Pascal User's Guide

THE NETWORK IS THE COMPUTER™

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 802-5761-10
Revision A, December 1996

Contents

Figures

Tables

Preface

This guide describes the Sun Workshop Compiler Pascal 4.2. This guide is to help you write and compile Pascal programs on a SPARCstation™ system.

The Pascal 4.2 compiler runs on Solaris 2.x systems. A previous major Pascal release ran also on Solaris 1.x. Features previously available on only Solaris 1.x systems generally have been dropped or superceded by others that now work on Solaris 2.x systems, as described in current documentation.

The `README` file that accompanies the product compiler may provide other release-specific information.

Note – References to Pascal in this manual refer to the Sun Workshop Compiler Pascal 4.2 unless otherwise indicated.

Operating Environment

The Sun Workshop Compiler Pascal 4.2 runs on Solaris™ 2.x systems.

For other release-specific information, see the `README` file.

Installation

Instructions for installing Pascal and other software on your SPARCstation are given in the Sun *WorkShop Installation and Licensing Guide*, which includes information on installing the online documentation.

Audience

This guide is for software engineers who write Pascal programs on a SPARCstation. It assumes you are familiar with ISO standard Pascal and the Solaris™ operating system.

Organization

This guide contains the following chapters:

- **Chapter 1, "Introduction,"** gives basic information about the Pascal compiler and related program development tools.
- **Chapter 2, "Pascal Programs,"** describes how to write, compile, and run a Pascal 4.2 program.
- **Chapter 3, "The Pascal Compiler,"** describes the `pc` command and its options.
- **Chapter 4, "Program Construction and Management,"** is an introduction to how complex programs are built in Pascal 4.2.
- **Chapter 5, "Separate Compilation,"** describes how programs can be divided into several units, and how they are compiled and linked.
- **Chapter 6, "The C–Pascal Interface,"** describes how to write programs that are partly in C and partly in Pascal.
- **Chapter 7, "The C++–Pascal Interface,"** describes how to write programs that are partly in C++ and partly in Pascal.
- **Chapter 8, "The FORTRAN–Pascal Interface,"** describes how to write programs that are partly in FORTRAN and partly in Pascal.
- **Chapter 9, "Error Diagnostics,"** describes the errors you may encounter while writing programs with Pascal.
- **Chapter 10, "Math Libraries,"** describes how to use the `libm` and `libsunmath` functions in Pascal programs.
- **Chapter 11, "The Multithread Library,"** describes how to use the multithread library in Pascal programs.
- **Appendix A, "Pascal Preprocessor,"** describes the Pascal preprocessors, with emphasis on the nonstandard preprocessor, `cppas`.
- **Appendix B, "Error Messages,"** lists all error messages Pascal produces.

Conventions Used in This Guide

This guide contains syntax diagrams of the Pascal language in extended Backus-Naur Formalism (BNF) notation. Here are the meta symbols:

Table P-1 BNF Meta Symbols

Meta Symbol	**Description**
::=	Defined as
|	Can be used as an alternative
(*a* | *b*)	Either *a* or *b*
[*a*]	Zero or one instance of *a*
{ *a* }	Zero or more instances of *a*
`'abc'`	The characters `abc`

The following table describes the type styles and symbols used in this guide:

Table P-2 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `hostname% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`hostname%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read the *User's Guide.* These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-3 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

Related Documentation

This manual is designed to accompany the following documents:

- *Pascal Language Reference*, which describes extensions to standard Pascal
- *Pascal Quick Reference Card*, which summarizes the compiler options

Both this guide and the *Pascal Language Reference* are available in the AnswerBook® system, an online documentation viewing tool that uses dynamically linked headings and cross-references.

Manual Page

Pascal 4.2 provides an online manual page (also known as a `man` page), on `pc`(1), that describes the Pascal compiler. This document is included in the Pascal 4.2 package and must be installed with the rest of the software.

Once you install the documentation, you can read about `pc` by entering the `man` command followed by the command name, as in:

```
hostname% man pc
```

`README` *Files*

The `README` default directory is `/opt/SUNWspro/READMEs`

This directory contains the following files:

- A Pascal 4.2 `README`, called `pascal`, which describes the new features, software incompatibilities, and software bugs of Pascal 4.2.
- A floating-point white paper, *What Every Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg, in PostScript™ format. The file is called `floating-point.ps`, and can be printed on any PostScript-compatible printer that has Palatino font. It can be viewed online by using the `imagetool` command:

  ```
  hostname% imagetool floating-point.ps
  ```

  This paper is available also in the AnswerBook system.

Other Related Documentation

Other reference material includes:

Incremental Link Editor (`ild`*)*
Numerical Computation Guide
Performance Profiling Tools

Documents in Hard Copy and in AnswerBook

The following table shows what documents are online, in hard copy, or both:

Table P-4 Documents in Hard Copy and in AnswerBook

Title	Hard Copy	Online
Pascal User's Guide	X	X (AnswerBook)
Pascal Language Reference	X	X (AnswerBook)
Pascal Quick Reference Card	X	
Incremental Link Editor (`ild`*)*	X	X (AnswerBook)
Numerical Computation Guide	X	X (AnswerBook)
Performance Profiling Tools	X	X (AnswerBook)
`pascal` [README file]		X (CD-ROM)
What Every Scientist Should Know About Floating-Point Arithmetic		X (AnswerBook and CD-ROM)

Introduction 1

This chapter gives an overview of the features of Sun Workshop Compiler Pascal 4.2, including compatibility, internationalization, and licensing. This chapter contains the following sections:

Standards

Sun Workshop Compiler Pascal 4.2 is a derivative of the Berkeley Pascal system distributed with UNIX® 4.2 BSD. It complies with FIPS PUB 109 ANSI/IEEE 770 X3.97-1983 and BS6192/ISO7185 at both level 0 and level 1.

Pascal Compiler

The name of the Pascal compiler is `pc`. If given an argument file name ending with `.p` or `.pas`, `pc` compiles the file and leaves the result in an executable file, called `a.out` by default.

Features

Pascal includes many extensions to the standard, including the following:

- Separate compilation of programs and modules
- `dbx` (symbolic debugger) support
- Optimizer support
- Multiple `label`, `const`, `type`, and `var` declarations
- Variable-length character strings
- Compile-time initializations
- `static` and `extern` declarations
- Different sizes of integer and `real` data types
- Integer constants in any base, from 2 to 16
- Extended input/output facilities
- Extended library of built-in functions and procedures
- Universal and function and procedure pointer types
- Direction of parameter passing: into a routine, out of a routine, or both
- Functions that return structured-type results

Note – For other release-specific information, please refer to the `README` file that accompanies the product release.

Compatibility

In general, Pascal 4.2 runs in the Solaris 2.x or above operating environment. This product is not compatible with `/usr/ucblib/libucb.a` on the Solaris 2.x environment.

Text Editors

The operating system provides two main editors:

- **Text Editor**—A window-based text editor that runs in the OpenWindows environment. Start this tool by typing **`textedit`** at the system prompt.
- **vi**—The standard visual display editor that offers the capabilities of both a line and a screen editor. It also provides several commands for editing programs. For example:
  - The `autoindent` option provides white space at the beginning of a line.
  - The `showmatch` option shows matching parentheses.

Debuggers

Sun offers a variety of programming tools that run in the Solaris operating environment. For debugging, the following tools are available:

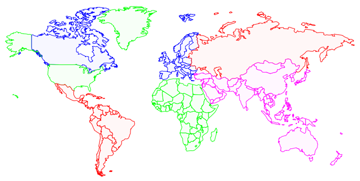
- `dbx`—A symbolic debugger
- `debugger`—A window- and mouse-based version of the symbolic debugger

You can use Pascal with fix-and-continue, a debugger functionality. See the debugger documentation for details of this feature.

Native Language Support

Sun supports the development of applications in languages other than English. These languages include most European languages and Japanese. As a result, you can easily switch your application from one native language to another. This feature is known as internationalization.

Internationalization



A product can support up to four levels of internationalization:

- **Level 1**—Allows native-language characters (such as the a-umlaut). This is referred to as the 8-bit clean model because the eighth bit in each byte is used to represent native-language characters.

- **Level 2**—Recognizes and displays international date and time formats, such as 26.07.90 in Italian; international decimal units, such as 1.234.567,89 in French; and international monetary units, such as 1.234,56 Pts in Spanish.
- **Level 3**—Contains support for localized messages and text presentation.
- **Level 4**—Contains Asian language support.

Pascal supports all four levels. See the *Pascal Language Reference* for a description of the `date` and `time` functions in internationalized formats.

Pascal does not allow number denotations *within* Pascal programs to be represented in various international formats. For example, use of a comma instead of a decimal point in numbers in Pascal program source code in French locale is not allowed. To allow such usage would not comply with the Pascal language standard, ANSI / IEEE 770 X3.97-1983. However, for localization, Pascal allows input and output of numbers in the appropriate international format locales.

For example, the standard specifies a period (`.`) as the decimal unit in the floating-point representation. Consider the following program, which prints a floating-point value:

```
program sample(output);

var r : real := 1.2;

begin
    writeln(r);
end.
```

When you compile and run the program on the internationalized Pascal compiler, the output is:

```
1.20000000000000e+00
```

If you reset your system locale to, for example, France, and rerun the program, the output would be slightly different. Namely, Pascal would replace the period with a comma, the French decimal unit.

Locale

You can change your application from one native language to another by setting the locale. For information on this and other native language support features, see the Solaris documentation on internationalization.

Licensing

This compiler uses network licensing, as described in the manual, *Sun WorkShop Installation and Licensing Guide*.

When you invoke the compiler, if a license is available, the compiler starts. If no license is available, your request for a license is put on a queue, and your compile job continues when a license becomes available. A single license can be used for any number of simultaneous compiles by a single user on a single machine. There are two licensing-related options:

- `-noqueue`—Does not queue request if no license is available.
- `-xlicinfo`—Returns information on the status of licensing.

The `-xlicinfo` option does not check out a license.

For details on how to obtain a license—where to call, what information to have ready—refer to the manual, *Sun WorkShop Installation and Licensing Guide*.

Pascal Programs 2

This chapter cites two examples that illustrate how to compile and execute a program. It also explains how to use the traceback facility to find out why a program fails. The sections are:

Building a program with SPARCompiler Pascal requires three steps:

1. **Writing a program in Pascal using an editor, and saving it in a file with a** `.p` **or** `.pas` **suffix**

2. **Compiling the** <*name*>`.p` **or** <*name*>`.pas` **file using the** `pc` **command**

3. **Executing the program by typing the name of the executable file at the system prompt**

A Simple Pascal Program

The following is a simple Pascal program that converts temperatures from degrees Fahrenheit to degrees Celsius. Use an editor to type the code on your system and save it in a file called `temp.p`.

```
program temperature(output) ;

{ Program to convert temperatures from
 Fahrenheit to Celsius. }

const
    MIN = 32 ;
    MAX = 50 ;
    CONVERT = 5 / 9 ;

var
    fahren: integer ;
    celsius: real ;

begin
    writeln('Fahrenheit     Celsius') ;
    writeln('----------     -------') ;
    for fahren := MIN to MAX do begin
        celsius := CONVERT * (fahren - 32) ;
        writeln(fahren: 5, celsius: 18: 2) ;
 end ;
end.
```

Compiling the Program

Now compile the program with `pc`, the Pascal compiler, by typing at the system prompt:

```
hostname% pc temp.p
```

Pascal names the compiled version of the program `a.out` by default.

Running the Program

To run the program, enter `a.out` at the prompt. The output of `temp.p` is then displayed:

```
hostname% a.out
  Fahrenheit     Celsius
  ----------     -------
     32          0.00
     33          0.56
     34          1.11
     35          1.67
     36          2.22
     37          2.78
     38          3.33
     39          3.89
     40          4.44
     41          5.00
     42          5.56
     43          6.11
     44          6.67
     45          7.22
     46          7.78
     47          8.33
     48          8.89
     49          9.44
     50          10.00
```

Renaming the Executable File

It is inconvenient to have the result of every compilation in a file called `a.out`. If such a file already exists, it is overwritten. You can avoid this in either of the two following ways:

- Change the name of `a.out` after each compilation with the `mv` command:

  ```
  hostname% mv a.out temp
  ```

- Use the compiler `–o` option to name the output executable file. This example places the executable code in the file `temp`:

  ```
  hostname% pc –o temp temp.p
  ```

Now run the program by typing the name of the executable file. The output follows:

```
hostname% temp
Fahrenheit     Celsius
----------     -------
   32          0.00
   33          0.56
   34          1.11
    .            .
    .            .
    .            .
```

An Interactive Pascal Program

In Pascal, the predefined file variable, `input`, is equivalent to the operating system standard input, `stdin`. Similarly, the file variable, `output`, is equivalent to the standard output, `stdout`.

Following is a Pascal program that copies `input` to `output`. Use an editor to type the code on your system and store it in a file called `copy.p`:

```
program copy(input, output);

{ This program copies input to output. }

var
    c: char;

begin
    while not eof do begin
      while not eoln do begin
          read(c);
          write(c)
      end;
      readln;
      writeln
    end
end. { copy }
```

Compiling the Program

Use the `pc` command to compile the program and store it in the executable file `copy`. Here is the command format:

```
hostname% pc -o copy copy.p
```

Running the Program

Because the standard `input` and standard `output` default to the keyboard and the display screen, respectively, the program simply echoes on the screen each line you type on the keyboard. The program terminates when you type the end-of-file (Control-d) character at the beginning of a line. Try it:

```
hostname% copy<Return>
Hello, are you listening?<Return>
Hello, are you listening?
Goodbye, I must go now.<Return>
Goodbye, I must go now.
(Control-d)
hostname%
```

Redirecting I/O

To write the output to a file instead of to the display screen, use the redirection operator, >, followed by a file name. For instance, to write to a file called `data`, enter the following:

```
hostname% copy > data<Return>
Hello, are you listening?<Return>
Goodbye, I must go now.<Return>
(Control-d)
hostname%
```

Using the same program, but with the < operator to redirect input, you can print the file on the display screen:

```
hostname% copy < data<Return>
Hello, are you listening?
Goodbye, I must go now.
```

Using a File Name as a File Variable

You can redirect the program input or output to files by listing the files as `file` variables in the program statement. The Pascal library associates each (input, output) file variable with a file named in the program statement. For example, `copy2.p` lists `data` as the input file variable and `output` as the output file variable:

```
program copy2(data, output);

{ This program redirects input. }

var
    c: char;
    data: text;

begin
    reset(data);
      while not eof(data) do begin
      while not eoln(data) do begin
          read(data, c);
          write(c)
      end;
      readln(data);
      writeln
    end
end. { copy2 }
```

Assuming that the file `data` is still in the current directory, you can compile and run the program as follows:

```
hostname% pc -o copy2 copy2.p
hostname% copy2
Hello, are you listening?
Goodbye, I must go now.
```

Where Did My Program Fail?

SPARCompiler Pascal can trace why a program failed; its traceback utility finds the routine that triggers the error.

Using Pascal Traceback

Pascal traceback installs signal handlers on selected signals and dumps a backtrace when those signals are caught. The backtrace shows the chain of calls leading from the routine in which the error occurred, all the way back to the main program.

Pascal catches the following set of signals:

SIGQUIT	SIGIOT	SIGFPE	SIGSYS	SIGTERM
SIGILL	SIGABRT	SIGBUS	SIGPIPE	SIGLOST
SIGTRAP	SIGEMT	SIGSEGV		

See the `signal`(3) `man` page for further information on these signals.

After the system produces the traceback, it continues with whatever action it would have taken if the interposer had *not* been in place, including calling a user signal handler that was previously set.

The traceback facility uses the debugger `dbx`. To obtain a traceback, SPARCworks must be installed on your system, and the directory containing `dbx` must be in your `PATH` environment variable. If the traceback routine cannot find `dbx`, it does not produce the traceback.

Use the `-notrace` command-line option to disable traceback.

Using a Sample Program with Segmentation Violation

A segmentation violation occurs when your program tries to reference memory outside your address space. The operating system detects this action and generates an error message. Following is an example program, `SegViol.p`, which contains a segmentation violation:

```
program SegmentationViolation;
type
  Pinteger = ^integer;

procedure ErrorInHere;
var
  IntVar:  integer;
  NullPtr: Pinteger;
begin
  NullPtr := nil;
  { Next statement causes a SEGV }
  IntVar := NullPtr^;
end;

procedure Call1;
  procedure Call2;
  begin
    ErrorInHere;
  end;
begin
  Call2;
end;

begin
  Call1;
end.
```

Compiling and Running the Program

When you compile and run the program, you receive output similar to the following. The first line indicates the name of the offending signal—in this case, a segmentation violation.

```
hostname% pc SegViol.p
hostname% a.out

*** a.out terminated by signal 11: segmentation violation
*** Traceback being written to a.out.trace
Abort (core dumped)

hostname% more a.out.trace

*** Stacktrace of a.out
*** Program terminated due to segmentation violation
  [3] __PC0__sigdie(0xb, 0xefffedf0, 0xefffec30, 0x0, 0x1, 0x0), at 0x12128
  ---- called from signal handler with signal 11 (SIGSEGV) ------
  [4] ErrorInHere(), at 0x115ec
  [5] Call2(0xefffefc8, 0xefffefa8, 0xefffef88, 0x0, 0xef74dd58, 0x0), at 0x11624
  [6] Call1(0x25400, 0x25800, 0x25b80, 0x25b80, 0x3, 0x0), at 0x11660
  [7] program(0x1, 0xeffff0fc, 0x4, 0xef7d0000, 0x2, 0xef74dae8), at 0x116a4
  [8] main(0x1, 0xeffff0fc, 0xeffff104, 0x25000, 0x0, 0x0), at 0x116e0
detaching from process 17266
```

In this example, `ErrorInHere` reported the error. The `ErrorInHere` procedure was called by `Call1.Call2`, which was in turn called by the main program. Routine names, such as `Call1.Call2`, indicate a nested routine. If Pascal cannot find the name of a routine, for example, because the executable file has been stripped, it prints the hex address.

Using the `-g` *Option*

If you compile the program with the `–g` option, the traceback also reports the arguments, the line number, and the file name of each routine.

Try compiling `SegViol.p` with `–g`:

```
hostname% pc -g SegViol.p
hostname% a.out

*** a.out terminated by signal 11: segmentation violation
*** Traceback being written to a.out.trace
Abort (core dumped)

hostname% more a.out.trace

*** Stacktrace of a.out
*** Program terminated due to segmentation violation
  [3] __PC0__sigdie(0xb, 0xefffedf0, 0xefffec30, 0x0, 0x1, 0x0), at 0x12128
  ---- called from signal handler with signal 11 (SIGSEGV) ------
  [4] ErrorInHere(), line 12 in “SegViol.p”
  [5] Call2(), line 18 in "SegViol.p"
  [6] Call1(), line 21 in "SegViol.p"
  [7] program(), line 25 in "SegViol.p"
detaching from process 17285
```

The program prints the ASCII values of character variables.

If you compile some modules with `–g` and others without, the line numbers may not be accurate for all the routines.

The Pascal Compiler 3

The name of the Sun Workshop Compiler Pascal is `pc`. If you give `pc` a file name as an argument, and the file name ends with `.p` or `.pas`, `pc` compiles the file and leaves the result in an executable file, called `a.out` by default.

The syntax of this command is:

`pc [`*options*`]` *filename*

This chapter contains the following sections:

`pc` *Version Number*

To identify the version number of `pc` when you compile your program, call the compiler with the `-V` option. This option instructs the compiler to produce output that identifies the versions of all the programs used in compiling, the compiler itself, the code generator, and so on.

To identify the version number given an executable or object file created by the Pascal compiler, use the following command.

```
hostname% mcs -p a.out | grep Pascal
SC4.2 18 Sept 1996 Pascal 4.2
```

Compile and Link Sequence

You can compile the file `any.p` with the following command-line:

```
hostname% pc any.p
```

This command actually invokes the compiler driver, which calls several programs or passes of the program, each of which processes the program. The output of each pass is the input to the next one.

After several passes, the object file `any.o` is created. An executable file is then generated with the default name `a.out`. Finally, the file `any.o` is removed.

`pc` calls:

- `cpp`, the C preprocessor, or `cppas`, the preprocessor used when you use the `-xl` option
- `pc0`, the Pascal front end
- The global optimizer if you use the `-O` option
- `cg`, the code generator, which generates the relocatable object file
- `pc3`, which checks for conflicts in symbol names
- `ld`, the linker, which generates the executable files using any libraries necessary to resolve undefined symbols

The above is the default action of `pc`; some compiler options change what `pc` calls.

Figure 3-1 shows the sequence of events when you invoke `pc`.

