4–9
  specifying the amount, 2–18
  split lifetime analysis, 4–9
  strength reduction, 4–9
  when to avoid, 2–19
OPTIMIZE attribute, C–25
Options
  file selection, 2–15
  of compiler, 2–6
  of preprocessor, 2–11
OPTIONS statement, 2–4
OTHERWISE reserved word
  initializing an array (example), 3–2
Output files
  changing output file names (pc), 2–3
Overflow checking
  effect on efficiency, 4–12
Overview of record layout, 8–5

P

PACKED ARRAY OF CHAR type
  as function result type, 6–2
Parameter
  for condition handler, 7–5
Parameter list, 6–1
Parameter-passing semantics, 6–4
Parentheses
  effect on efficiency, 4–12
Pascal
  sample main and subprogram, 1–3, 1–4
PASCAL command, 3–3
Passing mechanisms, 6–4
  by descriptor, 6–6
  by immediate value, 6–5

Passing mechanisms (cont'd)
  by reference, 6–6
pc command
  examples, 1–5
    cord, 4–24
    profiling (gprof), 4–21
    profiling (prof), 4–20
    renaming output file, 2–3
  sample use, 1–3
pc command
  and other software components, 1–6
  consistent use of options for multiple
    compilations, 2–4
  driver program, 1–6
  effect of options on output files, 2–30
  examples
    compiling multiple files, 2–31
    linking object file, 2–3
    multiple source files, 2–2
    preserving object file, 2–3
    requesting listing file, 2–31
    requesting software pipelining, 2–3
    using .c file, 2–3, 2–10, 2–30
    using different suffix characters, 2–2
  format, 2–1, 2–31
  functions provided by options, 2–5
  groups of options, 2–5
  interpretation of suffix characters, 2–31
  introduction, 2–1
  linking, 2–1
  list of options, 2–4
  name of compiler, 1–7
  processes used by, 1–6
  sample use, 1–4
  specifying
    directory for temporary files, 2–29
    input files, 2–2, 2–30
  suffix characters and interaction with
    options, 2–31
  temporary files, 2–30
pc command examples, 2–2
Performance
  checking process limits, 1–1
  compilation, 1–1
  cord and feedback files, 4–23
Performance (cont'd)
  measuring using shell commands, 4–18
  profiling code, 4–19 to 4–25
    basic block sampling, 4–22
    call graph information, 4–21
    PC sampling, 4–20
    source line sampling, 4–23
Pipelining
  software, 4–11
pixie command, 1–9
  for basic block sampling, 4–22
  for source line sampling, 4–23
  use with *Compaq Pascal*, 4–19
  use with feedback files, 4–23
  use with pc, 4–25
Predeclared function
  optimization of, 4–7
Preprocessor
  cpp, 2–31
  defining symbols, 2–12
  inhibiting of, 2–17
  options affecting it, 2–11
Procedure calling standard, 6–1
Procedures
  analyzing performance, 4–19
Process limits
  checking
    file descriptor limit, 1–2
    stack size, 1–1
  increasing stack size, 1–1
prof command, 1–9
  for basic block sampling, 4–22
  for PC sampling, 4–20
  for source line sampling, 4–23
  options to limit report contents, 4–20
  use with *Compaq Pascal*, 4–19
  use with feedback files, 4–23
  use with pc, 4–25
PROFDIR
  environment variable, 4–20
Profiling code
  basic block sampling, 4–22
  call graph information, 4–21
  PC sampling, 4–20
  source line sampling, 4–23

Profiling code (cont'd)
 Tru64 UNIX tools, 1–9
 use with pc, 4–25
 with *Compaq Pascal*, 4–19
Program
 See also Separate compilation
 compiling, 1–1, 1–3, 1–5
 exit status
  definition of, 7–2
 modularity of, 3–1 to 3–16
 size
  process stack size, 1–1
PROGRAM statement, 1–3
Propagation
 value, 4–8

R

ranlib command, 1–10
READONLY attribute
 on pointer variables, 4–15
Record
 efficient use of nonstatic fields, 4–13
 variant (example), 3–10
Record layout
 overview, 8–5
Register
 assignment of variables to, 4–2
 effects of optimization, 4–16
REPEAT statement
 effect on efficiency, 4–12
Resignal
 definition of, 7–2
Return value
 function, 6–2
REVERT procedure, 7–5
RISC
 migrating programs, 9–1
Routine
 requirements for linking modules, 3–3
Routine activation
 definition of, 7–1
RUN command, 3–3
Run-time
 range checking, 2–9
Run-time error
 detected by STATUS or STATUSV, A–1
 messages, C–83
Run-time library
 location of, 2–26
 search path, 2–26
 standard directories, 2–17

S

Schema type
 efficient use of nonstatic fields, 4–13
 of variant record (example), 3–10
Semantics
 of parameter passing, 6–4
Separate compilation, 3–1 to 3–16
 cascading inheritance (figure), 3–4
 cascading interfaces (figure), 3–8
 examples, 3–13
 interface inheritance path (figure), 3–6
 multiply declared identifiers, 3–4
 relinking implementation modules, 3–7
 TO BEGIN DO section, 3–10
Shared library, 2–10
 choosing how to create, 2–29
 creating, 2–27
  using the `pc` and `ld`, 2–28
  using the `pc` Command, 2–28
 obtaining information about, 1–9
 specifying shared object, 2–27
Signal
 definition of, 7–2
Signal array
 in condition handlers, 7–5
size command, 1–9
Software
 pipelining, 4–11
Source files
 analyzing source code using *Tru64 UNIX* tools, 1–8
 building using *Tru64 UNIX* tools, 1–9
 creating and revising, 1–3
 managing using *Tru64 UNIX* tools, 1–8

Split lifetime analysis, 4–9
Stack
  increasing size per process, 1–1
Stack frame
  definition of, 7–1
STATUS function
  conditions detected by, A–1 to A–4
STATUSV function
  conditions detected by, A–1 to A–4
Strength reduction, 4–9
STRING type
  as function result type, 6–2
Strings
  substitution by cpp, 5–3
strings command, 1–9
strip command, 1–10
Structured statement
  effect on efficiency, 4–12
  optimization of, 4–5
Subexpression
  optimization of, 4–4
Subprogram
  external
    example, 1–4
Subrange-checking, 2–12
Symbol table
  created by compiler, 1–7

T

Temporary files
  created by pc, 2–29
  directory used by pc, 2–29
  TMPDIR environment variable used by pc, 2–29
Temporary variable
  effect on efficiency, 4–12
TMPDIR environment variable
  use during compilation, 2–29
TO BEGIN DO section, 3–10, 3–13
Type conversion
  of constants, 4–3

U

UNBOUND attribute
  effect on routine call, 6–3
Unwind
  definition of, 7–2
Utilities
  entry points, B–1

V

Variable
  effect on efficiency, 4–12
  initialization of, 3–11
  multiply declared identifiers, 3–4
  pointer, 4–15
  requirements for linking modules, 3–3
Variant record
  initial-state specifiers for (example), 3–10
  optimization considerations, 4–16
VARYING OF CHAR type
  as function result type, 6–2
VOLATILE attribute
  on pointer variables, 4–15

W

WHILE statement
  effect on efficiency, 4–12
WITH statement
  effect on efficiency, 4–12