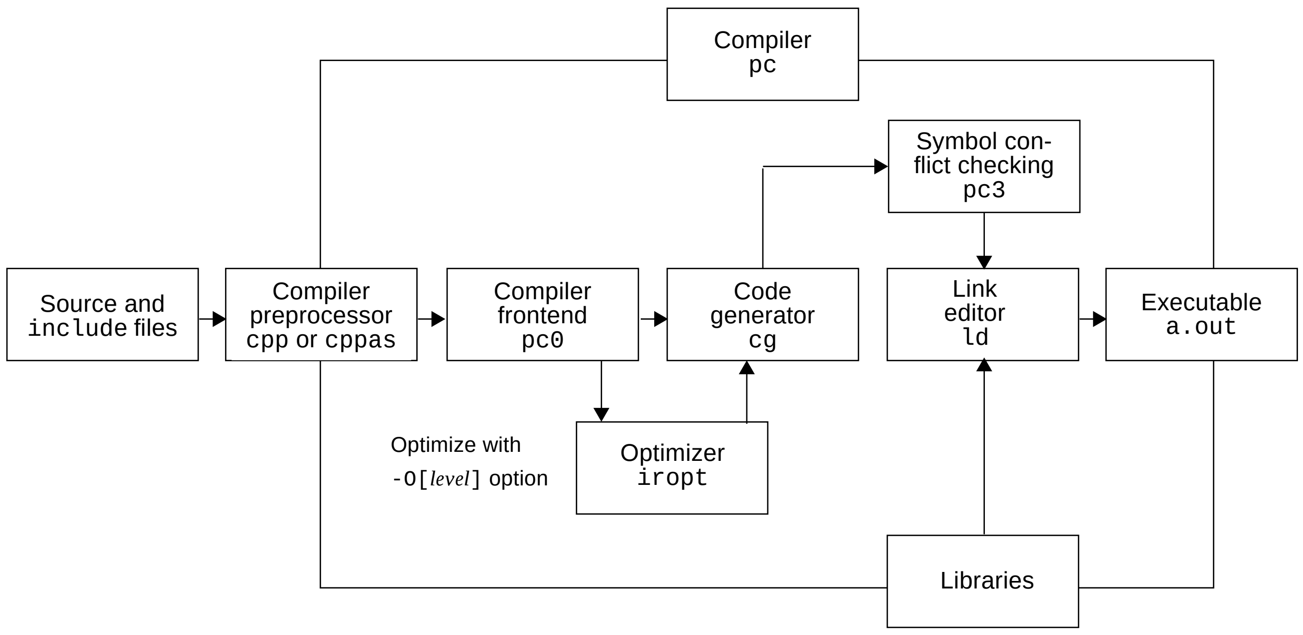


Figure 3-1 Organization of Pascal Compilation

Language Preprocessor

The `cpp`(1) program is the C language preprocessor. The compiler driver `pc` normally calls `cpp`(1) during the first pass of a Pascal compilation. If you use the `–xl` switch, `pc` calls the alternate preprocessor `cppas`. Then `cpp`(1) and `cppas` operate on files that contain the extension `.p` or `.pas`.

You can give directives to `cpp`(1) or `cppas` to define constants, conditionally compile parts of your program, include external files, and take other actions. For example, the following program shows the use of an `include` directive, which asks `cpp`(1) to copy the named file into the program before compilation.

```
program showinclude;
#include "file.i"
begin
...
end.
```

See the man page for `cpp`(1) for information on its directives and other features. Appendix A, "Pascal Preprocessor," describes `cppas`.

File Name Extensions Accepted By `pc`

Pascal source files generally use the extension `.p`. The compiler recognizes other file name extensions. Table 3-1 lists the most important extensions.

The table notes that `pc` can produce assembler source files as well as unlinked object files. In each case, you can pass these partially compiled files to `pc`, which then finishes the compilation, linking, and loading.

Table 3-1 File Name Suffixes Recognized by Pascal

Suffix	**Description**
`.p`	Usual extension for Pascal source files.
`.pas`	Valid extension for a Pascal source file. The extension instructs `pc` to put object files in the current directory. The default name of the object file is the name of the source file, but with a `.o` suffix.
`.pi`	Default extension for Pascal source files that have been processed by the Pascal preprocessor (either `cpp` or `cppas`).
`.s`	Extension for assembler source files that are produced when you call `pc` with the `-S` option.
`.o`	Extension for object files that are generated by the compiler when you call `pc` with the `-c` option.

Option-Passing on the Command-Line

To pass an option on the command-line, use a dash (`-`) followed by the option name. In some cases, you must supply additional information, such as a file name. For example, this command activates the listing option `-l`, which is off by default:

```
hostname% pc -l rmc.p
```

The following command causes the generated object file to be named `rmc` instead of the default, `a.out`.

```
hostname% pc -o rmc rmc.p
```

Option-Passing in the Program Text

Some options can be passed to the compiler in program text as well as on the command-line. With this facility, you can use different option values for different parts of a program.

Here are four examples of how options can be passed in program text:

```
{$P+}
{$H*}
(*$I-*)
{$l+,L-,n+}
```

Table 3-2 shows the options that can be passed in program text.

Table 3-2 Options That Can Be Passed in Program Text

Option	Description
`b`	Uses buffering of the file `output`.
`C`	Uses runtime checks (same as `t`).
`calign`	Uses C data formats.
`H`	Uses check heap pointers.
`l`	Makes a listing.
`L`	Maps identifiers and keywords to lowercase.
`p`	Uses statement limit counting (different from command-line `p`[1]). See `stlimit` in the *Pascal 4.2 Reference Manual*.

Table 3-2 Options That Can Be Passed in Program Text *(Continued)*

Option	Description
P	Uses partial evaluation of boolean expressions.
t	Uses runtime checks (same as C, but different from the command-line t[1]).
u	Trims to 72-character line (not usable on command-line).
w	Prints warning diagnostics.

1. The options p and t are different when they are used within program text and when they are used on the command-line because they are received directly by pc0 when they are used in program text, while the compiler driver gives them to other compiler passes when they are given on the command-line. If you want to set them on the command-line and also want them to have the same effect as passing them in program text, use the Qoption command-line option to pass them directly to pc0.

You set options within comments, which can be delimited by either { and } or (* and *). The first character in the comment must be the $ (dollar sign). $ must be immediately followed by an option letter. Next must be either +, -, or *.

If you want to set more than one option, you can insert a comma after the first option, followed by another option letter and +, -, or *. You can set any number of options on a single line. There must be no embedded spaces. You can place spaces and other ordinary comment text after the last +, -, or *.

The new option setting takes effect at the next noncomment token.

The symbols + (plus sign) and - (minus sign) turn the option on and off, respectively. To understand the symbol *, you must know how options work in Pascal.

Except for b, each option in Table 3-2 has a current value and a “first in, last out” stack of up to 16 values. Again, except for b, each option can be on or off.

When you begin compiling a program, each stack is empty and each option has an initial current value, which may be the option default value or may have been set on the command line.

When the compiler encounters an option followed by...	This is what happens...
+	The current value is pushed onto the stack, and the current value becomes ON.
-	The current value is pushed onto the stack, and the current value becomes OFF.
*	The last value is popped off the stack and becomes the current value.

If no values have been pushed onto the stack, the effect of * is undefined.

Figure 3-2 illustrates how options are passed in program text.

Program:

```
program options (output);
begin
{$l+ Turns on listing}
    writeln ('After $l-');
{$l- Turns off listing}
{Notice that this line prints.}
    writeln ('After $l+');
{$l* Turns listing on again}
{Notice that this line does not print.}
    writeln ('After $l*')
end.
```

Output:

```
hostname% pc options.p
Fri Mar 1 17:33:18 1995 options.p:
 4  writeln ('After $l-');
 5 {$l- Turns off listing}
 6 {Notice that this line prints.}
 10 writeln ('After $l*')
 11 end.
```

Figure 3-2 Options in Program Text

Options

This section describes all the pc command options in alphabetical order. As described elsewhere in the documentation, Pascal 4.2 runs only in SPARC Solaris 2.x environments, which is where these options are designed for.

In general, processing of the compiler options is from left to right, so selective overriding of macros can be done. This rule does not apply to linker options.

–a

The –a option is the old style of basic block profiling for tcov. See -xprofile=tcov for information on the new style of profiling and the tcov(1) man page for more details. Also see the manual, *Profiling Tools*.

The –a option inserts code to count how many times each basic block is executed. It also calls a runtime recording mechanism that creates a .d file for every .p file at normal termination. The .d file accumulates execution data for the corresponding source file. The tcov(1) utility can then be run on the source file to generate statistics about the program.

If set at compile-time, the TCOVDIR environment variable specifies the directory of where the .d files are located. If this variable is not set, then the .d files remain in the same directory as the .f files.

The -xprofile=tcov and the -a options are compatible in a single executable. That is, you can link a program that contains some files which have been compiled with -xprofile=tcov, and others with -a. You cannot compile a single file with both options.

–B*binding*

The –B option specifies whether libraries for linking are static (not shared, indicated with -Bstatic), or dynamic (shared, indicated with -Bdynamic).

Link editing is the set of operations necessary to build an executable program from one or more object files. Static linking indicates that the results of these operations are saved to a file. Dynamic linking refers to these operations when performed at runtime. The executable that results from dynamic linking appears in the running process, but is not saved to a file.

–b

It is inefficient for Pascal to send each character to a terminal as it generates its output. It is even less efficient if the output is the input of another program, such as the line printer daemon, lpr(1).

To gain efficiency, Pascal buffers output characters; it saves the characters in memory until the buffer is full and then outputs the entire buffer in one system interaction. By default, Pascal output is line-buffered.

The `-b` option on the command-line turns on block-buffering with a block size of 1,024. You cannot turn off buffering from the command-line.

If you give the `-b` option in a comment in the program, you can turn off buffering or turn on block buffering. The valid values are:

`{$b0}`	No buffering
`{$b1}`	Line buffering
`{$b2}`	Block buffering. The block size is 1,024.

Any number greater than 2 (for example, `{$b5}`) is treated as `{$b2}`. You can only use this option in the main program. The block buffering value in effect at the end of the main program is used for the entire program.

`-C`

The `-C` option enables runtime checks that verifies that:

- Subscripts and subranges are in range.
- The number of lines written to output does not exceed the number set by the `linelimit` procedure. (See the *Pascal 4.2 Reference Manual* for information on `linelimit`.)
- Overflow, underflow, and divide-by-zero do not exist.
- The `assert` statement is correct. (See the *Pascal 4.2 Reference Manual* for information on `assert`.)

If you do not specify `-C`, most runtime checks are disabled, and `pc` treats the `assert` statement as a comment and never uses calls to the `linelimit` procedure to halt the program. However, divide-by-zero checks are always made.

The `-V0` and `-V1` options implicitly turn on `-C`.

`-c`

The `-c` option instructs the compiler *not* to call the linker, `ld`(1). The Pascal compiler, `pc`, leaves a `.o` or object file for each source file. Without `-c`, `pc` calls the linker when it finishes compilation, and produces an executable file, called `a.out` by default.

-calign

The -calign option instructs the compiler to allocate storage in records the same way as the C compiler allocates structures. See the *Pascal 4.2 Reference Manual* for details of how data is aligned with and without -calign.

You can use calign within a program as well as on the command-line. However, calign only has an effect in the type and var sections of the program. Any types you define when calign is on use C-style alignment whenever you define variables of that type.

–cg89

The –cg89 option is a macro for:
-xarch=v7 -xchip=old -xcache=64/32/1.

–cg92

The –cg92 option is a macro for:
-xarch=v8 -xchip=super -xcache=16/32/4:1024/32/1

-cond

You can only use this option when you also use the -xl option.

The –cond option instructs pc to compile the lines in your program that begin with the %debug compiler directive. If you compile your program without –cond, pc treats lines with the %debug directive as comments.

–xl runs your program through the preprocessor cppas, which handles the Apollo DOMAIN®-style Pascal compiler directives, such as %debug.

See Appendix A, “Pascal Preprocessor,” for a complete description of conditional variables, cppas, and compiler directives.

–config

You can only use this option when you also use the -xl option.

The –`config` option sets a conditional variable to `true`. You can only use this option when you use the preprocessor `cppas`, which is invoked when you use the `-xl` option.

Pascal supports the –`config` option with only one value. For example, Pascal accepts –`config one`, but not –`config one two`. To specify more than one variable, use multiple –`config` options on the command-line.

If you use `-config` but do not give a variable, the value of the predefined conditional variable `%config` is set to `true`.

–`xl` runs your program through the preprocessor `cppas`, which handles the Apollo DOMAIN-style Pascal compiler directives, such as `%config`.

See Appendix A, "Pascal Preprocessor," for a complete description of conditional variables, `cppas`, and compiler directives.

–D*name*[=*def*]

The –`D` option defines a symbol name to the C preprocessor, `cpp`. It is equivalent to using the `#define` statement in your program. If you do not include a definition, *name* is defined as 1. See `cpp`(1) for more information.

If you use this option with the `-xl` option, `-D` is equivalent to using the `%config` directive in your program.

–dalign

The –`dalign` option instructs the compiler to generate double load and store instructions wherever possible for faster execution. All double-typed data become double-aligned, so do not use this option when correct alignment is not ensured.

-dn

The `-dn` option specifies static linking in the link editor.

-dryrun

The –dryrun option instructs the compiler to show, but not execute, the commands constructed by the compilation driver. You can then see the order of execution of compiler passes without actually executing them.

-dy

The -dy option specifies dynamic linking in the link editor.

–fast

The –fast option includes -fns -ftrap=%none; that is, it turns off all trapping. In previous releases, the -fast macro option included -fnonstd; now it does not.

-fast includes -native in its expansion.

The code generation option, the optimization level, and using inline template files can be overridden by subsequent switches. For example, although the optimization part of -fast is -O2, the optimization part of -fast -O3 is -O3. Do not use this option for programs that depend on IEEE standard exception handling. It can produce different numerical results, premature program termination, or unexpected SIGFPE signals.

Note – The criteria for the -fast option vary with the compilers from SunSoft: C, C++, FORTRAN 77, Fortran 90, and Pascal. See the appropriate documentation for the specifics.

-flags

The –help or -flags option lists and summarizes all available options.

-fnonstd

The -fnonstd option causes nonstandard initialization of floating-point arithmetic hardware. By default, IEEE 754 floating-point arithmetic is nonstop, and underflows are gradual. (See the *Numerical Computation Guide* for details.) The –fnonstd option causes hardware traps to be enabled for floating-point overflow, division by zero, and invalid operation exceptions. These hardware traps are converted into `SIGFPE` signals, and if the program has no `SIGFPE` handler, it terminates with a memory dump.

`-fnonstd` also causes the math library to be linked in by passing `-lm` to the linker.

This option is a synonym for `-fns -ftrap=common`.

–fns

The `-fns` option turns on the SPARC non-standard floating-point mode.

The default is the SPARC standard floating-point mode.

If you compile one routine with `-fns`, then compile all routines of the program with the `–fns` option; otherwise, unexpected results may occur.

–fround=*r*

The `-fround=`*r* option sets the IEEE 754 rounding mode that is established during program initialization.

r must be one of: `nearest`, `tozero`, `negative`, `positive`.

The default is `-fround=nearest`.

The meanings are the same as those for the `ieee_flags` subroutine.

If you compile one routine with `-fround=`*r*, compile all routines of the program with the same `–fround=`*r* option; otherwise, unexpected results may occur.

–ftrap=*t*

The `-ftrap=`*t* option sets the IEEE 754 trapping mode in effect at startup.

t is a comma-separated list of one or more of the following: `%all`, `%none`, `common`, `[no%]invalid`, `[no%]overflow`, `[no%]underflow`, `[no%]division`, `[no%]inexact`. The default is `-ftrap=%none`.

This option sets the IEEE 754 trapping modes that are established at program initialization. Processing is left-to-right. The *common* exceptions, by definition, are invalid, division by zero, and overflow.

Example: `-ftrap=%all,no%inexact` means set all traps, except `inexact`.

The meanings are the same as for the `ieee_flags` function, except that:

- `%all` turns on all the trapping modes.
- `%none`, the default, turns off all trapping modes.
- A `no%` prefix turns off that specific trapping mode.

If you compile one routine with `-ftrap=`*t*, compile all routines of the program with the same `-ftrap=`*t* option; otherwise, unexpected results may occur.

-G

The `-G` option builds a shared library. All object files specified with this command option should have been compiled with either the `-pic` or the `-PIC` option.

–g

The `–g` option instructs `pc` to produce additional symbol table information for `dbx` and `debugger`. With `-g`, the incremental linker, `ild`, is called, instead of the linker, `ld`.

You can compile using both the `-g` and `-O` options. However, there are some side effects:

- The `next` and `step` commands do not work, but the `cont` command does.
- If you have makefiles that rely on `-g` overriding `-O`, you must revise those files.
- If you have makefiles that check for a warning message that `-g` overrides `-O`, you must revise those make files.

Note – Special case: `-04 -g`. The combination `-04 -g` turns off inlining that you usually get with `-04`.

–H

The –H option instructs pc to compile code to perform range-checking on pointers into the heap. This option is implicitly turned on by the –V0 and –V1 options.

-h*name*

The -h*name* option names a shared dynamic library and provides a way to have versions of a shared dynamic library.

This is a loader option, passed to ld. In general, the name after -h should be exactly the same as the one after -o. A space between the -h and *name* is optional.

The compile-time loader assigns the specified name to the shared dynamic library you are creating. It records the name in the library file as the intrinsic name of the library. If there is no -h*name* option, then no intrinsic name is recorded in the library file.

Every executable file has a list of needed shared library files. When the runtime linker links the library into an executable file, the linker copies the intrinsic name from the library into that list of needed shared library files. If there is no intrinsic name of a shared library, then the linker copies the path of the shared library file instead.

–help

The –help or -flags option lists and summarizes all available options.

–I*pathname*

The –I option gives the preprocessor additional places to look for #include and %include files. For example,

```
hostname% pc -I/home/incfiles -I/usr/incfiles program.p
```

The preprocessor searches for #include and %include files in this order:

1. In /opt/SUNWspro/SC4.0/include/pascal

2. In the directory containing the source file, except when you use the `#include` <*file*> form, in which case this directory is not searched
3. In directories named with `–I` options, if any, in left to right order
4. In `/usr/include`

`–i` *name*

The `–i` option produces a listing for the specified procedure, function, `#include`, or `%include` file. For example, this command instructs the compiler to make a listing of the routines in the file `scanner.i`.

```
hostname% pc –i scanner.i program.p
```

See `cpp`(1), or or Chapter 5, “Separate Compilation,” for information on `include` files.

`-keeptmp`

The `-keeptmp` option keeps temporary files that are created during compilation, so they are retained instead of being deleted automatically.

`–L`

The `–L` option maps all keywords and identifiers to lowercase. In Pascal, uppercase and lowercase are not interchangeable in identifiers and keywords. In standard Pascal, the case is insignificant outside of character strings and character constants. The `–L` option is most useful for transporting programs from other systems. See also the `–s` option.

–l

The –l option produces a listing of the program. For example:

```
hostname% pc -l random.p
Pascal PC -- Version SC4.0 09 Jan 1995 Pascal 4.0

Mon Jan 09 09:04 1995 random.p:
    1   program random_number(output);
    2   var
    4       i: integer;
    5       x: integer;
    6
    7   begin
    8       for i := 1 to 5 do begin
    9               write(trunc(random(x) * 101))
    10      end;
    11      writeln
    12      end.
```

The first line identifies the version of the compiler you are using. The next line gives the modification time of the file being compiled. The remainder of the listing is the source program.

-L*directory*

The -L*directory* option adds *directory* to the ld library search path for ld.

-libmieee

Forces IEEE 754 style return values for math routines in exceptional cases. In such cases, no exception message is printed, and errno is not set.

–libmil

The –libmil option instructs the compiler to select the best inline templates for the floating-point option and operating system release available on this system.

–l*lib*

The –l*lib* option links ld(1) with the object library, *lib*.

Do not use the -lucb option because Pascal is not compatible with the object library, libucb.

–misalign

The –misalign option allows for misaligned data in memory. Use this option only if you receive a warning message that your data is misaligned.

With the –misalign option, pc generates much slower code for references to formal parameters. If possible, recode the indicated section instead of recompiling your program with this option.

-mt

The -mt option uses multithread-safe libraries, eliminates conflicts between threads, so that Pascal library routines can be safely used in a multiprocessing environment.

The MT-safe library for Pascal is called libpc_mt.

On a single-processor system, the code that is generated with this option runs more slowly; the degradation in performance is usually insignificant, however.

Refer to the *Multithreaded Programming Guide* in the Solaris documentation for more information.

–native

The –native option causes pc to generate code for the best floating-point hardware available on the machine you are compiling on.

The -fast macro includes -native in its expansion.

This option is a synonym for -xtarget=native.

-nolib

The -nolib option instructs the compiler *not* to link any libraries by default—that is, no -l options are passed to ld. Normally, the pc driver passes -lc to ld.

When you use -nolib, pass all -l options yourself. For example, the following command links libm statically and the other libraries dynamically:

```
hostname% pc -nolib -Bstatic -lm -Bdynamic -lc test.p
```

–nolibmil

The –nolibmil option instructs the compiler to reset –fast so that it does *not* include inline templates. Use this option *after* the –fast option, as in:

```
hostname% pc –fast –nolibmil myprog.p
```

-non_init[=yes|no]

This option turns on *early warnings* of the use of uninitialized local variables. That is, if the pc compiler is given the -non_init (or -non_init=yes) option, it issues a warning at the time it finds an uninitialized local variable. This warning is issued even if the variable is initialized later in the routine.

The -non_init=yes option invokes extra compilation processing overhead for checking statements that change program control flow such as calls, gotos, NEXT, and other such possible elements. These checks are performed every time the compiler is run with this option.

By default (or if given the -non_init=no option), the pc compiler issues late warnings. That is, at the *end* of each function it checks the use of each local variable and issues a warning for each local variable that was used but never initialized, as discussed in “Uninitialized Variables” on page 220. The default -non_init=no option avoids the extra compilation processing overhead associated with -non_init=yes option.

The default late warnings are actually useful only for local variables that are *never* assigned. Commonly, however, a local variable is inappropriately first used and later initialized, as shown in the following example:

The Pascal program, `iw1.p`, showing use of local variable before it is initialized

```
program iw1.p;
procedure q;
var x: integer;
begin
        writeln(x);
        x:= 1;
end;
begin
        q;
end.
```

In this `iw1.p` program, the local variable `x` is used at line 5 before initialization at line 6. If `iw1.p` is compiled without the `-non_init` option, the compiler does not issue a warning because the local variable is finally initialized.

However, if the `pc` compiler is given the `-non_init` option, it issues a warning immediately upon finding an uninitialized variable, as follows:

```
hostname% pc -non_init iw1.p
w 18280 line 5 - variable x is used but never set
hostname%
```

Given the `-non_init` option, the compiler tracks the program control flow, and issues a warning each time it encounters an uninitialized local variable.

Two example Pascal programs with control flow changes are given in following boxes. For these examples, `-non_init` warnings are not issued because the compiler sees they would make no sense or the control flow is changed.

The Pascal program, iw2.p, showing change in control flow

```
program iw2;
procedure q;
label 1;
var x: integer;
begin
          goto 1;
          writeln(x);
       1: x:= 1;
end;
begin
   q;
end.
```

Given the -non_init option for compiling iw2, the pc compiler issues no non-initialized-variable warning because the goto statement changes the control flow of the program:

```
hostname% pc -non_init iw2.p
w 18630 line 7 - Unreachable statement
hostname%
```

The following example program, iw3.p, uses indirect variable p^ (p as a local pointer variable). However, the order of statements is wrong. That is, p^ is undefined until line 7, after the execution statements at lines 5 and 6.

The Pascal program, iw3.p, showing incorrect use of indirect variable

```
program iw3;
procedure q;
var p: ^integer;
begin
   writeln(p^);
   new(p);
   p^:= 1;
end;
begin
   q;
end.
```

The -non_init option sets compile-time checking for appropriate usage of indirect variables. Given the following command with the -non_init option, the compiler displays a warning message resulting from compiling the error in the iw3.p program:

```
hostname% pc -non_init iw4.p
w 18280 line 6 - variable p is used but never set
hostname%
```

Use With -Rw

The -non_init option can be used together with the -Rw option, which provides warnings for record fields. Compiling with a combination of the -non_init and -Rw options can capture most usages of uninitialized whole and component variables.

Pascal program, iw_rw.p, showing incorrect use of indirect variable and whole and component variables

```
program iw_rw;
procedure q;
type t = record a,b: integer end;
var p: ^integer;
    r: t;
    i: integer;
begin
        writeln(p^);
        writeln(r.a);
        writeln(i);
        new(p);
        r.a:= 1;
        i:= 0;
end;

begin
    q;
end.
```

Compiling iw_rw with a combination of the -non_init and -Rw options captures the usages of uninitialized whole and component variables:

```
hostname% pc -non_init -Rw iw_rw.p
w 18280 line 8 - variable p is used but never set
w 18280 line 9 - variable r is used but never set
w 18280 line 9 - field r.a is used but never set
w 18280 line 10 - variable i is used but never set
hostname%
```

Lack of Effect of `-non_init`

The `-non_init` option has no effect in the following two cases:

- Compiling with the `-Z` option, which initializes all local variables to zero, renders `-non_init` irrelevant because all local variables are initialized on calling their host routine.
- `-non_init` checks are not performed for global variables defined at the outermost level of a program or module because each global variable can potentially be initialized in another separate module linked to the executable program.

`-noqueue`

The `-noqueue` option instructs the compiler not to queue this compilation if a license is not available. Under normal circumstances, if no license is available, the compiler waits until one becomes available. With this option, the compiler returns immediately.

`-notrace`

The `-notrace` option disables runtime traceback. It is only effective when compiling the main program.

`–O[`*level*`]`

The `–O` option instructs the compiler to run the compiled program through the object code optimizer. The `–O` option also calls the `–P` option, which ensures `boolean` expressions are only evaluated as much as needed to determine the result. This process causes an increase in compile time in exchange for a decrease in compiled code size and execution time.

There are four levels of optimization. You indicate the level by specifying a digit from 1 to 4 after the `–O` option. If you leave out the digit, the optimization level defaults to `–O2`.

The level numbers are interpreted as follows:

`–O`

This is the most likely level of optimization to give fastest performance for most reasonable applications. The default is `–O2`.

`–O1, -xO1`

This is the minimum amount of optimization (peephole) and is postpass assembly-level. Do not use `–O1` unless `-O2` and `-O3` result in excessive compilation time or shortage of swap space.

`–O2, -xO2`

This is the basic local and global optimization—induction-variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, control-flow optimization, tail-recursion elimination, dead-code elimination, and tail-call elimination.

Level `-O2` does not optimize references to or definitions of external or indirect variables. This level is the appropriate level for device drivers and programs that modify external variables from within signal handlers.

`–O3, -xO3`

Same as `-O2`, but optimizes the uses and definitions of external variables. Level `-O3` does not trace the effects of pointer assignments. Do not use Level `-O3` when compiling device drivers or programs that modify external variables from within signal handlers.

–O4, -xO3
: Same as -O3, but traces the effects of pointer assignments and gathers alias information. Do not use Level -O4 when compiling device drivers or programs that modify external variables from within signal handlers.

-O5, -xO5
: Generates the highest level of optimization. This level uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time.

  Optimization at this level is more likely to improve performance if it is done with profile feedback. See -xprofile and -xO5.

Note – Levels -O3 and -O4 may result in an increase in the size of the executables. When optimizing for size, use level -O2. For most programs, –O4 is faster than –O3, which is faster than –O2, which is faster than –O1. However, in a few cases –O2 may be faster than the others, and –O3 may be faster than –O4. You can try compiling with each level to see if you have one of these rare cases.

If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization, then resumes subsequent procedures at the original level specified in the –O command-line option.

If you optimize at –O3 or –O4 with very large procedures (thousands of lines of code in a single procedure), the optimizer may require an unreasonable amount of memory. Such cases may result in degraded machine performance.

You can prevent this from happening in the C shell by limiting the amount of virtual memory available to a single process. To do this, use the limit command (see csh(1)).

For example, to limit virtual memory to 16 megabytes:

```
hostname% limit datasize 16M
```

This command causes the optimizer to try to recover if it reaches 16 megabytes of data space.

This limit cannot be greater than the machine's total available swap space, and in practice, should be small enough to permit normal use of the machine while a large compilation is in progress. For example, on a machine with 32 megabytes of swap space, the command `limit datasize 16M` ensures that a single compilation never consumes more than half of the machine's swap space.

The best setting of data size depends on the degree of optimization requested and the amount of real memory and virtual memory available. To find the actual swap space:

```
hostname% /usr/sbin/swap -s
```

To find the actual real memory:

```
hostname% /usr/sbin/prtconf | grep Memory
```

`–o` *filename*

The `–o` option instructs the compiler to name the generated executable, *filename*. The default file name for executable files is `a.out`; for object files, it is the source file name with a `.o` extension. For example, the following command stores the executable in the file, `myprog`:

```
hostname% pc -o myprog myprog.p
```

If you use this option with the `-c` option, the name you give is used as the name for the object file. The default file name for object files is the source file name with a `.o` extension. You cannot give the object file the same name as the source file.

`–P`

The `–P` option causes the compiler to use partial evaluation semantics on the `boolean` operators, `and` and `or`. Left-to-right evaluation is guaranteed, and the second operand is evaluated only if necessary to determine the result.

`–p` *and* `–pg`

The `–p` and `–pg` options instruct the compiler to produce code that counts the number of times each routine is called. The profiling is based on a periodic sample taken by the system, rather than by line counters.

Using the `-p` *Option*

To generate an execution profile using the `–p` option:

1. **Compile with the** `–p` **option.**
2. **Run** `a.out`**, which produces a** `mon.out` **executable file.**
3. **Type** `prof a.out`**. The program prints a profile.**

Using the `-pg` *Option*

To generate an execution profile using the `–pg` option:

1. **Compile with the** `–pg` **option.**
2. **Run** `a.out`**, which produces a** `gmon.out` **executable file, a more sophisticated profiling tool than** `mon.out`**.**
3. **Type** `gprof a.out`**. The program prints a profile.**

`–pic`, `-Kpic` *and* `–PIC`, `-KPIC`

The `-pic` and `-PIC` options cause the compiler to generate position-independent code (PIC). One of these options should be used for objects which are then put into shared libraries. With PIC, each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in `pc`-relative addressing mode through a procedure linkage table.

The size of the global offset table is limited to 8Kbytes with `-pic`. The `-PIC` option expands the global offset table to handle 32-bit addresses for those rare cases where there are too many data objects for `-pic`.

For more information on `PIC`, see the section on shared libraries in the Solaris documentation.

–Qoption

The –Qoption passes an option to the program. The option value must be appropriate to that program and can begin with a plus or minus sign. The program value can be either cpp(1), cppas, iropt, ld(1), ild(1), pc0, or pc3. For example, the following command passes the option -R to cpp and allows recursive macros:

```
hostname% pc -Qoption cpp -R myprog.p
```

–Qpath *pathname*

The –Qpath option inserts a path name into the compilation search path, hence providing an alternate path to search for each compiler component. You can select this option, for instance, to use a different linker or code generator. In the following command, pc searches /home/pascal/sparc for the compiler components and uses them if it finds them; if pc does not find the specified components, it uses the default components:

```
hostname% pc -Qpath /home/pascal/sparc testp.p
```

–Qproduce

The –Qproduce option instructs pc to produce source code of the type *sourcetype*, which can be one of the following:

.o	Object file from as(1)
.pi	Preprocessed Pascal source from cpp(1)
.s	Assembler source. This option is the same as the -S option.

For example, the following command produces the file, hello.s:

```
hostname% pc -Qproduce .s hello.p
```

-qp

The -qp option is the same as -p option.

-R *path*[:*dir*]

The -R*path*[:*dir*] option passes a colon-separated list of directories that specify the library search path used by the runtime linker. If present and not null, it is recorded in the output object file and passed to the runtime linker.

If both LD_RUN_PATH and the -R option are specified, the -R option takes precedence.

-Rw

The -Rw option checks and issues warnings on record fields which are used, but not set.

By default, the Pascal compiler generates warnings of this kind for whole variables, but not for fields.

This option works only for local record variables that are defined in procedures or functions, not for global variables, that is, variables that are in the main program or in a separately compiled module. This is because global variables may appear to be initialized *not* in the main program itself, but in some procedure or function that is compiled separately, which is subsequently linked to the executable program.

This option is suppressed when the -Z option is on. See "–Z" on page 70. In this case, all local variables and their components are initialized by zero values.

When this option is on, the compiler performs a full analysis (as far as possible at compile time) of how record fields are assigned and used. Warnings contain full access constructs for fields which are used, but not set, for example, V.F1.F2^.F3.

The compiler issues warnings at the end of the procedure where the record variables are defined, that is, when some of the fields are definitely *not* set. However, no warnings are issued if fields are used in the source before they are initialized, as the control flow may be different.

In some cases, it is not possible to determine at compile time whether the fields have actually been initialized. For example:

- For the array variable V, whose elements are records, if any assignment of the kind `V[i]:= X or V[i].F:= Y` occurs, the compiler considers the corresponding fields of `V[i]` for all values of `i` to be initialized. If such a field is used, but not set, it is denoted as `V[...].F` in the warning message.
- All formal parameters are assumed to be initialized. Consequently, the compiler does not perform any checking for these component fields.

With the `-Rw` option, the compiler takes into account built-in procedures which initialize their argument variables, for example, `reset(f)` for the file buffer variable `f^` and its components. `rewrite(f)` does not initialize `f^`. The compiler also examines field handling inside `WITH` statements.

Use the `-Rw` option to check the use of "unsafe" variant records, such as the assignment of a variant to a field, or the use of another field from a "parallel" variant. These practices may result in runtime errors which are hard to find.

Note – The `-Rw` option requires extra compile-time, and is, therefore, recommended for use in debugging only.

Examples:

The Pascal main program, r.p (record and array of records)

```
program p;
procedure qq;
type compl = record re, im: integer end;
    arc = array[1..2] of compl;
var z: compl;
    a: arc;
begin
writeln(z.im);
writeln(a[1].re);
end;
begin
end.
```

The commands to compile r.p and the -Rw warnings that are issued

```
hostname% pc -Rw r.p
Fri Jan 27 17:35:50 1995  r.p:
In procedure qq:
w 18280 field z.im is used but never set
w 18280 field a[...].re is used but never set
```

The Pascal main program, rr.p (two records)

```
program p;
type r = record a,b: integer end;
procedure qq;
var r1, r2: r;
var i: integer;
begin
 i:=r1.a;
 i:=r2.a;
 i:=r1.b;
 i:=r2.b;
end;
begin
 qq;
end.
```

The commands to compile `rr.p` and the `-Rw` warnings that are issued

```
hostname% pc -Rw rr.p
Mon Feb 20 14:59:04 1995  pas/rr.p:
In procedure qq:
w 18280 field r1.b is used but never set
w 18280 field r1.a is used but never set
w 18280 field r2.b is used but never set
w 18280 field r2.a is used but never set
```

The Pascal main program, `recvar.p` (variant record)

```
program p;
procedure qq;
type r = record
      x,y: integer;
             case integer of
             0:(a: integer);
             1: (b: char);
    end;
var v: r;
begin
 v.x:= 1;
 writeln(v.y);
end;
begin
qq;
end.
```

The commands to compile `recvar.p`

```
hostname% pc -Rw recvar.p
Mon Feb 20 15:55:18 1995  recvar.p:
In procedure qq:
w 18260 field v.a is neither used nor set
w 18260 field v.b is neither used nor set
w 18280 field v.y is used but never set
hostname% a.out
          0
```

The Pascal main program, with.p (with statement)

```
program p;
type C = record re, im: integer end;
     AC = array[1..2] of C;
     RC = record C1, C2: C end;
     PRC = ^RC;
procedure qq;
var
 c: C;
 ac: AC;
 rc: RC;
 prc: PRC;
begin
 ac[1]:= c;
 with ac[1] do
 begin
  re:= 1;
 writeln(im);
 end;
 with prc^.C1 do
 begin
  writeln(im);
 end;
end;
begin
qq;
end.
```

The commands to compile and execute with.p

```
hostname% pc -Rw with.p
Mon Feb 20 16:28:34 1995 with.p:
In procedure qq:
w 18280 variable c is used but never set
w 18260 variable rc is neither used nor set
w 18280 field prc^.C1.im is used but never set
hostname% a.out
          0

*** a.out terminated by signal 11: segmentation violation
*** Traceback being written to a.out.trace
Abort (core dumped)
```

–S

The –S option compiles the program and outputs the assembly language in the file, *sourcefile*.s. For example, the following command places the assembly language translation of rmc.p in the file rmc.s. No executable file is created.

```
hostname% pc –S rmc.p
```

–s[*level*]

The –s option instructs the compiler to accept standard Pascal only. Pascal has two levels of compliance with standard Pascal: Level 0 and Level 1. The only difference between the two is that Level 1 also allows conformant arrays.

Specify the level of compliance as follows:

- –s0 Accept Level 0 compliance with standard Pascal
- -s or –s1 Accept Level 1 compliance with standard Pascal

This option causes many features of Pascal that are not found in standard Pascal to be diagnosed with warning messages. These features include:

- Nonstandard procedures and functions
- Extensions to the procedure write
- Padding of constant strings with blanks
- Preprocessor directives

In addition, all letters, except character strings and constants, are mapped to lowercase. Thus, the case of keywords and identifiers is ignored.

This option is most useful when a program is to be ported to other machines.

–sb

The –sb option produces a database for source browsing.

–sbfast

The –sbfast option performs the same task as -sb, but does not compile.

-tc

The `-tc` option instructs the compiler to generate `pc3` stab information that allows cross-module type checking.

This option can be used for two purposes:

- To check for any name conflicts that your program may have with the standard libraries with which it is to be linked, such as `libc`. The linker allows name conflicts, which may cause erroneous runtime behavior in your program.

  For example, the following program has a name conflict with `libc`:

```
program p(output);
var time: integer;
begin
  writeln(wallclock);
end.
```

  When the program is compiled with the `-tc` option, `pc3` issues a warning that the name `time` is already defined as a `libc` routine. Running `a.out` causes a core dump. To avoid this problem, change the name of the variable that has the conflict—in this case, `time`.

- To check for possible name conflicts in the various modules of your program. These conflicts arise if you define a routine with the same name in several modules, or refer to an external, but undefined, variable. The linker detects these error situations and does not create the executable file.

–`temp=`*dir*

The `–temp` option instructs `pc` to locate the temporary files that it creates during compilation in the directory named *dir*. For example, the following command puts the temporary files in the current directory.

```
hostname% pc -temp=. hello.p
```

If you do not specify a temporary directory, `pc` places the temporary files in the `/tmp` directory.

–time

The –time option instructs the compiler to report execution performance statistics for the various compilation passes. Here is some sample output; some spaces have been removed so the output would fit on the page.

```
hostname% pc -time hello.p
cpp:time U:0.0s+S:0.1s=0.2s REAL:1.6s 11%. core T:0k D:0k. io IN:4b OUT:3b. pf IN:25p OUt:184p.
pc0:time U:0.0s+S:0.3s=0.4s REAL:3.2s 13%. core T:0k D:4k. io IN:4b OUT:4b. pf IN:70pOUT:131p.
cg: time U:0.0s+S:0.1s=0.2s REAL:2.0s 12%. core T:0k D:1k. io IN:2b OUT:1b. pf IN:39p OUT:163p.
as: time U:0.0s+S:0.2s=0.3s REAL:1.5s 19%. core T:0k D:1k. io IN:3b OUT:10b.pf IN:33pOUT:117p.
pc3:time U:0.1s+S:0.1s=0.3s REAL:0.9s 31%. core T:0k D:1k. io IN:7b OUT:0b. pf IN:20pOUT:109p.
ld:time U:0.8s+S:0.9s=1.8sREAL:10.2s 17%. core T:0k D:21k.io IN:74bOUT:29b.pf IN:89pOUT:184p.
```

Each line begins with the name of the compiler pass. The rest of the line is divided into four parts: time, core, io, and pf.

- time gives the time used by that pass of the compiler, in this order:
  a. User time
  b. System time
  c. Total CPU time, which is the sum of user and system time
  d. Real (clock) time
  e. Percent of real time used by CPU time
- core gives memory usage statistics for the pass, in this order:
  a. The first item is always 0, and currently has no meaning.
  b. The second item is the integral resident set size.
- The io section gives the volume of input and output operations, expressed in blocks.
- The pf section gives the amount of page faults, expressed in pages, in this order:
  a. Page faults not requiring physical I/O
  b. Page faults requiring physical I/O

–U *name*

The –U option removes any initial definition of the cpp(1) symbol *name*. See cpp(1) for more information. You cannot use this option with the -xl option.

–V

The –V option prints the version number of each compilation pass.

–V0 *and* –V1

The –V0 and –V1 options turn on sets of options that insert checks into the object file, as follows:

–V0	Equivalent to -C, -H, -L, and -s0
–V1	Equivalent to –C, –H, –L, and –s1

–v

The –v (verbose) option prints the command line used to call each compilation pass.

–w

By default, the compiler prints warnings about inconsistencies it finds in the input program. The –w option turns off the warnings.

To turn off warnings in a program comment, use this command:

```
hostname% {$w-}
```

-xa

Same as -a.

-xarch=*a*

The -xarch=*a* option limits the set of instructions the compiler may use.

a must be one of: `generic`, `v7`, `v8`, `v8a`, `v8plus`, `v8plusa`.

Although this option can be used alone, it is part of the expansion of the `xtarget` option; its *primary use* is to override a value supplied by the `xtarget` option.

This option limits the instructions generated to those of the specified architecture, and *allows* the specified set of instructions. It does not guarantee an instruction is used; however, under optimization, it is usually used.

If this option is used with optimization, the appropriate choice can provide good performance of the executable on the specified architecture. An inappropriate choice can result in serious degradation of performance.

`v7, v8,` and `v8a` are all binary compatible. `v8plus` and `v8plusa` are binary compatible with each other and forward, but not backward. For any particular choice, the generated executable can run much more slowly on earlier architectures (to the left in the above list).

Table 3-3 The `-xarch` Values

Value	**Meaning**
`generic`	Get good performance on most SPARCs, and major degradation on none. This is the default. This option uses the best instruction set for good performance on most SPARC processors without major performance degradation on any of them. With each new release, this best instruction set will be adjusted, if appropriate.
`v7`	Limit the instruction set to V7 architecture. This option uses the best instruction set for good performance on the V7 architecture, but without the quad-precision floating-point instructions. This is equivalent to using the best instruction set for good performance on the V8 architecture, but *without* the following instructions: The quad-precision floating-point instructions The integer `mul` and `div` instructions The `fsmuld` instruction Examples: SPARCstation 1, SPARCstation 2
`v8a`	Limit the instruction set to the V8a version of the V8 architecture. This option uses the best instruction set for good performance on the V8 architecture, but without: The quad-precision floating-point instructions The `fsmuld` instruction Example: Any machine based on MicroSPARC I chip architecture

Table 3-3 The `-xarch` Values *(Continued)*

Value	Meaning
`v8`	Limit the instruction set to V8 architecture. This option uses the best instruction set for good performance on the V8 architecture, but without quad-precision floating-point instructions. Example: SPARCstation 10
`v8plus`	Limit the instruction set to the V8plus version of the V9 architecture. By definition, V8plus, or V8+, means the V9 architecture, except: Without the quad-precision floating point instructions Limited to the 32-bit subset defined by the V8+ specification Without the VIS instructions This option uses the best instruction set for good performance on the V9 architecture. In V8+, a system with the 64-bit registers of V9 runs in 32-bit addressing mode, but the upper 32 bits of the i and l registers must not affect program results. Example: Any machine based on UltraSPARC chip architecture. Use of this option also causes the `.o` file to be marked as a Sun-specific V8+ binary; such files will not run on a `v7` or `v8` machine.
`v8plusa`	Limit the instruction set to the V8plusa version of the V9 architecture. By definition, V8plusa means the V8plus architecture, plus: The UltraSPARC-specific instructions The VIS instructions This option uses the best instruction set for good performance on the UltraSPARC™ architecture but limited to the 32-bit subset defined by the V8+ specification. Example: Any machine based on UltraSPARC chip architecture. Use of this option also causes the `.o` file to be marked as a Sun-specific V8+ binary; such files will not run on a `v7` or `v8` machine.

`-xcache=`*c*

The `-xcache=`*c* option defines the cache properties for use by the optimizer.

c must be one of the following:

- `generic`
- *s1*/*l1*/*a1*
- *s1*/*l1*/*a1*:*s2*/*l2*/*a2*
- *s1*/*l1*/*a1*:*s2*/*l2*/*a2*:*s3*/*l3*/*a3*

The *si*/*li*/*ai* are defined as follows:

si	The size of the data cache at level *i*, in kilobytes
li	The line size of the data cache at level *i*, in bytes
ai	The associativity of the data cache at level *i*

Although this option can be used alone, it is part of the expansion of the `-target` option. Its *primary use* is to override a value supplied by the `-target` option.

This option specifies the cache properties that the optimizer can use. It does not guarantee that any particular cache property is used.

Table 3-4 The `-xcache` Values

Value	**Meaning**
`generic`	Define the cache properties for good performance on most SPARCs. This is the default value which directs the compiler to use cache properties for good performance on most SPARC processors, without major performance degradation on any of them. With each new release, these best timing properties will be adjusted, if appropriate.
s1/*l1*/*a1*	Define level 1 cache properties.
s1/*l1*/*a1*:*s2*/*l2*/*a2*	Define levels 1 and 2 cache properties.
s1/*l1*/*a1*:*s2*/*l2*/*a2*:*s3*/*l3*/*a3*	Define levels 1, 2, and 3 cache properties

Example: `-xcache=16/32/4:1024/32/1` specifies the following:

Level 1 cache has:	Level 2 cache has:
16K bytes	1024K bytes
32 bytes line size	32 bytes line size
4-way associativity	Direct mapping associativity

`-xchip=`*c*

The `-xchip=`*c* option specifies the target processor for use by the optimizer.

c must be one of: `generic`, `old`, `super`, `super2`, `micro`, `micro2`, `hyper`, `hyper2`, `powerup`, `ultra`

Although this option can be used alone, it is part of the expansion of the `-target` option; its *primary* use is to provide a value supplied by the `-target` option.

This option specifies timing properties by specifying the target processor.

Some effects are:

- The ordering of instructions, that is, scheduling
- The way the compiler uses branches
- The instructions to use in cases where semantically equivalent alternatives are available

Table 3-5 The `-xchip` Values

Value	**Meaning**
`generic`	Use timing properties for good performance on most SPARCs. This is the default value that directs the compiler to use the best timing properties for good performance on most SPARC processors, without major performance degradation on any of them.
`old`	Use timing properties of pre-SuperSPARC™ processors.
`super`	Use timing properties of the SuperSPARC chip.
`super2`	Use timing properties of the SuperSPARC II chip.
`micro`	Use timing properties of the MicroSPARC™ chip.

Table 3-5 The `-xchip` Values *(Continued)*

Value	**Meaning**
`micro2`	Use timing properties of the MicroSPARC II chip.
`hyper`	Use timing properties of the HyperSPARC™ chip.
`hyper2`	Use timing properties of the HyperSPARC II chip.
`powerup`	Use timing properties of the Weitek® PowerUp™ chip.
`ultra`	Use timing properties of the UltraSPARC chip.

`-xcg89`

Same as `-cg89`.

`-xcg92`

Same as `-cg92`.

`–xF`

The `–xF` option enables performance analysis of the executable file using the SPARCworks Performance Analyzer and Debugger. This option also causes the code generator to generate some debugging information in the object file, necessary for data collection. The compiler generates code that can be reordered at the function level. It takes each function in the file and places it into a separate section. For example, functions `fcn1()` and `fcn2()` are placed in the sections `.text%fcn1` and `.text%fcn2`. You can control the order of functions in the final executable by using the `–xF` and the loader `–Mmapfile` options.

In the map file, if you include the flag `O` in the string of segment flags, then the static linker `ld` attempts to place sections in the order they appear in the map file. See the Solaris documentation for details about this option, the segment flags, and the map file.

-xildoff

Turns off the incremental linker and forces the use of `ld`. This option is the default if you do *not* use the `-g` option, or you do not use the `-G` option, or any source files are present on the command line. Override this default by using the `-xildon` option.

-xildon

Turns on the incremental linker and forces the use of `ild` in incremental mode. This option is the default if you use the `-g` option, and you do not use the `-G` option, and there are no source files present on the command line. Override this default by using the `-xildoff` option.

–xl

The `–xl` option implements a set of features that provide broad compatibility with Apollo Pascal. We recommend using `–xl` only when porting Pascal systems from Apollo platforms to SPARC system platforms. See the *Pascal 4.2 Reference Manual* for details of the features controlled by `-xl`.

When you use `-xl`, the compiler invokes the `cppas` preprocessor in place of `cpp`(1). See Appendix A, “Pascal Preprocessor,” for information on `cppas`.

Modules compiled with `–xl` are *not* compatible with modules compiled without `–xl`. You should not link these two types of modules together.

-xlibmieee

Same as `-libmieee`.

-xlibmil

Same as `-libmil`.

-xlibmopt

Uses a math routine library optimized for performance. The results may be slightly different than those produced by the normal math library. This option is implied by the `-fast` option.

–xlicinfo

The `-xlicinfo` option returns information about the licensing system. In particular, it returns the name of the license server and the IDs of users who have licenses checked out. When you give this option, the compiler is not invoked and a license is not checked out.

–xMerge

The `–xMerge` option instructs `pc` to merge the data segment of the resulting program with the text segment.

-xnolib

Same as `-nolib`.

-xnolibmopt

Resets `-fast`, and does *not* use the math routine library.

Use this option *after* the `-fast` option on the command-line, as in:
`pc -fast -xnolibmopt ...`

-xO5

Optimizes the object code. See also `-O[`*level*`]`.

(Solaris 2.x) This option can be combined with `–g`, but not with `–xa`.

When `-O` is used with the `-g` option, a limited amount of debugging is available.

Generates the highest level of optimization. Uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time. Optimization at this level is more likely to improve performance if it is done with profile feedback. See `-xprofile=p`.

If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization and resumes subsequent procedures at the original level specified in the command-line option.

`-xpg`

Same as `-p` and `-pg`

`-xprofile=`*p*

The `-xprofile=`*p* option collects data for a profile or use a profile to optimize.

p must be `collect`, `use[`*`:name`*`]`, or `tcov`.

This option causes execution frequency data to be collected and saved during execution, then the data can be used in subsequent runs to improve performance.

Table 3-6 The `-xprofile` Values

Value	**Meaning**
`collect`	Collect and save execution frequency for later use by the optimizer. The compiler inserts code to measure the execution frequency at a low level. During execution, the measured frequency data is written into `.prof` files that correspond to each of the source files. If you run the program several times, the execution frequency data accumulates in the `.prof` files; that is, output from prior runs is not lost.
`use`	Use execution frequency data saved by the compiler. Optimize by using the execution frequency data previously generated and saved in the `.prof` files by the compiler. The source files and the compiler options (excepting only this option), must be exactly the same as for the compilation used to create the compiled program that was executed to create the `.prof` files.
`tcov`	Correctly collects data for programs that have source code in header files or make use of C++ templates. See `-a` for information on the old style of profiling, the `tcov`(1) man page, and the *Profiling Tools* manual for more details. Code instrumentation is performed similarly to that of `-a`, but `.d` files are no longer generated. Instead, a single file is generated, whose name is based off of the final executable. For example, if the program is run out of `/foo/bar/myprog`, then the data file is stored in `/foo/bar/myprog.profile/myprog.tcovd`. When running `tcov`, you must pass it the `-x` option to make it use the new style of data. If not, `tcov` uses the old `.d` files, if any, by default for data, and produces unexpected output. Unlike `-a`, the `TCOVDIR` environment variable has no effect at compile-time. However, its value is used at program runtime.

-xregs=*r*

The `-xregs=`*r* option specifies the usage of registers for the generated code.

r is a comma-separated list that consists of one or more of the following: `[no%]appl`, `[no%]float`.

Example: `-xregs=appl,no%float`

Table 3-7 The `-xregs` Values

Value	Meaning
`appl`	Allow using the registers `g2`, `g3`, and `g4`. In the SPARC ABI, these registers are described as *application* registers. Using these registers can increase performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.
`no%appl`	Do not use the `appl` registers.
`float`	Allow using the floating-point registers as specified in the SPARC ABI. You can use these registers even if the program contains no floating-point code.
`no%float`	Do not use the floating-point registers. With this option, a source program cannot contain any floating-point code.

The default is `-xregs=appl,float`.

–xs

The `-xs` option disables Auto-Read for `dbx` in case you cannot keep the `.o` files around. This option passes the `-s` option to the code generator and linker.

- **No Auto-Read**—This is the older way of loading symbol tables.
  - The compiler instructs the linker to place all symbol tables for `dbx` in the executable file.
  - The linker links more slowly and `dbx` initializes more slowly.
  - If you move the executables to another directory, then to use `dbx` you must move the source files, but you need not move the object (`.o`) files.

- **Auto-Read**—This is the newer (and default) way of loading symbol tables.
  - The compiler distributes this information in the `.o` files so that `dbx` loads the symbol table information only if and when it is needed.
  - The linker links faster and `dbx` initializes faster.
  - If you move the executables to another directory, then to use `dbx`, you must move both the source files and the object (`.o`) files.

`-xsafe=mem`

The `-xsafe=mem` option allows the compiler to assume no memory-based traps occur.

This option grants permission to use the speculative load instruction on V9 machines.

`-xsb`

Same as `-sb`.

`-xsbfast`

Same as `-sbfast`.

`-xspace`

The `-xspace` option does no optimizations that increase the code size.

Example: Do not unroll loops.

`-xtarget=`*t*

The `-xtarget=`*t* option specifies the target system for the instruction set and optimization.

t must be one of: `native`, `generic`, *system-name*.

The `-xtarget` option permits a quick and easy specification of the `-xarch`, `-xchip`, and `-xcache` combinations that occur on real systems. The only meaning of `-xtarget` is in its expansion.

Table 3-8 The `-xtarget` Values

Value	**Meaning**
`native`	Get the best performance on the host system. The compiler generates code for the best performance on the host system. It determines the available architecture, chip, and cache properties of the machine on which the compiler is running.
`generic`	Get the best performance for generic architecture, chip, and cache. The compiler expands `-xtarget=generic` to: `-xarch=generic -xchip=generic -xcache=generic` This is the default value.
system-name	Get the best performance for the specified system. You select a system name from Table 3-9 that lists the mnemonic encodings of the actual system names and numbers.

The performance of some programs may benefit by providing the compiler with an accurate description of the target computer hardware. When program performance is critical, the proper specification of the target hardware could be very important. This is especially true when running on the newer SPARC processors. However, for most programs and older SPARC processors, the performance gain is negligible and a generic specification is sufficient.

Each specific value for `-xtarget` expands into a specific set of values for the `-xarch`, `-xchip`, and `-xcache` options. See Table 3-9 for the values. For example:

`-xtarget=sun4/15` is equivalent to:
`-xarch=v8a -xchip=micro -xcache=2/16/1`

Table 3-9 `-xtarget` Expansions

-xtarget	-xarch	-xchip	-xcache
sun4/15	v8a	micro	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1
sun4/30	v8a	micro	2/16/1
sun4/40	v7	old	64/16/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/65	v7	old	64/16/1
sun4/75	v7	old	64/32/1
sun4/110	v7	old	2/16/1
sun4/150	v7	old	2/16/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/630	v7	old	64/32/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1
sslx	v8a	micro	2/16/1

-xtarget	-xarch	-xchip	-xcache
sslx2	v8a	micro2	8/64/1
ssslc	v7	old	64/16/1
ss1	v7	old	64/16/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1
ss2p	v7	powerup	64/31/1
ss4	v8a	micro2	8/64/1
ss5	v8a	micro2	8/64/1
ssvyger	v8a	micro2	8/64/1
ss10	v8	super	16/32/4
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/20	v8	super	16/32/4
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/402	v8	super	16/32/4
ss10/412	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1
ss10/612	v8	super	16/32/4:1024/32/1

-xtarget	-xarch	-xchip	-xcache
ss10/712	v8	super2	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss20/51	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1
ss20/91	v8	super2	16/32/4:1024/32/1
ss20/502	v8	super	16/32/4
ss10/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/712	v8	super	16/32/4:1024/32/1
ss20/912	v8	super	16/32/4:1024/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/	v8	super	16/32/4:1024/32/1
ss600/	v8	super	16/32/4:1024/32/1
ss600/	v8	super	16/32/4:1024/32/1
ss1000	v8	super	16/32/4:1024/32/1
sc2000	v8	super	16/32/4:1024/64/1
cs6400	v8	super	16/32/4:2048/64/1

-xtarget	-xarch	-xchip	-xcache
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ultra	v8	ultra	16/32/1:512/64/1
ultra1/140	v8	ultra	16/32/1:512/64/1
ultra1/170	v8	ultra	16/32/1:512/64/1
ultra2/1170	v8	ultra	16/32/1:512/64/1
ultra2/1170	v8	ultra	16/32/1:512/64/1
ultra2/1170	v8	ultra	16/32/1:1024/64/1

–Z

The `–Z` option instructs `pc` to insert code that initializes all local variables to zero. Standard Pascal does not allow initialization of variables.

-ztext

The `-ztext` option forces a fatal error if relocations remain against non-writable, allocatable sections.

Program Construction and Management 4

This chapter is an introduction to the methods generally used to construct and manage programs using Pascal. It describes units and libraries in two separate sections:

Units

For many reasons, it is often inconvenient to store a program in a single file, as in the case of a very large program.

You can break up a program in several ways. Perhaps the simplest way is to use an `include` file. An `include` file is a separate file that is copied in by the compiler when it encounters an `include` compiler directive. For example, in the following program:

```
program include (output);
#include "includefile"
```

the line `#include "includefile"` is a compiler directive to `cpp`(1), the Pascal compiler's preprocessor. The directive instructs `cpp`(1) to find the file `includefile` and copy it into the stream before continuing.

The actual `includefile` looks like this:

```
begin
   writeln ('Hello, world.')
end.
```

In this example, the `include` file contains the entire program. In reality, an `include` file probably contains a set of variable or procedure declarations. `include` files are often used when a set of declarations needs to be shared among a number of programs.

However, suppose your program is very large and takes a long time to compile. Using `include` files may make editing more convenient, but when you make a change in one part of your program, you still must recompile the entire program. As another example, suppose you want to be able to share compiled code with other people, but for reasons of security or convenience, do not want to share the source code.

Both of these problems are solved by separately compiled units, generally called units. A unit is a part of a program stored in its own file and linked with the rest of the program after compilation.

Using Program Units and Module Units

There are two kinds of units:

- **Program unit**—This unit looks like any program. It begins with a program header and contains the main program.

  Here is an example:

```
program program_unit (output);
procedure say_hello; extern;
begin
   say_hello
end.
```

The body of the procedure say_hello is not defined in this program unit, but the program unit does contain a declaration of the interface to the procedure. The keyword extern declares that say_hello is declared in a module unit.[1]

- **Module unit**—This unit can begin with an optional module header, followed by a set of compilable declarations and definitions. Modules that call externally defined routines must have declarations for those routines.

  Here is an example:

```
module module_unit;
procedure say_hello;
begin
   writeln ('Hello, world.')
end;
```

Every program must have one and only one program unit; a program can have any number of module units. Any unit can call procedures declared in any other unit; each unit must have external declarations for every procedure it uses that is not defined in that unit.

A module unit can also be used as a library, that is, as a collection of useful routines that is shared among a number of programs.

Compiling with Units

Consider the units given in the previous section, “Using Program Units and Module Units.” You can compile and link these units on a single line by executing the following command, which then produces the executable, a.out.

```
hostname% pc program_unit.p module_unit.p
```

1. A statement that shows the interface of a routine is called a declaration, because it declares the name and parameters of the routine. The set of statements that shows the entire routine, including the body, is called the definition of the routine. There can be only one definition for a given routine, but every routine must be declared in every module or program unit that uses it.

You can also separate the compilation and linking or loading steps, as follows:

```
hostname% pc program_unit.p -c
hostname% pc module_unit.p -c
hostname% pc program_unit.o module_unit.o
```

In this case, you call `pc` on each unit with the "compile only" option (`-c`), which produces an object file with the extension `.o`. When you use this option, the compiler driver does not call the linker, `ld`. You then call `pc` a second time, giving the names of the object files, and `pc` calls `pc3` to check for name and type conflicts before calling the linker.

Calling the linker or loader `ld`(1) directly does not have the same effect as calling `pc`; when you call `ld`(1) directly, the files are linked and loaded, but they are not checked for conflicts.

Using Units and Header Files

A complex program may have many routines defined in modules. Each routine must have a declaration (for example, `procedure proc; extern;`) in each file that calls the routine. The easiest way to be sure that you have a correct and consistent set of declarations is to create a header file.

A header file is a file that contains a set of declarations, nothing else. You use a header file by using an `include` directive to include the header file in the compilation.

For example, here is a modified version of the program, `program_unit2`, that uses a header file:

```
program program_unit2 (output);
#include "header.h"
begin
    say_hello
end.
```

In this case, the content of `header.h` is very simple:

```
procedure say_hello; extern;
```

In a real program, `header.h` would probably contain many declarations and would be included in several modules. Aside from routine declarations, header files often contain constant, type, and variable declarations.

Sharing Variables Between Units

Variables that are global across a unit (that is, not declared locally in a routine) can be `public` or `private`. A `public` variable can be shared by any unit that is linked to the unit that declares the variable. A `private` variable cannot be shared.

You can use the `public` and `private` reserved words to declare that a `var` section declares `public` or `private` variables. For example:

```
program program_unit3 (output);
public var
    x : integer;
private var
    y : integer;
```

When you do not use `public` or `private`, variables are `public` by default. However, when you compile with the `-xl` option, variables are `private` by default.

To share a `public` variable, simply declare it in each unit where you want to share it. As long as the variable is `public`, each reference to that variable accesses the same data.

Here is a program unit that declares a variable:

```
program program_unit3 (output);
var
    x : integer;

procedure say_hello; external;

begin
    for x := 1 to 5 do say_hello
end.
```

Here is a module unit that declares a variable with the same name:

```
module module_unit3;
var
    x : integer;

procedure say_hello;

begin
    writeln ('Hello, world for the', x, ' time.')
end;
```

By default, both definitions of variable `x` are `public`. Thus, when you compile and link the program and module units, references to `x` refer to the same variable, as follows:

```
hostname% pc program_unit3.p module_unit3.p
program_unit.p:
module_unit.p:
Linking:
hostname% a.out
Hello, world for the 1 time.
Hello, world for the 2 time.
Hello, world for the 3 time.
Hello, world for the 4 time.
Hello, world for the 5 time.
```

If you compile the program giving the `-xl` option, the variables are `private` by default, as follows:

```
hostname% pc -xl program_unit.p module_unit.p
program_unit.p:
module_unit.p:
Linking:
hostname% a.out
Hello, world for the 0 time.
Hello, world for the 0 time.
Hello, world for the 0 time.
Hello, world for the 0 time.
Hello, world for the 0 time.
```

You can get the same effect by explicitly declaring the variable in a `private var` section. Similarly, when you use `-xl`, you can create `public` variables by declaring them in a `public var` section.

As with routine declarations, it is often a good idea to declare `public` variables in an `include` file. Doing so makes it easier to keep your declarations consistent.

There are other methods for making variables visible to different units. See Chapter 5, “Separate Compilation,” for more information.

Libraries

You can use a module unit as a library of useful functions. The simplest way to do so is to create a source file containing the definitions of your library routines and then compile it using the `-c` option. You can then link the resulting `.o` file to any number of files. For convenience, you probably should create a header file containing the routine declarations for the library.

A simple library as described above has two problems:

- When a library grows in size, it may become inconvenient to store its source in a single file, both for ease of editing and so you can avoid recompiling a large file when you change only part of it.

  On the other hand, it would be inconvenient to have to name many library modules on the command-line when you link your program. Thus, it would be helpful to be able to combine a number of library modules.

- Several programs that you run at the same time may share the same library. Under the scheme described above, each program has its own copy of the library. It saves space and even I/O time if several programs share library code.

Both problems have solutions. First, you can combine or archive modules together. Secondly, you can create a shared library.

See the Solaris documentation on the linker and libraries for information on creating archived and shared libraries.

Separate Compilation 5

This chapter describes how to compile Pascal programs in separate units. Chapter 4, “Program Construction and Management,” gives an introduction to the concepts in this chapter. Following are the sections:

In separate compilation, a program is divided into several units that can be separately compiled into object (`.o`) files. The object files are then linked using `pc`, which invokes `pc3` to check for the consistent use of global names and declarations across the different units, and then invokes `ld`(1) to link and load the units. You can also give `pc` the names of all the units at once, in which case `pc` compiles all the units, checks for consistency, and links the units in one step.

Separate compilation is different from independent compilation. In independent compilation, you invoke `ld` directly, so there is no consistency checking. Independent compilation is not addressed in this guide.

Working with Units

Pascal provides two types of source files or units: the program unit and the module unit.

Using Program Units

The program unit is the source program with the program header. It has the following syntax:

<program unit> `::=` *<program heading>* *<declaration list>* *<program body>*

Each program you write can have only one program unit. The program body is the first code that Pascal executes.

Using Module Units

A module unit is a source program that does not have a program header. It has the following syntax:

<module unit> `::= [` *<module heading>* `]` *<declaration list>*

The module heading contains the reserved word `module` followed by an identifier:

<module heading> `::= [ 'module'` *<identifier>* `';' ]`

For example:

```
module sum;
```

This is a legal module heading. The module heading is optional.

Sharing Variables and Routines Across Multiple Units

Pascal supports three methods of sharing variables and routines between units:

- `include` files
- Multiple variable declarations
- `extern/define` variable declarations

These methods are not mutually exclusive; for example, you can declare a variable as either `extern` or `define` in an `include` file.

The following sections describe these methods.

Compiling without the `-xl` *Option*

There are three ways of sharing variables and routines across units when you compile your program without the `–xl` option.

Sharing Public Variables

If you declare a variable in two or more separate units and the variable is `public` in both places, that variable is shared between units. Variables are `public` by default, unless you compile with the `-xl` option, in which case variables are `private` by default. In this example, the variable `global` is `public` by default, and thus shared between the program and the module.

The program unit, `shrvar_prog.p`

```
program shrvar_prog;

var
    global: integer;

procedure proc; external;

begin { program body }
    global := 1;
    writeln('From MAIN, before PROC: ', global);
    proc;
    writeln('From MAIN,  after PROC: ', global)
end. { shrvar_prog }
```

The module unit, `shrvar_mod.p`. The assignment of a new value to `global` and `max_array` in the procedure `proc` in `shrvar_prog.p` is repeated in `shrvar_mod.p`.

```
module shrvar_mod;

var
   global: integer;

procedure proc;

begin
   writeln('From PROC: ',global);
   global := global + 1
end; { proc }
```

The commands to compile and execute `shrvar_prog.p` and `shrvar_mod.p`

```
hostname% pc shrvar_prog.p shrvar_mod.p
shrvar_prog.p:
shrvar_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC: 1
From PROC             : 1
From MAIN, after PROC : 2
```

Using `extern` *Option to Share Routines*

If a program or module calls a procedure not defined in that unit, you must declare it with either the `extern` or `external` routine option. For instance, in the previous example, the procedure `proc` is defined in `shrvar_mod.p`, but used in `shrvar_prog.p`. Thus, it is declared as `external` in `shrvar_prog.p`. Also, `proc` must also be defined as public in `shrvar_mod.p`, which is the default.

Using `include` *Files to Share Variables and Routines*

The `include` file contains the declarations for the program. Placing all program declarations in a single file makes your program more consistent and easier to maintain.

To use this feature, place the number sign character (#) in the first position of a line immediately followed by the word `include`, and then a file name enclosed in angle brackets (< and >) or double quotation marks ("). The different enclosures (<> and `""`) affect the search order for files. The syntax for the `#include` directive is determined by `cpp`(1).

When the compiler encounters the `#include` in the input, it inserts the lines from the included file into the input stream.

The program unit, inc_prog.p, which includes the file include.h

```
program inc_prog;

#include "include.h"

begin { program body}
    global := 1;
    writeln('From MAIN, before PROC: ', global);
    proc;
    writeln('From MAIN,  after PROC: ', global)
end. { inc_prog }
```

The module unit, inc_mod.p, which also includes the file include.h

```
module inc_mod;

#include "include.h"

procedure proc;

begin
    writeln('From PROC             : ', global);
    global := global + 1
end; { proc }
```

The include file, include.h

```
var
    global : integer;

procedure proc; extern;
```

The commands to compile and execute inc_prog.p and inc_mod.p

```
hostname% pc inc_prog.p inc_mod.p
inc_prog.p:
inc_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC:     1
From PROC :                 1
From MAIN, after PROC:      2
```

Using the `-xl` Option

When you use the `–xl` option, variables and top-level procedures and functions declared in the program unit default to `private`. Look at the difference when you compile and execute `shrvar_prog.p` and `shrvar_mod.p` with `–xl`. See the source code in "Sharing Public Variables" on page 81.

The commands to compile and execute `shrvar_prog.p` and `shrvar_mod.p` with the `–xl` option

```
hostname% pc -xl shrvar_prog.p shrvar_mod.p
shrvar_prog.p:
shrvar_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC: 1
From PROC : 0
From MAIN, after PROC: 1
```

Without `–xl`, the variable `global` in `shrvar_mod.p` is treated as `public`; here, `global` is treated as `private`. Thus, the assignment:

```
global := global + 1;
```

is not reflected in `shrvar_prog.p`; instead, each file uses its own private copy of `global`.

The following sections describe five ways of sharing variables and routines across units when you compile your program with `–xl`.

Using `public var` Declarations

The following examples uses the `public` attribute in the `var` declaration to make `global public` when you compile your program with `–xl`.

The program unit, pubvar_prog.p, which declares global as public

```
program pubvar_prog;

public var
    global: integer;

procedure proc;
    external;

begin
    global := 1;
    writeln('From MAIN, before PROC: ', global);
    proc;
    writeln('From MAIN,  after PROC: ', global)
end. { pubvar_prog }
```

The module unit, pubvar_mod.p, which also declares global as public

```
module pubvar_mod;

public var
    global : integer;

procedure proc;

begin
    writeln('From PROC             :',global);
    global := global + 1;
end; { proc }
```

The commands to compile and execute pubvar_prog.p and pubvar_mod.p

```
hostname% pc -xl pubvar_prog.p pubvar_mod.p
pubvar_prog.p:
pubvar_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC: 1
From PROC : 1
From MAIN, after PROC: 2
```

Using the `define` *Variable Attribute*

This example makes `global public` using the define attribute of the variable declaration.

The program unit, `defvar_prog.p`

```
program defvar_prog;

var
    global: extern integer;

procedure proc;
    external;

begin
    global := 1;
    writeln('From MAIN, before PROC: ', global);
    proc;
    writeln('From MAIN,  after PROC: ', global);
end. { defvar_prog }
```

The module unit, `defvar_mod.p`, which makes `global` public using the `define` attribute

```
module defvar_mod;

var
    global : define integer;

procedure proc;

begin
    writeln('From PROC             : ',global);
    global := global + 1;
end; { proc }
```

The commands to compile and execute `defvar_prog.p` and `defvar_mod.p`

```
hostname% pc -xl defvar_prog.p defvar_mod.p
defvar_prog.p:
defvar_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC:     1
From PROC             :     1
From MAIN, after PROC :     2
```

Using the `define` *Declaration*

This example defines `global` in the module `defvar_mod2` using the `define` declaration. The advantage of using the `define` declaration over the `define` variable attribute is that the `define` declaration can be easily converted to use `include` files.

The program unit, `defvar_prog.p`

```
program defvar_prog;

var
    global: extern integer;

procedure proc;
    external;

begin
    global := 1;
    writeln('From MAIN, before PROC: ', global);
    proc;
    writeln('From MAIN,  after PROC: ', global)
end. { defvar_prog }
```

The module unit, `defvar_mod2.p`, which defines `global` in a `define` declaration

```
module defvar_mod2;

var
    global : extern integer;

define
    global;

procedure proc;

begin
    writeln('From PROC             : ',global);
    global := global + 1;
end; { proc }
```

The commands to compile and execute `defvar_prog.p` and `defvar_mod2.p`

```
hostname% pc -xl defvar_prog.p defvar_mod2.p
defvar_prog.p:
defvar_mod2.p:
Linking:
hostname% a.out
From MAIN, before PROC:     1
From PROC             :     1
From MAIN, after PROC :     2
```

Using `include` *Files*

In the following example, the `extern` declaration for the variable `global` is in the `include` file, `inc_prog2.p`, and is therefore included in both files. The `define` declaration in file `inc_mod2.p` cancels the `extern` definition.

The program unit, inc_prog2.p

```
program inc_prog2;

%include "include2.h";

procedure proc; extern;

begin
    global := 1;
    writeln('From MAIN, before PROC: ',global);
    proc;
    writeln('From MAIN,  after PROC: ',global);
end. { proc }
```

The module unit, inc_mod2.p

```
module inc_mod2;

define
    global;

%include "include2.h";

procedure proc;

begin
    writeln('From PROC             : ',global);
    global := global + 1;
end; { proc }
```

The include file, include2.h

```
var
    global : extern integer;
```

The commands to compile and execute `inc_prog2.p` and `inc_mod2.p`

```
hostname% pc -xl inc_prog2.p inc_mod2.p
inc_prog2.p:
inc_mod2.p:
Linking:
hostname% a.out
From MAIN, before PROC:  1
From PROC             :  1
From MAIN, after PROC :  2
```

Using `extern`

In the previous example, the `extern` definition for variables is put into an `include` file and then shared. You can do the same for the `extern` procedure definition. In doing so, you must also declare the variable with a `define` declaration in the module that defines the procedure. This declaration nullifies the effect of the `extern` declaration.

The program unit, `ext_prog.p`

```
program ext_prog;

%include "extern.h";

begin
    global := 1;
    writeln('From MAIN, before PROC: ',global);
    proc;
    writeln('From MAIN,  after PROC: ',global);
end. { ext_prog }
```

The module unit, ext_mod.p

```
module ext_mod;

define
    global, proc;

%include "extern.h";

procedure proc;

begin
    writeln('From PROC : ',global);
    global := global + 1;
end; { proc }
```

The include file, extern.h

```
var
    global : extern integer;

procedure proc; extern;
```

The commands to compile and execute ext_prog.p and ext_mod.p

```
hostname% pc -xl ext_prog.p ext_mod.p
ext_prog.p:
ext_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC: 1
From PROC : 1
From MAIN, after PROC: 2
```

Sharing Declarations in Multiple Units

Using `extern` and `external` directives for procedure and function declarations, you can optionally specify the source language of a separately compiled procedure or function. For example, `extern fortran` directs the compiler to generate calling sequences compatible with the FORTRAN compiler from SunSoft. Then, `external c` directs the compiler to generate calling sequences compatible with SunSoft C compiler.

For routines declared `extern fortran` or `external fortran`, the changes in the calling sequence are as follows:

- For value parameters, the compiler creates a copy of the actual argument value in the caller's environment and passes a pointer to the temporary variable on the stack. Thus, you need not create (otherwise useless) temporary variables.
- The compiler appends an underscore to the name of the external procedure to conform to a naming convention of the `f77`(1) compiler. Pascal procedure names called from FORTRAN must supply their own trailing underscore (_).
- Multidimensional Pascal arrays are not compatible with FORTRAN arrays. Because FORTRAN uses column-major ordering, a multidimensional Pascal array passed to FORTRAN appears transposed.

For routines declared `extern c` or `external c`, a warning is generated if you attempt to pass a nested function.

When you compile your program with the `-xl` option, you can also use `nonpascal` to declare non-Pascal routines when porting DOMAIN programs written in DOMAIN Pascal, FORTRAN, and C.

The C–Pascal Interface 6

This chapter describes how to mix C and Pascal modules in the same program. It contains the following sections:

The examples in this chapter assume that you are using the ANSI C compiler. To invoke ANSI C:

- On the Solaris 1.x environment, use the `acc` command
- On the Solaris 2.x environment, use the `cc` command

Compilation of Mixed-Language Programs

You must use the compiler option `–lpc` when you compile a C main routine that calls Pascal. `–lpc` includes the Pascal object library `libpc`. For example:

```
hostname% pc -c -calign my_pascal.p
hostname% cc my_pascal.o my_c.c -lpc
```

The `-c` option produces an unlinked object file. The `-calign` option causes `pc` to use C-like data formats for aggregate objects.

When you compile a Pascal main routine that calls C, you don't have to use any special options, but the `-calign` option is again useful. The C object library, `libc`, is automatically brought in for every Pascal compilation.

For example:

```
hostname% cc -c my_c.c
hostname% pc -calign my_c.o my_pascal.p
```

Compatibility of Types for C and Pascal

Table 6-1 and Table 6-2 list the default sizes and alignments of compatible types for C and Pascal.

Table 6-1 C and Pascal Size and Alignment of Compatible Types

Pascal Type	C Type	Size (bytes)	Alignment (bytes)
`double`	`double`	8	8
`longreal`	`double`	8	8
`real`	`double`	8	8
`single`	`float`	4	4
`shortreal`	`float`	4	4
`integer16`	`short int`	2	2
`integer32`	`int`	4	4
`integer`	`int`	4	4
`-128..127`	`char`	1	1
`boolean`	`char`	1	1
`alfa`	`char a[10]`	10	1
`char`	`char`	1	1
`string`	`char a[80]`	80	1
varying[*n*] of `char`	`struct{int, char[`*n*`]}`	-	4
`record`	`struct/union`	-	Same as element type

Table 6-1 C and Pascal Size and Alignment of Compatible Types *(Continued)*

Pascal Type	C Type	Size (bytes)	Alignment (bytes)
`array`	`array`	-	Same as element type
variant record	`struct/union`	-	-
fields in packed record	bit field	-	-

Table 6-2 C and Pascal Size and Alignment of Compatible Types with `-xl`

Pascal Type	C Type	Size (bytes)	Alignment (bytes)
`real`	`float`	4	4
`integer`	`short int`	2	2

Precautions with Compatible Types

This section describes the precautions you should take when working with compatible types.

The `shortreal` *Type*

The Pascal `shortreal` and C `float` compatibility works if you pass by reference. See “Value Parameters” on page 116 for examples that show you how to pass by value.

Character Strings

C has several assumptions about strings. All C strings are:

- Passed by reference since C strings are arrays
- Terminated by a null byte
- Located in static variable storage

You can satisfy these assumptions as follows:

- Pass by reference by making the strings `var`, `in`, `out`, or `in out` parameters.

- Provide the null byte explicitly before passing a string to C. Pascal guarantees the null byte only if the string is a constant. The null byte is not required by the ISO Pascal Standard.

Array Indexes

Pascal array indexes can start at any integer; C array indexes always start at zero.

Aggregate Types

Aggregate types include arrays, varying arrays, sets, strings, alphas, records, and variant records.

Pascal aggregate types may require alignment and layout adjustment to match C unions, structures, and arrays. Pascal aggregate types can be any of the following: arrays, varying arrays, sets, strings, alphas, records, or variant records.

However, you can use the `-calign` option to eliminate some of these differences. With `-calign`, the following assumptions are made:

- Pascal records have the same data layout as C structures.
- Arrays have the same data layout in both languages. However, if you use the `-xl` option in addition to `-calign`, `boolean` arrays with an odd number of elements are different.
- Pascal variants are the same as C unions.

Incompatibilities

This section describes the incompatibilities between C and Pascal types.

Enumerated Types

C enumerated types are incompatible with Pascal enumerated types. Pascal enumerated types are represented internally by sequences of integral values starting with 0. Storage is allocated for a variable of an enumerated type as if the type was a subrange of integer. For example, an enumerated type of fewer than 128 elements is treated as 0..127, which, according to the rules above, is equivalent to a `char` in C.

C enumerated types are allocated a full word and can take on arbitrary integer values.

Pascal Set Types

In Pascal, a set type is implemented as a bit vector, which is similar to a C short-word array, where each short-word contains two bytes. Thus, sets are bit-vectors, and they are allocated in multiples of 16 bits. To find out the size of a set, enter the following code:

```
ceiling( ord( highest_element ) / 16 )
```

Direct access to individual elements of a set is highly machine-dependent and should be avoided.

General Parameter Passing in C and Pascal

A few general rules apply to parameter passing:

- C passes all arrays by reference since C strings are arrays.
- C passes all structures by value.
- In C, if you want to pass anything else by reference, then you must explicitly prepend the reference ampersand (&), or pass an explicit pointer.
- Pascal passes all parameters by value unless you explicitly state that they are `var`, `in out`, or `out` parameters, in which case they are passed by reference.

Procedure Calls: C–Pascal

Here are examples of how a C main program calls a Pascal procedure:

The Pascal procedure, Samp, in the file Samp.p. Note the procedure definition.

```
procedure Samp(var i: integer; var r: real);
begin
    i := 9;
    r := 9.9
end; { Samp }
```

The C main program, SampMain.c. Note the procedure definition and call.

```
#include <stdio.h>

extern void Samp(int *, double *);

int main (void)
{
   int     i ;
   double  d ;

   Samp(&i, &d) ;
   printf ("%d  %3.1f \n", i, d) ;
}
```

The commands to compile and execute Samp.p and SampMain.c

```
hostname% pc -c Samp.p
hostname% cc Samp.o SampMain.c
hostname% a.out
 9 9.9
```

Variable Parameters

Pascal passes all variable parameters by reference, which C can do, too.

Simple Types without `-xl`

Without `-xl`, simple types match, as in the following example:

The Pascal procedure, `SimVar.p`

```
procedure SimVar(
    var t, f: boolean;
    var c: char;
    var si: integer16;
    var i: integer;
    var sr: shortreal;
    var r: real);
begin
    t := true;
    f := false;
    c := 'z';
    si := 9;
    i := 9;
    sr := 9.9;
    r := 9.9;
end; { SimVar }
```

The C main program, `SimVarMain.c`

```
#include <stdio.h>

extern void SimVar(char *, char *, char *, short *,
                   int *, float *, double *);

int main(void)
{
    char       t, f, c;
    short      si;
    int        i;
    float      sr;
    double     r;

    SimVar(&t, &f, &c, &si, &i, &sr, &r);
    printf(" %08o %08o %c %d %d %3.1f %3.1f \n",
        t, f, c, si, i, sr, r);
}
```

The commands to compile and execute `SimVar.p` and `SimVarMain.c`

```
hostname% pc -c SimVar.p
hostname% cc SimVar.o SimVarMain.c
hostname% a.out
 00000001 00000000 z 9 9 9.9 9.9
```

Simple Types with `-xl`

With the `-xl` option, the Pascal `real` must be paired with a C `float`, and the Pascal `integer` must be paired with a C `short int`.

Strings of Characters

The C counterpart to the Pascal `alfa` and `string` types are arrays; C passes all arrays by reference. The C counterpart to the Pascal `varying` is a structure; C passes structures by value.

Before you call Pascal with a null varying string, set the byte count to zero because that is what Pascal assumes about such strings.

C can pass a structure consisting of a four-byte integer and an array of characters to a Pascal procedure, expecting a `var` parameter that is a variable-length string.

See the following example:

The Pascal procedure, `StrVar.p`

```
type
    TVarStr = varying [25] of char;

procedure StrVar(
    var a: alfa;
    var s: string;
    var v: TVarStr);
begin
    a := 'abcdefghi' + chr(0);
    s := 'abcdefghijklmnopqrstuvwxyz' + chr(0);
    v := 'varstr' + chr(0);
end;  { StrVar }
```

The C main program, `StrVarMain.c`

```
#include <stdio.h>
#include <string.h>

struct TVarLenStr {
    int nbytes;
    char a[25];
};

extern void StrVar(char *, char *, struct TVarLenStr *);

int main(void)
{
    struct TVarLenStr vls;
    char s10[10], s80[80], s25[25];

    vls.nbytes = 0;
    StrVar(s10, s80, &vls);
    strncpy(s25, vls.a, vls.nbytes);
    printf(" s10 = '%s' \n s80 = '%s' \n s25 = '%s' \n",
        s10, s80, s25);
    printf(" strlen(s25) = %d \n", strlen(s25));
}
```

The commands to compile and execute `StrVar.p` and `StrVarMain.c`

```
hostname% pc -c StrVar.p
hostname% cc StrVar.o StrVarMain.c -lpc
hostname% a.out
 s10='abcdefghi'
 s80='abcdefghijklmnopqrtstuvwxyz'
 s25='varstr'
 strlen(s25)=6
```

Fixed Arrays

For a fixed array parameter, pass the same type and size by reference, as shown in the following example:

The Pascal procedure, `FixVec.p`

```
type
    VecTyp = array [0..8] of integer;

procedure FixVec(var V: TVec; var Sum: integer);
var
    i: integer;
begin
    Sum := 0;
    for i := 0 to 8 do
        Sum := Sum + V[i]
end; { FixVec }
```

The C main program, `FixVecMain.c`

```
#include <stdio.h>

extern void FixVec(int [], int *);

int main(void)
{
    int Sum;
    static int a[] = {0,1,2,3,4,5,6,7,8};

    FixVec(a, &Sum);
    printf(" %d \n", Sum);
}
```

The commands to compile and execute `FixVec.p` and `FixVecMain.c`

```
hostname% pc -c -calign FixVec.p
hostname% cc FixVec.o FixVecMain.c -lpc
hostname% a.out
 36
```

Although it does not apply in this example, arrays of aggregates in Pascal have, by default, a size that is always a multiple of four bytes. When you use the `-calign` option to compile the Pascal code, that difference with C is eliminated.

The following example illustrates this point. The string `'Sunday'` only gets through to the C main program when you compile the Pascal routine using `-calign`.

The Pascal procedure, `DaysOfWeek.p`

```
type
    TDay= array [0..8] of char;
    TWeek = array [0..6] of day;
    TYear = array [0..51] of week;

procedure DaysOfWeek(var Y: TYear);
begin
    v[1][1] := 'Sunday';
end;
```

The C main program, `DaysOfWeekMain.c`

```
#include <stdio.h>

extern void DaysOfWeek(char [][7][9]);

int main(void)
{
    char Year[52][7][9];

    DaysOfWeek(Year);
    printf(" Day = '%s' \n", Year[1][1]);
}
```

The commands to compile and execute `DaysOfWeek.p` and `DaysOfWeekMain.c` ***without*** `-calign`

```
hostname% pc -c DaysOfWeek.p
hostname% cc DaysOfWeek.o DaysOfWeekMain.c -lpc
hostname% a.out
 Day = ''
```

The commands to compile and execute `DaysOfWeek.p` and `DaysOfWeekMain.c` ***with*** `-calign`

```
hostname% pc -c -calign DaysOfWeek.p
hostname% cc DaysOfWeek.o DaysOfWeekMain.c -lpc
hostname% a.out
day = 'Sunday '
```

The `univ` *Arrays*

You can pass any size array to a Pascal procedure expecting a `univ` array, although there is no special gain in doing so, because there is no type or size checking for separate compilations. However, if you want to use an existing Pascal procedure that has a `univ` array, you can do so. All `univ` arrays that are `in`, `out`, `in out`, or `var` parameters pass by reference.

The Pascal procedure, `UniVec.p`, which defines a 10-element array

```
type
    TVec = array [0..9] of integer;

procedure UniVec(
    var V: univ TVec;
    in Last: integer;
    var Sum: integer);
var
    i: integer;
begin
    Sum := 0;
    for i := 0 to Last do
        Sum := Sum + V[i];
end; { UniVec }
```

The C main program, `UniVecMain.c`, which passes a 3-element array to the Pascal procedure written to do a 10-element array

```
#include <stdio.h>

extern void UniVec(int *, int, int *);

int main(void)
{
    int Sum;
    static int a[] = {7, 8, 9};

    UniVec(a, 2, &Sum);
    printf(" %d \n", Sum);
}
```

The commands to compile and execute `UniVec.p` and `UniVecMain.c` with `-calign`

```
hostname% pc -c -calign UniVec.p
hostname% cc UniVec.o UniVecMain.c -lpc
hostname% a.out
 24
```

Conformant Arrays

For single-dimension conformant arrays, pass upper and lower bounds, placed after the declared parameter list. If the array is multidimensional, pass element widths as well, one element width for each dimension, except the last one.

See this example:

```
function ip(var x: array [lb..ub: integer] of real): real;

extern double ip(double [], int, int);
        ...
    double v1[10];
    double z;
    z = ip(v1, 0, 9);
        ...
```

One bounds pair may apply to several arrays if they are declared in the same parameter group:

```
function ip(var x,y:array[lb..ub:integer] of real):real;
     ...
     double v1[10], v2[10] ;
     extern double ip() ;
     double z ;
     z = ip ( v1, v2, 0, 9 ) ;
     ...
```

With multidimensional arrays, for all dimensions but the last one, pass the low bound, high bound, and element width.

Examples of single-dimension, multidimension, and array-of-character conformant arrays follow. Conformant arrays are included here only because they are a relatively standard feature; there are usually more efficient and simpler ways to do the same thing.

Example 1: Single-Dimension Array

The Pascal procedure, `IntCA.p`. Pascal passes the bounds pair.

```
procedure IntCA(var a: array [lb..ub: integer] of integer);
begin
    a[1] := 1;
    a[2] := 2
end; { IntCA }
```

The C main program, `IntCAMain.c`

```
#include <stdio.h>

extern void IntCA(int [], int, int);

int main(void)
{
    int k ;
    static int s[] = { 0, 0, 0 };

    IntCA (s, 0, sizeof(s)-1);
    for (k=0 ; k < 3 ; k++)
        printf(" %d \n", s[k]);
}
```

The commands to compile and execute `IntCA.p` and `IntCAMain.c` with `-calign`

```
hostname% pc -c -calign IntCA.p
hostname% cc IntCA.o IntCAMain.c -lpc
hostname% a.out
     0
     1
     2
```

Example 2: Multi-Dimension Array

The Pascal procedure, RealCA.p. Pascal passes low bound, high bound, and element width.

```
procedure RealCA(var A: array [r1..r2: integer] of
                        array [c1..c2: integer] of real);
var
    col, row: integer;
begin
    for row := r1 to r2 do
        for col := c1 to c2 do
            if row = col then
                A[row, col] := 1.0
            else
                A[row, col] := 0.0
end; { RealCA }
```

The C main program, RealCAMain.c. Array M has 2 rows of 3 columns each. c1 and c2 are the first and last columns. r1 and r2 are the first and last rows. wc is the width of a column element (smallest) and is equal to sizeof(M[0][0]). wr is the width of a row element (next largest) and is equal to (c2-c1+1) * wc.

```
#include <stdio.h>
#define NC 3
#define NR 2
extern void RealCA(double [][NC], int, int, int, int, int);
int main(void)
{
    double        M[NR][NC];
    int           col, c1, c2, row, r1, r2, wc, wr;

    c1 = 0;
    r1 = 0;
    c2 = NC - 1;
    r2 = NR - 1;
    wc = sizeof(M[0][0]);
    wr = (c2 - c1 + 1) * wc;
    RealCA(M, r1, r2, wr, c1, c2);
    for (row = r1; row <= r2; row++) {
        printf("\n");
          for (col = c1; col <= c2; col++)
            printf("%4.1f", M[row][col]);
    };
    printf("\n");
}
```

The commands to compile and execute `RealCA.p` and `RealCAMain.c` with `-calign`

```
hostname% pc -c -calign RealCA.p
hostname% cc RealCA.o RealCAMain.c -lpc
hostname% a.out

    1.0    0.0    0.0
    0.0    1.0    0.0
```

If `wc` is the width of the smallest element, as determined by `sizeof()`, then the width of the next largest element is the number of those smaller elements in the next larger element multiplied by `wc`.

width of next largest element `= (ub - lb + 1) * wc`

In general, (`lb`, `ub`, `wc`) are the bounds and element width of the next lower dimension of the array. This definition is recursive.

Example 3: Array of Characters

The Pascal procedure, `ChrCAVar.p`

```
procedure ChrCAVar(var a: array [lb..ub: integer] of char);

begin
    a[0]  := 'T';
    a[13] := 'o';
end; { ChrCAVar }
```

The C main program, `ChrCAVarMain.c`. For C, the lower bound is always 0.

```
#include <stdio.h>

extern void ChrCAVar(char [], int, int);

int main(void)
{
    static char s[] = "this is a string" ;

    ChrCAVar( s, 0, sizeof(s)-1) ; /*(s, lower, upper)*/
    printf("%11s \n", s) ;
}
```

The commands to compile and execute `ChrCAVar.p` and `ChrCAVarMain.c`

```
hostname% pc -c -calign ChrCAVar.p
hostname% cc ChrCAVar.o ChrCAVarMain.c -lpc
hostname% a.out
This is a string
```

Records and Structures

In most cases, a Pascal record describes the same objects as its C structure equivalent, provided that the components have compatible types and are declared in the same order. The compatibility of the types depends mostly on size and alignment. For more information on size and alignments of simple components, see "Compatibility of Types for C and Pascal" on page 94.

By default, the alignment of a record is always four bytes and the size of a record is always a multiple of four bytes. However, when you use `-calign` in compiling the Pascal code, the size and alignment of the Pascal record matches the size and alignment of the equivalent C structure.

A Pascal record of an integer and a character string matches a C structure of the same constructs, as follows:

The Pascal procedure, `StruChr.p`. It is safer for the Pascal procedure to explicitly provide the null byte and include it in the count before the string is passed to C.

```
type
    TLenStr = record
          nbytes: integer;
          chrstr: array [0..24] of char
      end;

procedure StruChr(var v: TLenStr);
begin
    v.NBytes := 14;
    v.ChrStr := 'St. Petersburg' + chr(0);
end; { StruChr }
```

The C main program, `StruChrMain.c`

```
#include <stdio.h>
#include <string.h>

struct TVarLenStr {
    int NBytes;
    char a[25];
};

extern void StruChr(struct TVarLenStr *);

int main(void)
{
    struct TVarLenStr vls;
    char   s25[25];

    vls.NBytes = 0;
    StruChr(&vls);
    strncpy(s25, vls.a, vls.NBytes);
    printf(" s25 = '%s' \n", s25);
    printf(" strlen(s25) = %d \n", strlen(s25));
}
```

The commands to compile and execute `StruChr.p` and `StruChrMain.c`

```
hostname% pc -c StruChr.p
hostname% cc StruChr.o StruChrMain.c -lpc
hostname% a.out
 s25='St. Petersburg'
 strlen(s25) = 13
```

The record in the example above has, by default, the same size and alignment as the equivalent C record. Some records, though, are laid out differently unless you use the `-calign` option.

Consider this example:

The Pascal routine, `DayWeather.p`

```
type
    TDayWeather = record
        TDay: array [0..8] of char;
        TWeather: array [0..20] of char;
    end;

    TDayWeatherArray = array [0..1] of TDayWeather;

procedure DayWeather(var W: TDayWeatherArray;
                    var WeatherSize: integer);
begin
    W[1].TDay := 'Sunday' + chr(0);
    W[1].TWeather := 'Sunny' + chr(0);
    WeatherSize := 5;
end;  { StruChr }
```

The C main program, `DayWeatherMain.c`

```
#include <stdio.h>
#include <string.h>

struct TDayRec {
    char TDay[9];
    char TWeather[21];
};

extern void DayWeather(struct TDayRec [], int *);

int main(void)
{
    char s25[25];
    char t25[25];
    struct TDayRec dr[2];
    int nbytes = 0;

    DayWeather(dr, &nbytes);
    strncpy(s25, dr[1].TDay, 6);
    printf(" day = '%s' \n", s25);
    strncpy(t25, dr[1].TWeather, nbytes);
    printf(" weather = '%s' \n", t25);
}
```

When you compile the Pascal routine without using the `-calign` option, the program does not work correctly.

The commands to compile and execute `DayWeather.p` and `DayWeatherMain.c` ***without*** `-calign`

```
hostname% pc -c DayWeather.p
hostname% cc DayWeather.o DayWeatherMain.c -lpc
hostname% a.out
 day = ''
 weather = ' sun'
```

The commands to compile and execute `DayWeather.p` and `DayWeatherMain.c` ***with*** `-calign`

```
hostname% pc -calign -c DayWeather.p
hostname% cc DayWeather.o DayWeatherMain.c -lpc
hostname% a.out
 day = 'Sunday'
 weather = 'sunny'
```

Variant Records

C equivalents of variant records can sometimes be constructed, although there is some variation with architecture and sometimes a need to adjust alignment. You can avoid the need to adjust alignment by using the `-calign` option.

The Pascal procedure, `VarRec.p`

```
type
    vr = record
          case tag: char of
              'a': (ch1, ch2: char);
              'b': (flag: boolean);
              'K': (ALIGN: integer);
      end;

procedure VarRec(var x: vr);
begin
    if x.ch1 = 'a' then
        x.ch2 := 'Z'
end; { VarRec }
```

The C main program, `VarRecMain.c`

```
#include <stdio.h>

    struct vlr {
        char tag;
        union {
            struct {
                char ch1, ch2;
            }a_var;
            struct {
                char flag;
            }b_var;
            struct {
                int ALIGN;
            }c_var;
        }var_part;
};

extern void VarRec(struct vlr *);

int main(void)
{
    struct vlr *x;

    x = (struct vlr *)malloc(sizeof(struct vlr));
    x->tag = 'a';
    x->var_part.a_var.ch1 = 'a';
    x->var_part.a_var.ch2 = 'b';
    VarRec(x);
    printf(" %c \n", x->var_part.a_var.ch2);
}
```

The commands to compile and execute `VarRec.p` and `VarRecMain.c`

```
hostname% pc -c -calign VarRec.p
hostname% cc VarRec.o VarRecMain.c -lpc
hostname% a.out
 Z
```

Pascal Set Type

In Pascal, a set type is implemented as a bit vector, which is similar to a C short-word array. Direct access to individual elements of a set is highly machine-dependent and should be avoided.

In Pascal, bits are numbered within a byte from the most significant to least, as shown in Table 6-3.

Table 6-3 Set Implementation

Set	Bit Numbering
set+3:	31, 30, 29, 28, 27, 26, 25, 24
set+2:	23, 22, 21, 20, 19, 18, 17, 16
set+1:	15, 14, 13, 12, 11, 10, 9, 8
set+0:	7, 6, 5, 4, 3, 2, 1, 0

In C, a set could be described as a short-word array beginning at an even address. With the current set representation, it does not matter what the lower-bound value is.

The *n*th element in a set `[lower...upper]` can be tested as follows:

```
#define LG2BITSLONG 5 /* log2( bits in long word) */
#define LG2BITSWORD 4 /* log2( bits in short word) */
#define MSKBITSLONG 0x1f
#define MSKBITSHORT 0x0

    short *setptr; /* set as array of shorts */
    int upper;     /* upper bound of the set */
    int elem;      /* ordinal value of set element */
    int i;

    if  ( setptr[elem >> LG2BITSWORD]  &
    (1 <<  (elem & MSKBITSWORD))      )  {
              /* elem is in set */
    }
```

Pascal `intset` *Type*

The Pascal `intset` type is predefined as `set of [0..127]`. A variable of this type takes 16 bytes of storage.

The Pascal procedure, `IntSetVar.p`, which has an `intset` of the elements `[1, 3, 7, 8]`

```
procedure IntSetVar(var s: intset);
begin
    s := [1, 3, 7, 8]
end; { IntSetVar }
```

The C main program, `IntSetVarMain.c`

```
#include <stdio.h>

extern void IntSetVar(unsigned int *);

int main(void)
{
    int  w ;
    unsigned int  *p, *s ;

    s = (unsigned int *) malloc(16);
    IntSetVar(s) ;
    for (w = 0, p = s ; w < 4 ; w++, p++)
        printf("%012o %3d \n", *p, w);
    printf(" 110 001 010 (binary, word 4) \n");
    printf(" 876 543 210 (bits, word 4)" \n");
}
```

The commands to compile and execute `IntSetVar.p` and `IntSetVarMain.c`. The output of this example depends on the architecture of your machine.

```
hostname% pc -c IntSetVar.p
hostname% cc IntSetVar.o IntSetVarMain.c -lpc
hostname% a.out
 000000000000 0
 000000000000 1
 000000000000 2
 000000000612 3
 110 001 010 (binary, word 4)
 876 543 210 (bits, word 4)
```

Value Parameters

There are three types of value parameters in Pascal.

Simple Types without `-xl`

Without `-xl`, simple types match, as in the following example:

The Pascal procedure, `SimVal`. p. `t`, `f`, `c`, `i`, `r`, and `s` are value parameters.

```
procedure SimVal(
    t, f: boolean;
    c:  char;
    si: integer16;
    i:  integer;
    sr: shortreal;
    r:  real;
    var reply: integer);

begin
    Reply := 0;
    if t then
        Reply := Reply + 1;
    if not f then
        Reply := Reply + 8;
    if c='z' then
        Reply := Reply + 64;
    if si=9 then
        Reply := Reply + 512;
    if i=9 then
        Reply := Reply + 4096;
    if sr=shortreal(9.9) then
        Reply := Reply + 32768;
    if r=9.9 then
        Reply := Reply + 262144;
end;  { SimVal }
```

The C main program, SimValMain.c

```
#include <stdio.h>

extern void SimVal(
    char, char, char,
    short,
    int,
    float,
    double,
    int *);

int main(void)
{
    char        t = 1, f = 0;
    char        c = 'z';
    short       si = 9;
    int         i = 9;
    float       sr = 9.9;
    double      r = 9.9;
    int         args;

    SimVal(t, f, c, si, i, sr, r, &args);
    printf(" args = %06o \n", args);
```

The commands to compile and execute SimVal.p and SimValMain.c

```
hostname% pc -c SimVal.p
hostname% cc SimVal.o SimValMain.c -lpc
hostname% a.out
 args=111111
```

If no function prototype is provided for `SimVal` in `SimValMain.c`, then `sr:shortreal` must be changed to `sr:real` in `SimVal.p`. This change is necessary because in C, a `float` is promoted to double in the absence of function prototypes. In `-xl` mode, change `sr:shortreal` to `sr:longreal`.

Simple Types with `–xl`

With `-xl`, the Pascal `real` must be paired with a C `float`, and the Pascal `integer` must be paired with a C `short int`.

Arrays

Since C cannot pass arrays by *value*, it cannot pass strings of characters, fixed arrays, or `univ` arrays by value.

Conformant Arrays

Pascal passes all *value* parameters on the stack or in registers, except for value conformant array parameters, which are handled by creating a copy in the caller environment and passing a pointer to the copy. In addition, the bounds of the array must be passed (see "Conformant Arrays" on page 105).

This example is the same as the single-dimension example in "Conformant Arrays," except that the `var` prefix is deleted.

The Pascal procedure, `ChrCAVal.p`

```
procedure ChrCAVal(a: array [lb..ub: integer] of char);
begin
    a[0] := 'T';
    a[13] := 'o';
end; { ChrCAVal }
```

The C main program, `ChrCAValMain.c`

```
#include <stdio.h>

extern void ChrCAVal(char [], int, int);

int main(void)
{
    static char s[] = "This is a string";

    ChrCAVal(s, 0, sizeof(s) -1);
    printf(" %11s \n", s);
}
```

The commands to compile and execute `ChrCAVal.p` and `ChrCAValMain.c` with `-calign`

```
hostname% pc -c -calign ChrCAVal.p
hostname% cc ChrCAVal.o ChrCAValMain.c -lpc
hostname% a.out
This is a string
```

Function Return Values

Function return values match types the same as with parameters, and they pass in much the same way.

Simple Types

The simple types pass in a straightforward way, as follows:

The Pascal function, `RetReal.p`

```
function RetReal(x: real): real;
begin
    RetReal := x + 1.0
end; { RetReal }
```

The C main program, `RetRealMain.c`

```
#include <stdio.h>

extern double RetReal(double);

int main(void)
{
    double  r, s;

    r = 2.0;
    s = RetReal(r);
    printf(" %f \n", s);
}
```

The commands to compile and execute RetReal.p and RetRealMain.c

```
hostname% pc -c RetReal.p
hostname% cc RetReal.o RetRealMain.c
hostname% a.out
 3.000000
```

Input and Output

If your C main program calls a Pascal procedure that does I/O, then include the following code before you call the Pascal procedure:

```
PASCAL_IO_INIT();
```

Also, in the C main program just before exit, add the following line:

```
PASCAL_IO_DONE();
```

See this example:

The Pascal procedure, pasc_read.p

```
procedure pasc_read;
var
    Tfile : text;
    data : integer;

begin
  writeln ('In Pascal procedure');
  reset(Tfile, 'data.txt');

  while not eof(Tfile)
    do
      begin
        readln (Tfile,data)
      end;

  writeln ('At end of Pascal procedure',data)
end;
```

The C main program, main.c

```
#include <stdio.h>

extern void pasc_read();

int main(void)
{
    FILE *ptr;
    printf ("Calling Pascal routine\n");
    PASCAL_IO_INIT();
    pasc_read();
    PASCAL_IO_DONE();
    printf ("After Pascal routine\n");
}
```

The commands to compile and execute pasc_read.p and main.c

```
hostname% pc pasc_read.p -c
hostname% cc -g main.c pasc_read.o -lpc
hostname% a.out
Calling Pascal routine
In Pascal procedure
At end of Pascal procedure1
After Pascal routine
```

Procedure Calls: Pascal–C

This section parallels the section, "Procedure Calls: C–Pascal" on page 97. Earlier comments and restrictions also apply here.

Variable Parameters

Pascal passes all *variable* parameters by reference, which C can do, too.

Simple Types

Simple types pass in a straightforward manner, as follows:

The C function, SimRef.c

```
void SimRef(
    char      *t,
    char      *f,
    char      *c,
    short     *si,
    int       *i,
    float     *sr,
    double    *r)
{
    *t = 1;
    *f = 0;
    *c = 'z';
    *si = 9;
    *i = 9;
    *sr = 9.9;
    *r = 9.9;
}
```

The Pascal main program, SimRefMain.p

```
program SimRefMain(output);
var
    t, f: boolean;
    c: char;
    si: integer16;
    i: integer;
    sr: shortreal;
    r: real;

procedure SimRef(
    var t, f: boolean;
    var c: char;
    var si: integer16;
    var i: integer;
    var sr: shortreal;
    var r: real);
    external c;
begin
    SimRef(t, f, c, si, i, sr, r);
    writeln(t, f: 6, c: 2, si: 2, i: 2, sr :4:1, r :4:1);
end.  { SimRefMain }
```

The commands to compile and execute `SimRef.c` and `SimRefMain.p`

```
hostname% cc -c SimRef.c
hostname% pc SimRef.o SimRefMain.p
hostname% a.out
true false z 9 9 9.9 9.9
```

Strings of Characters

The `alfa` and `string` types pass simply; varying strings are more complicated. All pass by reference.

The C function, `StrVar.c`

```
#include <string.h>

struct TVarLenStr {
    int nbytes;
    char a[26];
};

void StrVar(char *s10, char *s80, struct TVarLenStr *vls)
{
    static char ax[11] = "abcdefghij";
    static char sx[81] = "abcdefghijklmnopqrstuvwxyz";
    static char vx[6] = "varstr";

    strncpy(s10, ax, 11);
    strncpy(s80, sx, 80);
    strncpy(vls->a, vx, 6);
    vls->nbytes = 6;
}
```

The Pascal main program, StrVarMain.p

```
program StrVarMain(output);
type
    TVarStr = varying[26] of char;

var
    a: alfa;
    s: string;
    v: TVarstr;

procedure StrVar(var a: alfa; var s: string; var v: TVarStr);
       external c;

begin
    StrVar(a, s, v);
    writeln(a);
    writeln(s);
    writeln(v);
    writeln(' length(v) = ', length(v) :2);
end.  { StrVarMain }
```

The commands to compile and execute StrVar.c and StrVarMain.p

```
hostname% cc -c StrVar.c
hostname% pc StrVar.o StrVarMain.p
hostname% a.out
abcdefghij
abcdefghijklmnopqrtstuvwxyz
varstr
 length(v) = 6
```

Avoid constructs that rely on strings being in static variable storage. For example, you could use mktemp(3) in Pascal as follows:

Incorrect use of string in static variable storage

```
tmp := mktemp('/tmp/eph.xxxxxx')
```

This use is incorrect, since mktemp() modifies its argument. Instead, use the C library routine strncpy() (see string(3)) to copy the string constant to a declared char array variable, as in:

Correct solution using the C library routine `strncpy()`

```
program Use_mktemp ;

procedure strncpy( var dest: univ string ;
    var srce: univ string ;
    length: integer) ;  external c ;

procedure mktemp(var dest: univ string); external c;
    ...
var path: string ;
begin
    ...
strncpy( path, '/tmp/eph.xxxxxx', sizeof(path)) ;
mktemp( path ) ;
    ...
end .
```

Fixed Arrays

For a fixed-array parameter, pass the same type and size, as in this example:

The C function, FixVec.c

```
void FixVec(int V[9], int *Sum)
{
    int i;

    *Sum = 0;
    for (i = 0; i <= 8; i++)
        *Sum = *Sum + V[i];
}
```

The Pascal main program, FixVecMain.p

```
program FixVecMain(output);
type
    TVec = array [0..8] of integer;
var
    V: TVec := [0, 1, 2, 3, 4, 5, 6, 7, 8];
    Sum: integer;

procedure FixVec(var XV: TVec; var XSum: integer); external c;

begin
    FixVec(V, Sum);
    writeln(Sum: 3);
end. { FixVecMain }
```

The commands to compile and execute FixVec.c and FixVecMain.p

```
hostname% cc -c FixVec.c
hostname% pc -calign FixVec.o FixVecMain.p
hostname% a.out
 36
```

The `-calign` option is not needed for this example, but may be necessary if the array parameter is an array of aggregates.

The `univ` *Arrays*

The `univ` arrays that are `in`, `out`, `in out`, or `var` parameters pass by reference.

Here is an example:

The C function, `UniVec.c`

```
void UniVec(int V[3], int Last, int *Sum)
{
    int i;

    *Sum = 0;
    for (i = 0; i <= Last; i++)
        *Sum += V[i];
}
```

The Pascal main program, `UniVecMain.p`

```
program UniVecMain(output);
type
    TVec = array [0..9] of integer;
var
    Sum: integer;
    V: array [0..2] of integer;

procedure UniVec(var V: univ TVec; in Last: integer;
                    var Sum: integer);
        external c;

begin
    V[0] := 7;
    V[1] := 8;
    V[2] := 9;
    UniVec(V, 2, Sum);
    writeln(Sum);
end.  { UniVecMain }
```

The commands to compile and execute `UniVec.c` and `UniVecMain.p`

```
hostname% cc -c UniVec.c
hostname% pc -calign UniVec.o UniVecMain.p
hostname% a.out
            24
```

The `-calign` option is not needed for this example, but may be necessary if the array parameter is an array of aggregates.

Conformant Arrays

For single-dimension conformant arrays, pass upper and lower bounds placed after the declared parameter list. If the array is multidimensional, pass element widths as well, one element width for each dimension, except the last one. Chapter 8, "The FORTRAN–Pascal Interface," has an example of multidimensional conformant array passing.

The following example uses a single-dimension array:

The C function, `IntCA.c`

```
void IntCA(int a[], int lb, int ub)
{
    int k;

    for (k=0; k <= ub - lb; k++)
        a[k] = 4;
}
```

The Pascal main program, `IntCAMain.p`. Note that what Pascal passes as `s`, is received in C as `a`, `lb`, `ub`.

```
program IntCAMain(output);

var
    s: array [1..3] of integer;
    i: integer;

procedure IntCA(var a: array [lb..ub: integer] of integer);
           external c;

begin
    IntCA(s);
    for i := 1 to 3 do
      write(s[i]);
    writeln
end. { IntCAMain }
```

The commands to compile and execute `IntCA.c` and `IntCAMain.p`

```
hostname% cc -c IntCA.c
hostname% pc -calign IntCA.o IntCAMain.p
hostname% a.out
           4          4          4
```

The `-calign` option is not needed for this example, but may be necessary if the array parameter is an array of aggregates.

Records and Structures

In most cases, a Pascal record describes the same objects as its C structure equivalent, provided that the components have compatible types and are declared in the same order. For more information, see "Compatibility of Types for C and Pascal" on page 94.

Records that contain aggregates may differ because aggregates in C and Pascal sometimes have different sizes and alignments. If you compile the Pascal code with the `-calign` option, the differences are eliminated.

A Pascal record of an integer and a character string matches a C structure of an integer and an array of `char` values, as follows:

The C function, `StruChr.c`

```
#include <string.h>

struct TVarLenStr {
    int nbytes;
    char a[26];
};

void StruChr(struct TVarLenStr *v)
{
    strncpy(v->a, "strvar", 6);
    v->nbytes = 6;
}
```

The Pascal main program, `StruChrMain.p`

```
program StruChrMain(output);
type
    TVarLenStr = record
          nbytes: integer;
          a: array [0..25] of char
    end;
var
    vls: TVarLenStr;
    i: integer;

procedure StruChr(var vls: TVarLenStr); external c;

begin
    StruChr(vls);
    write(' string=''');
    for i := 0 to vls.nbytes - 1 do
        write(vls.a[i]);
    writeln('''');
    writeln(' length = ', vls.nbytes)
end. { StruChrMain }
```

The commands to compile and execute `StruChr.c` and `StruChrMain.p`

```
hostname% cc -c StruChr.c
hostname% pc -calign StruChr.o StruChrMain.p
hostname% a.out
 string=' strvar'
 length=          6
```

Variant Records

C equivalents of variant records can sometimes be constructed, although there is some variation with the architecture, and sometimes you have to adjust the alignment.

Following are some examples:

The C function, VarRec.c

```
struct vlr {
    char tag;
    union {
        struct {
            char ch1, ch2;
    }   a_var;
        struct {
            char flag;
    }   b_var;
        struct {
             int ALIGN;
    }   c_var;
    } var_part;
};

void VarRec(struct vlr *x)
{
    if (x->var_part.a_var.ch1 == 'a')
        x->var_part.a_var.ch2 = 'Z';
}
```

The Pascal main program, VarRecMain.p

```
program VarRecMain;
type
    vr = record
        case tag: char of
            'a': (ch1, ch2: char);
            'b': (flag: boolean);
            'K': (ALIGN: integer)
      end;
var
    x: vr;

procedure VarRec(var d: vr); external c;

begin
    x.tag := 'a';
    x.ch1 := 'a';
    x.ch2 := 'b';
    VarRec(x);
    writeln(x.ch2)
end. { VarRecMain }
```

The commands to compile and execute VarRec.c and VarRecMain.p

```
hostname% cc -c VarRec.c
hostname% pc -calign VarRec.o VarRecMain.p
hostname% a.out
Z
```

The `-calign` option is not needed in the previous example, but may be necessary if the record contains aggregates.

Non-Pascal Procedures

When you use the `-xl` option in compiling Pascal code, you can use the `nonpascal` keyword to declare that an external procedure is written in another language. This keyword generally causes everything to be passed by reference.

See this example:

The C function, NonPas.c. In the function for_C, s is a pointer (declared var in the procedure declaration), and len is not a pointer (not declared var in the procedure declaration). In the function for_nonpascal, s is still a pointer (though not declared var in the procedure declaration), and len is now a pointer (though not declared var).

```
#include <stdio.h>

void for_C(char *s, int len)
{
    int i;
    for (i = 0; i < len; i++)
        putchar(s[i]);
    putchar('\n');
}

void for_NonPascal(char *s, int *len)
{
    int i;
    for (i = 0; i < *len; i++)
        putchar(s[i]);
    putchar('\n');
}
```

The Pascal main program, NonPasMain.p

```
program NonPasMain;
var
    s: string;

procedure for_C(var s: string; len: integer); external c;
procedure for_NonPascal(var s: string; len: integer); nonpascal;

begin
    s :='Hello from Pascal';
    for_C(s, 18);
    for_NonPascal(s, 18);
end.  { NonPasMain }
```

The commands to compile and execute NonPas.c and NonPasMain.p

```
hostname% cc -c NonPas.c
hostname% pc NonPas.o NonPasMain.p
hostname% a.out
 Hello from Pascal
 Hello from Pascal
```

Value Parameters

In general, Pascal passes value parameters in registers or on the stack, widening to a full word if necessary.

Simple Types

With value parameters, simple types match, as in the following example:

The C function, SimVal.c

```
void SimVal(
    char       t,
    char       f,
    char       c,
    short      si,
    int        i,
    float      sr,
    double     r,
    int        *reply)
{
    *reply = 0;
    if (t)                 *reply +=      01;
    if (!f)                *reply +=     010;
    if (c == 'z')          *reply +=    0100;
    if (si == 9)           *reply +=   01000;
    if (i == 9)            *reply +=  010000;
    if (sr ==(float)9.9)  *reply += 0100000;
    if (r == 9.9)          *reply +=01000000;
}
```

The Pascal main program, `SimValMain.p`

```
program SimVal(output);

var
    t: boolean     := true;
    f: boolean     := false;
    c: char        := 'z';
    si: integer16  := 9;
    i: integer     := 9;
    sr: shortreal  := 9.9;
    r: double      := 9.9;
    args: integer;

procedure SimVal(
    t, f: boolean;
    c: char;
    si: integer16;
    i: integer;
    sr: shortreal;
    r: double;
    var Reply: integer);
    external c;
begin
    SimVal(t, f, c, si, i, sr, r, args);
    writeln(' args = ', args :6 oct);
end.  { SimVal }
```

The commands to compile and execute `SimVal.c` and `SimValMain.p`

```
hostname% cc -c SimVal.c
hostname% pc SimVal.o SimValMain.p
hostname% a.out
 args=111111
```

Function Return Values

Function return values match types in the same manner as with parameters, and they pass in much the same way. See “Variable Parameters” on page 98. The following example shows how to pass simple types.

The C function, `RetReal.c`

```
double RetReal(double *x)
{
    return(*x + 1.0);
}
```

The Pascal main program, `RetRealMain.p`

```
program RetRealMain;
var
    r, s: real;

function RetReal(var x: real): real; external c;

begin
    r := 2.0;
    s := RetReal(r);
    writeln(r: 4: 1, s: 4: 1)
end. { RetRealMain }
```

The commands to compile and execute `RetReal.c` and `RetRealMain.p`

```
hostname% cc -c RetReal.c
hostname% pc RetReal.o RetRealMain.p
hostname% a.out
 2.0 3.0
```

Parameters That Are Pointers to Procedures

Pascal has a special type that is a pointer to a procedure. A variable of this type can be used as a parameter, as follows:

The C function, `ProcPar.c`

```
#include <string.h>

void proc_c (void (*p)())  /* a pointer to procedure argument */
{
    char *s ;
    s = "Called from C";
    (*p)( s, strlen(s));  /* Call the Pascal routine */
}
```

The Pascal main program, ProcParMain.p, which calls the C procedure, proc_c, passing it the address of the Pascal procedure, proc_pas. The C procedure assigns a value to the string s, and calls the procedure whose pointer it just received. Then the Pascal procedure, proc_pas, writes a literal constant and the string it just received.

```
program ProcParMain;
type
    { Declare a procedure pointer type. }
    proc_ptr = ^procedure(var s: string; i: integer);

{Declare an external C procedure which takes a procedure argument.}

procedure proc_c(p: proc_ptr); external c;

procedure proc_pas(var cstr: string; strlen: integer);
var
    i: integer;
begin
    write('Hello from PROC_PASCAL: ');
    for i := 1 to strlen do
        write(cstr[i])
    writeln;
end; { proc_pas }

begin
    { Call the C routine. }
    proc_c(addr(proc_pas))
end. { ProcParMain }
```

The commands to compile and execute ProcPar.c and ProcParMain.p

```
hostname% cc -c ProcPar.c
hostname% pc ProcPar.o ProcParMain.p
hostname% a.out
 Hello from PROC_PASCAL: Called from C
```

Procedures and Functions as Parameters

It is probably clearer to pass a pointer to a procedure than to pass the procedure name itself. See “Procedure Calls: Pascal–C” on page 121.

A procedure or function passed as an argument is associated with a static link to its lexical parent’s activation record. When an outer block procedure or function is passed as an argument, Pascal passes a null pointer in the position normally occupied by the passed routine’s static link. So that procedures and functions can be passed to other languages as arguments, the static links for all procedure or function arguments are placed after the end of the conformant array bounds pairs, if any.

Routines in other languages can be passed to Pascal; a dummy argument must be passed in the position normally occupied by the passed routine's static link. If the passed routine is not a Pascal routine, the argument is used only as a placeholder.

Global Variables in C and Pascal

If the types are compatible, a global variable can be shared between C and Pascal.

An example:

The Pascal procedure, `GloVar.p`

```
var
    Year: integer;

procedure GloVar;
begin
    Year := 2001
end; { GloVar }
```

The C main program, `GloVarMain.c`

```
#include <stdio.h>

extern void GloVar();

int Year;

int main(void)
{
    Year = 2042;
    GloVar();
    printf( " %d \n", Year ) ;
}
```

The commands to compile and execute GloVar.p and GloVarMain.c without -xl. With -xl, the Pascal integer must be paired with a C short int and declared public since the default visibility is private.

```
hostname% pc -c GloVar.p
hostname% cc GloVar.o GloVarMain.c
hostname% a.out
 2001
```

File-Passing Between Pascal and C

You can pass a file pointer from Pascal to C, then have C do the I/O, as in:

The C procedure, UseFilePtr.c

```
#include <stdio.h>

void UseFilePtr (FILE *ptr)
{
    { /* Write to the file: */
    fprintf( ptr, "[1] Passing the file descriptor \n") ;
    fprintf( ptr, "[2] and writing information \n") ;
    fprintf( ptr, "[3] to a file \n") ;
}
```

The Pascal main program, UseFilePtrMain.p

```
program UseFilePtrMain;
var
    f: text;
    cfile: univ_ptr;

procedure UseFilePtr(cf: univ_ptr); external c;

begin
    rewrite(f, 'myfile.data'); { Make the file. }
    cfile := getfile(f);       { Get a file pointer. }
    UseFilePtr(cfile);         { Call the C function. }
end. { UseFilePtrMain }
```

The commands to compile and execute UseFilePtc.c and UseFilePtrMain.p

```
hostname% cc -c UseFilePtr.c
hostname% pc UseFilePtr.o UseFilePtrMain.p
hostname% a.out
hostname% cat myfile.data
[1] Passing the file descriptor
[2] and writing information
[3] to a file
```

The C++–Pascal Interface 7

This chapter describes how to mix C++ and Pascal modules in the same program. It contains the following sections:

Sample Interface

You must use the compiler option `-lpc` when you use `CC` to link a C++ main routine that calls Pascal. `-lpc` denotes linking with the Pascal runtime support library `libpc`. On the Solaris 1.x environment, if you use `pc` to link, you must add the `-lc` option.

The `-calign` option causes `pc` to use data formats for aggregate objects similar to those in C++.

Compatibility of Types for C++ and Pascal

Table 6-1 and Table 6-2 on page 94 list the default sizes and alignments of compatible types for C and Pascal. They apply to C++ as well.

C++ Name Encoding

To implement function overloading and type-safe linkage, the C++ compiler normally appends type information to the function names. To prevent the C++ compiler from doing so, and to allow Pascal to call a C++ function, declare the C++ function with the `extern "C"` language construct. One common way to do this is in the declaration of a function, like this:

```
extern "C" void f (int);
...
void f (int) { /* ...body of f... */ }
```

For brevity, you can also combine `extern  "C"` with the definition of the function, as in:

```
extern "C" void f (int)
{ /* ...body of f... */ }
```

Procedure Calls: C++–Pascal

Following are examples that illustrate how a C++ main program calls a Pascal procedure. Included in each example are the Pascal procedure, the C++ main program, and the commands to compile and execute the final program.

The Pascal procedure, `Samp`, in the file, `Samp.p`

```
procedure Samp (var i: integer; var r: real);

begin
   i := 7;
   r := 3.14;
end
```

The C++ main program, `SampMain.cc`

```
#include <stdio.h>

extern "C" void Samp (int&, double&);
int main(void)
{
   int     i;
   double d;
   Samp (i, d);
   printf ("%d %3.2f \n", i, d);
}
```

The commands to compile and execute `Samp.p` and `SampMain.cc`

```
hostname% pc -c Samp.p
hostname% CC Samp.o SampMain.cc -lpc
hostname% a.out
7 3.14
```

Arguments Passed by Reference

C++ arguments can be passed by reference. This section describes how they work with Pascal.

Simple Types without the `-xl` *Option*

Without the `-xl` option, simple types match, as in the following example:

The Pascal procedure, `SampRef`, in the file, `Samp.p`

```
procedure SamRef (
   var t, f: boolean;
   var c: char;
   var i: integer;
   var s: integer16;
   var r: shortreal;
   var d: real
   );

begin
   t := true;
   f := false;
   c := 'z';
   i := 9;
   s := 9;
   r := 9.9;
   d := 9.9;
end;
```

The C++ main program, `SamRefMain.cc`

```
#include <stdio.h>

extern "C" void SamRef (
   char     &,
   char     &,
   char     &,
   int      &,
   short    &,

   float    &,
   double   &);

int main(void)
{
   char     t, f, c;
   int      i;
   short    s;

   float   r;
   double d;

SamRef (t, f, c, i, s, r, d);
printf ("%08o %08o %c %d %d %3.1f %3.1f \n",
          t,    f,    c, i, s, r,     d);

}
```

The commands to compile and execute `SamRef.p` and `SamRefMain.cc`

```
hostname% pc -c SamRef.p
hostname% CC SimRef.o SamRefMain.cc -lpc
hostname% a.out
00000001 00000000 z 9 9 9.9 9.9
```

Simple Types with the `-xl` *Option*

With the `-xl` option, the Pascal `real` must be paired with a C++ `float`; the Pascal integer must be paired with a C++ `short int`.

Strings of Characters

The C++ counterpart to the Pascal alfa and string types are arrays. The C++ counterpart to the Pascal varying type is a structure.

Here is an example:

The Pascal procedure, `StrRef.p`

```
type
   TVarStr = varying [25] of char;

procedure StrRef (
   var a: alfa;
   var s: string;
   var v: TVarStr
   );

begin
   a := 'abcdefghi' + chr(0);
   s := 'abcdefghijklmnopqrstuvwxyz' + chr(0);
   v := 'varstr' + chr(0);
end;
```

The C++ main program, `StrRefMain.cc`

```
#include <stdio.h>
#include <string.h>

  struct TVarLenStr {
   int NBytes;
   char a[25];
};

extern "C" void StrRef (
  char  *,
  char  *,
  TVarLenStr &);

int main(void)
{
  struct TVarLenStr vls;
  char s10[10],
       s80[80],
       s25[25];

  vls.NBytes = 0;
  StrRef (s10, s80, vls);
  strncpy (s25, vls.a, vls.NBytes);
  printf (" s10 = '%s' \n s80 = '%s' \n s25 = '%s' \n",
              s10,           s80,           s25);
  printf (" strlen (s25) = %d \n", strlen(s25));

}
```

The commands to compile and execute `StrRef.p` and `StrRefMain.cc`

```
hostname% pc -c StrRef.p
hostname% CC StrRef.o StrRefMain.cc -lpc
hostname% a.out
s10 = 'abcdefghi'
s80 = 'abcdefghijklmnopqrstuvwxyz'
s25 = 'varstr'
strlen (s25) = 6
```

Fixed Arrays

The Pascal procedure, `FixVec.p`

```
type
   TVec = array [0..8] of integer;

procedure FixVec (
   var V: TVec;
   var Sum: integer
      );
   var
      i: integer;

begin
   Sum := 0;
   for i := 0 to 8 do
      Sum := Sum + V[i];
end;
```

The C++ main program, `FixVedMain.cc`

```
#include <stdio.h>

extern "C" void FixVec (
   int [],
   int &);

int main(void)
{
   int Sum;
   static int a[] = {1,2,3,4,5,6,7,8,9};

   FixVec (a, Sum);

   printf (" %d \n", Sum);
}
```

The commands to compile and execute `FixVec.p` and `FixVecMain.cc`

```
hostname% pc -c FixVec.p
hostname% CC FixVec.o FixVecMain.cc -lpc
hostname% a.out
45
```

Although it does not apply to this example, arrays of aggregates in Pascal have, by default, a size that is a multiple of four bytes. When you use the `-calign` option to compile Pascal code, that difference from C++ is eliminated.

The following example illustrates this point. The string `'Sunday'` gets through to the C++ main program only when you compile the Pascal routine using the `-calign` option.

The Pascal procedure, `DaysOfWeek.p`

```
type
   TDay  = array [0..8] of char;
   TWeek = array [0..6] of TDay;
   TYear = array [0..51] of TWeek;

procedure DaysOfWeek (
   var Y: TYear
    );

begin
   Y[1][1] := 'Sunday';
end;
```

The C++ main program, `DaysOfWeekMain.cc`

```
#include <stdio.h>

extern "C" void DaysOfWeek (
   char [52][7][9]);

int main(void)
{
   char Year [52][7][9];

   DaysOfWeek (Year);

   printf (" Day = '%s' \n", Year[1][1]);
}
```

The commands to compile and execute `DaysOfWeek.p` and `DaysOfWeekMain.cc` ***without*** the `-calign` option

```
hostname% pc -c DaysOfWeek.p
hostname% CC DaysOfWeek.o DaysOfWeekMain.cc -lpc
hostname% a.out
Day = ''
```

The commands to compile and execute `DaysOfWeek.p` and `DaysOfWeekMain.cc` ***with***the `-calign` option

```
hostname% pc -c -calign DaysOfWeek.p
hostname% CC DaysOfWeek.o DaysOfWeekMain.cc -lpc
hostname% a.out
Day = 'Sunday'
```

Records and Structures

A Pascal record of an integer and a character string matches a C++ structure of the same constructs, as in this example:

The Pascal procedure, `StruChr.p`. It is safer for the Pascal procedure to explicitly provide the null byte and include it in the count before the string is passed to C++.

```
type
   TLenStr = record
      NBytes: integer;
      ChrStr: array [0..24] of char;
   end;

procedure StruChr (
   var v: TLenStr
   );

begin
   v.NBytes := 14;
   v.ChrStr := 'St.Petersburg' + chr(0);
end;
```

The C++ main program, `StruChrMain.cc`

```
#include <stdio.h>
#include <string.h>

  struct TVarLenStr {
    int NBytes;
    char a[25];
  };

extern "C" void StruChr (
  TVarLenStr &);

int main(void)
{
  struct TVarLenStr vls;
  char   s25[25];

  vls.NBytes = 0;
  StruChr (vls);
  strncpy (s25, vls.a, vls.NBytes);
  printf ("s25 = '%s' \n", s25);
  printf ("strlen (s25) = %d \n", strlen(s25));

}
```

The commands to compile and execute `StruChr.p` and `StruChr.cc`

```
hostname% pc -c StruChr.p
hostname% CC StruChr.o StruChrMain.cc -lpc
hostname% a.out
s25 = 'St.Petersburg'
strlen (s25) = 13
```

Consider this example:

The Pascal procedure, `DayWeather.p`

```
type
  TDayWeather = record
    TDay: array [0..8]  of char;
    TWeather:array [0..20] of char;
  end;
  TDayWeatherArray = array [0..1] of TDayWeather;

procedure DayWeather (
  var W: TDayWeatherArray;
  var WeatherSize: integer
  );

begin
  W[1].TDay := 'Sunday';
  W[1].TWeather := 'Sunny';
  WeatherSize := 5;
end;
```

The C++ main program, `DayWeatherMain.cc`

```
#include <stdio.h>
#include <string.h>

   struct TDayRec {
      char TDay[9];
      char TWeather[21];
   };

extern "C" void DayWeather (
   TDayRec [2],
   int &);

int main(void)
 {
   struct TDayRec dr[2];
   int NBytes;
   char    s25[25];
   char    t25[25];
   NBytes = 0;
   DayWeather (dr, NBytes);

   strncpy (s25, dr[1].TDay, 6);
   printf ("  day = '%s' \n", s25);
   strncpy (t25, dr[1].TWeather, NBytes);
   printf ("  weather = '%s' \n", t25);

}
```

When you compile the Pascal routine without the `-calign` option, the program does not work correctly.

```
hostname% pc -c DayWeather.p
hostname% CC DayWeather.o DayWeatherMain.cc -lpc
hostname% a.out
day = ''
weather = '   Sun'
```

Compile with the `-calign` option. The program now works correctly.

```
hostname% pc -calign -c DayWeather.p
hostname% CC DayWeather.o DayWeatherMain.cc -lpc
hostname% a.out
   day = 'Sunday'
   weather = 'Sunny'
```

Arguments Passed by Value

C++ arguments can be passed by value. In this section, we describe how they work with Pascal.

Simple Types without the `-xl` Option

Without the `-xl` option, simple types match, as in the following example:

The Pascal procedure, `SimVal.p`

```
procedure SimVal(
   t, f: boolean;
   c: char;
   si:integer16;
   i: integer;
   sr:shortreal;
   r: real;
   var Reply: integer);

begin
   Reply := 0;
   if t then
      Reply := Reply + 1;
   if not f then
      Reply := Reply + 8
   if c='z' then
      Reply := Reply + 64;
   if si=9 then
      Reply := Reply + 512;
   if i=9 then
      Reply := Reply + 4096;
   if sr=shortreal(9.9) then
      Reply := Reply + 32768;
   if r=9.9 then
      Reply := Reply + 262144;

end;
```

The C++ main program, `SimValMain.cc`

```
#include <stdio.h>

extern "C" void SimVal(
  char,
  char,
  char,
  short,
  int,
  float,
  double,
  int &);

int main(void)
{
   char     t = 1, f = 0, c= 'z';
   short     si = 9;
   int      i=9;
   float    sr = 9.9;
   double   r =9.9;
   int      args;

   SimVal (t, f, c, si, i, sr, r, args);
   printf (" args = %07o \n", args);
   return 0;
}
```

The commands to compile and execute `SimVal.p` and `SimVal.cc`

```
hostname% pc -c SimVal.p
hostname% CC SimVal.o SimValMain.cc -lpc
hostname% a.out
args = 111111
```

Function Return Values

Function return values match types in the same manner as with parameters. They pass in much the same way.

Simple Types

Simple types pass in a straightforward way, as in the following example:

The Pascal function, `RetReal.p`

```
function RetReal (r: real): real;

begin
   RetReal := r + 1
end;
```

The C++ main program, `RetRealMain.cc`

```
#include <stdio.h>

extern "C" double RetReal (double);

int main(void)
{
   double  r, s;
   r = 2.0;

   s = RetReal (r);

   printf (" %f \n", s);

}
```

The commands to compile and execute `RetReal.p` and `RetRealMain.cc`

```
hostname% pc -c RetReal.p
hostname% CC RetReal.o RetRealMain.cc -lpc
hostname% a.out
3.000000
```

Type `shortreal`

The Pascal function, `RetShortReal.p`

```
function RetShortReal (r: shortreal): shortreal;

begin
   RetShortReal := r + 1.0
end;
```

The C++ main program, `RetShortRealMain.cc`

```
#include <stdio.h>
#include <math.h>

extern "C" float RetShortReal (float);

int main(void)
{
   float  r, s;
   r = 2.0;

   s = RetShortReal(r);

   printf (" %8.6f \n", s);

}
```

The commands to compile and execute `RetShortReal.p` and `RetRealMain.cc`

```
hostname% pc -c RetShortReal.p
hostname% CC RetShortReal.o RetShortRealMain.cc -lpc
hostname% a.out
3.000000
```

Input and Output

The Pascal function, `IO.p`

```
procedure IO;
begin
  writeln ('Hello, Pascal & St.Petersburg !');
end;
```

The C++ main program, `IOMain.cc`

```
#include <stdio.h>

extern "C" {
   void IO ();
};

int main(void)
{
   IO ();

   printf ("Hello, C++ ! \n");

}
```

The commands to compile and execute `IO.p` and `IOMain.cc`

```
hostname% pc -c IO.p
hostname% CC IO.o IOMain.cc -lpc
hostname% a.out
Hello, Pascal & St.Petersburg !
Hello, C++ !
```

Procedure Calls: Pascal–C++

A Pascal main program can also call C++ functions. The following examples show you how to pass simple types and arguments and include the commands that are used to compile and execute the final programs.

Arguments Passed by Reference

Pascal arguments can be passed by reference. Here we discuss how they work with C++.

Simple Types Passed by Reference

Simple types pass in a straightforward manner, as follows:

The C++ function, `SimRef.cc`

```
extern "C"
void SimRef (

   char    &t,
   char    &f,
   char    &c,
   int     &i,
   short   &s,

   float   &r,
   double  &d)
{
   t = 1;
   f = 0;
   c = 'z';
   i = 9;
   s = 9;

   r = 9.9;
   d = 9.9;
}
```

The Pascal main program, `SimRefMain.p`

```
program SimRefMain (output);
var
   t, f: boolean;
   c: char;
   i: integer;
   s: integer16;

   r: shortreal;
   d: real;

procedure SimRef (
   var t, f: boolean;
   var c: char;
   var i: integer;
   var s: integer16;

   var r: shortreal;
   var d: real
   ); external C;

begin
   SimRef (t, f, c, i, s, r, d);
   writeln (t, f: 6, c: 2, i: 2, s: 2, r: 4: 1, d: 4: 1);
end.
```

The commands to compile and execute `SimRef.cc` and `SimRefMain.p`

```
hostname% CC -c SimRef.cc
hostname% pc SimRef.o SimRefMain.p
hostname% a.out
true false z 9 9 9.9 9.9
```

Arguments Passed by Value

Pascal arguments can also be passed by value. Here is how they work with C++.

Simple Types

Simple types match with value parameters. See the following example:

The C++ function, `SimVal.cc`

```
extern "C" void SimVal(
  char   t,
  char   f,
  char   c,
  short  si,
  int    i,
  float  sr,
  double r,
  int&   Reply)
{
  Reply = 0;
  if (t)                 Reply +=       01;
  if (! f)               Reply +=      010;
  if (c == 'z')          Reply +=     0100;
  if (si == 9)           Reply +=    01000;
  if (i == 9)            Reply +=   010000;
  if (sr == (float)9.9)  Reply +=  0100000;
  if (r == 9.9)          Reply += 01000000;
}
```

The Pascal main program, SimValMain.p

```
program SimValMain(output);
var
  t: boolean  := true;
  f: boolean  := false;
  c: char     := 'z';
  si:integer16:= 9;
  i: integer  := 9;
  sr:shortreal:= 9.9;
  r: real     := 9.9;
  args: integer;

procedure SimVal(
  t, f: boolean;
  c: char;
  si:integer16;
  i: integer;
  sr:shortreal;
  r: real;
  var Reply: integer); external C;

begin
  SimVal(t, f, c, si, i, sr, r, args);
  writeln(' args = ', args :7 oct);
end.
```

The commands to compile and execute SimVal.cc and SimValMain.p

```
hostname% CC -c SimVal.cc
hostname% pc SimVal.o SimValMain.p
hostname% a.out
args = 111111
```

Function Return Values

Function return values match types in the same manner as with parameters. They pass in much the same way.

The following example shows how to pass simple types:

The C++ function, `RetReal.cc`

```
extern "C"
double RetReal (double &x)
{
  return (x + 1.0);
}
```

The Pascal main program, `RetRealMain.p`

```
program RetRealMain (output);
var
   r, s: real;

function RetReal (var x: real): real; external C;

begin
   r := 2.0;
   s := RetReal (r);
   writeln ( r: 4: 1,' Return - ', s: 4: 1);
end.
```

The commands to compile and execute `RetReal.cc` and `RetRealMain.p`

```
hostname% CC -c RetReal.cc
hostname% pc RetReal.o RetRealMain.p
hostname% a.out
  2.0 Return -  3.0
```

Global Variables in C++ and Pascal

If the types are compatible, a global variable can be shared between C++ and Pascal. See this example:

The Pascal procedure, `GloVar.p`

```
var
   Year: integer;

procedure GloVar;

begin
   Year := 1995;
end;
```

The C++ main program, `GloVarMain.cc`

```
#include <stdio.h>

extern "C" void GloVar ();

int Year;

int main(void)
{
   Year = 2042;
   GloVar ();
   printf (" %d \n", Year);
}
```

The commands to compile and execute `GloVar.p` and `GloVarMain.cc`

```
hostname% pc -c GloVar.p
hostname% CC GloVar.o GloVarMain.cc -lpc
hostname% a.out
1995
```

Pascal File Pointers to C++

You can pass a file pointer from Pascal to C++, then have C++ do the I/O. See this example.

The C++ procedure, `UseFilePtr.cc`

```
#include <stdio.h>

extern "C"
void UseFilePtr (FILE* ptr)
{
   fprintf (ptr, "[1] \n");
   fprintf (ptr, "[2] \n");
   fprintf (ptr, "[3] \n");
}
```

The C++ main program, `UseFilePtrMain.p`

```
program UseFilePtrMain (output);
var
   f: text;
   cfile: univ_ptr;

procedure UseFilePtr (cf: univ_ptr); external C;

begin
   rewrite (f, 'myfile.data');
   cfile := getfile (f);
   UseFilePtr (cfile);
end.
```

The commands to compile and execute `UseFilePtr.cc` and `UseFilePtrMain.p`

```
hostname% CC -c UseFilePtr.cc
hostname% pc UseFilePtr.o UseFilePtrMain.p
hostname% a.out
[1]
[2]
[3]
```

The FORTRAN–Pascal Interface 8

This chapter describes how to mix FORTRAN 77 and Pascal modules in the same program. It contains the following sections:

Compiler Mixed-Language Programs

When you compile with the `-v` (verbose) option, the Pascal driver brings in the runtime libraries for the main module.

However, when you compile a module that is not the main module, and which is written in a language different from the main module, you must explicitly bring in the runtime library on the command-line.

For example, you must use the compiler options `–lpfc` and `–lpc` when you compile a FORTRAN main routine that calls Pascal. The `–lpfc` option links the common startup code for programs containing mixed Pascal and FORTRAN object libraries. The `–lpc` option includes the Pascal object library, `libpc`.

Specify –lpfc on the command-line before –lpc. For example:

```
hostname% pc -c my_pascal.p
hostname% f77 my_pascal.o my_fortran.f -lpfc -lpc
Sampmain.f:
 MAIN:
```

The -c option to pc produces an unlinked object file.

When you compile a Pascal main routine that calls FORTRAN, you must use the compiler options –lpfc and –lF77. The –lF77 option links the FORTRAN object library, libf77.

You must specify -lpfc on the command-line before -lF77. For example:

```
hostname% f77 -c my_fortran.f
hostname% pc my_fortran.o my_pascal.p -lpfc -lF77
my_fortran.f:
    MAIN:
```

You can omit the libraries if the foreign language module does not interact with the runtime environment, that is, it does no I/O, memory allocation, and so on. However, there is no overhead to linking to an unused library; therefore, always link in the appropriate runtime libraries, even if you think you may not need them.

Compatibility of Types for FORTRAN and Pascal

Table 8-1 lists the default sizes and alignments of compatible types for FORTRAN and Pascal.

Table 8-1 Default Sizes and Alignments of Compatible Types (Pascal and FORTRAN)

Pascal Type	FORTRAN Type	Size (Bytes)	Alignment (Bytes)
`double`	`double precision`	8	8
`longreal`	`double precision`	8	8
`real`	`double precision`	8	8
`single`	`real`	4	4
`shortreal`	`real`	4	4
`integer16`	`integer*2`	2	2
`integer32`	`integer*4`	4	4
`integer`	`integer*4`	4	4
`-128..127`	`logical*1`, byte, or character	1	1
`boolean`	`logical*1`, byte, or character	1	1
`alfa`	`character*10`	10	1
`char`	`character`	1	1
`string`	`character*80`	80	1
varying[*n*] of `char`	structure /v/ `integer*4` `character*n` end structure	-	4
`array`	`array`	Same as element type	
`record`	structure	-	4

Table 8-2 lists the default sizes and alignments of compatible types for FORTRAN and Pascal with the `-xl` option:

Table 8-2 Sizes and Alignments of Compatible Types (Pascal and FORTRAN) with `–xl`

Pascal Type	FORTRAN Type	Size (Bytes)	Alignment (Bytes)
`real`	`real`	4	4
`integer`	`integer*2`	2	2

Precautions with Compatible Types

This section describes the precautions you must take when working with character strings and array indexes.

Character Strings

There are some precautions to take with character strings regarding the null byte, passing by value, and static storage:

- Set the byte count to zero before calling Pascal with a null varying string, because that is what Pascal assumes about such strings.
- Pass a structure consisting of a 4-byte integer and an array of characters from FORTRAN to a Pascal procedure, expecting a `var` parameter that is a variable-length string.
- Pass by reference by making the strings `var`, `in`, `out`, or `in out` parameters.
- Set the string to constant because FORTRAN and Pascal each guarantees the null byte only if the string is a constant. Neither of them relies on the null byte, which is not required by the ISO Pascal Standard.

Array Indexes

The Pascal and FORTRAN array indexes can start at any integer; be sure they match.

Incompatibilities

There are several incompatibilities between Pascal and FORTRAN variant records, enumerated types, and set types.

Variant Records

In general, Pascal variant records require adjustment of alignment to match with FORTRAN unions and structures.

Enumerated Types

Pascal enumerated types have no comparable type in FORTRAN.

Pascal Set Types

In Pascal, a set type is implemented as a bit vector, which is similar to a FORTRAN 16-bit word. Direct access to individual elements of a set is highly machine-dependent and should be avoided.

Multidimensional Arrays

Pascal multidimension arrays are incompatible with FORTRAN multi-dimension arrays. Since Pascal arrays use row-major indexing, and FORTRAN arrays use column-major indexing, an array passed in either direction appears to be transposed.

General Parameter-Passing in FORTRAN and Pascal

A few general rules apply to passing parameters:

- By default, FORTRAN passes all parameters by reference.
- In FORTRAN, if you want to pass anything by value, then you must explicitly use the nonstandard function `%VAL()`.
- Pascal passes all parameters by value unless you explicitly state that they are `var`, `out`, or `in out` parameters, in which case they are passed by reference.
- The routine options `nonpascal`, `extern fortran`, and `external fortran` pass by reference.

Procedure Calls: FORTRAN-Pascal

Here are examples of how a FORTRAN main program calls a Pascal procedure.

The Pascal procedure, `Samp.p`. Note the procedure definition. The procedure name in the procedure statement is in lowercase with a trailing underscore (_). This format is required to match the conventions of the FORTRAN compiler. `var` parameters are used to match FORTRAN defaults.

```
procedure samp_(var i: integer; var r: real);

begin
    i := 9;
    r := 9.9
end; { samp_ }
```

The FORTRAN main program, `Sampmain.f`. Note the procedure declaration and call. FORTRAN converts to lowercase by default; you do not explicitly give the underscore (_).

```
integer i
        double precision  d

        call Samp ( i, d )
        write( *, '(I2, F4.1)')  i, d
        stop
        end
```

The commands to compile and execute `Samp.p` and `Sampmain.f`

```
hostname% pc -c Samp.p
hostname% f77 Samp.o Sampmain.f -lpfc -lpc
Sampmain.f:
 MAIN:
hostname% a.out
 9 9.9
```

Variable Parameters

Pascal passes all `var` parameters by reference, FORTRAN's default.

Simple Types without the `–xl` Option

With `var` parameters, simple types match.

See the following example:

The Pascal procedure, SimVar.p

```
procedure simvar_(var t, f: boolean; var c: char;
                  var i: integer; var r: real;
                  var si: integer16; var sr: shortreal);

begin
    t := true;
    f := false;
    c := 'z';
    i := 9;
    r := 9.9;
    si := 9;
    sr := 9.9
end; { simvar_ }
```

The FORTRAN main program, SimVarmain.f

```
        logical*1         t, f
        character         c
        integer*4         i
        double precision  d
        integer*2         si
        real              sr

        call SimVar ( t, f, c, i, d, si, sr )

        write(*, "(L2,L2,A2,I2,F4.1,I2,F4.1)")
        &                  t, f, c, i, d,   si,sr
        stop
        end
```

The commands to compile and execute SimVar.p and SimVarmain.f

```
hostname% pc -c SimVar.p
hostname% f77 SimVar.o SimVarmain.f -lpfc -lpc
SimVarmain.f:
 MAIN:
hostname% a.out
 T F z 9 9.9 9 9.9
```

Simple Types with the -xl Option

When you pass the `-xl` option, the Pascal data type `real` must be paired with a FORTRAN data type `real`; the Pascal data type `integer` must be paired with a FORTRAN data type, `integer*2`.

Strings of Characters

The FORTRAN counterpart to the Pascal `alfa` and `string` types is a character string, and the FORTRAN counterpart to the Pascal `varying` is a structure. By default, FORTRAN, passes all by reference:

The Pascal procedure, `StrVar.p`

```
type
    varstr = varying [25] of char;

procedure strvar_(var a: alfa; var s: string;
                  var v: varstr);

begin
    a := 'abcdefghij';
    s := 'abcdefghijklmnopqrtstuvwxyz';
    v := 'oyvay'
end; { strvar_ }
```

The FORTRAN main program, `StrVarmain.f`

```
           structure /VarLenStr/
           integer   nbytes
           character a*25
        end structure
        record /VarLenStr/ vls
        character s10*10, s80*80, s25*25
        vls.nbytes = 0
        Call StrVar( s10, s80, vls )
        s25(1:5) = vls.a(1:vls.nbytes)
        write (*, 1) s10, s80, s25
 1      format("s10='", A, "'",
&            / "s80='", A, "'",
&            / "s25='", A, "'"  )
        end
```

The commands to compile and execute `StrVar.p` and `StrVarmain.f`

```
hostname% pc -c StrVar.p
hostname% f77 StrVar.o StrVarmain.f -lpfc -lpc
StrVarmain.f:
    MAIN:
hostname% a.out
s10='abcdefghij'
s80='abcdefghijklmnopqrtstuvwxyz
s25='oyvay'
```

Fixed Arrays

For a fixed array parameter, pass the same type and size by reference, as shown in the following example:

The Pascal procedure, `FixVec.p`

```
type
    VecTyp = array [0..8] of integer;

procedure fixvec_(var V: VecTyp; var Total: integer);

var
    i: integer;

begin
    Total := 0;
    for i := 0 to 8 do
      Total := Total + V[i]
end; { fixvec_ }
```

The FORTRAN main program, `FixVecmain.f`

```
        integer Sum
        integer a(9)
        data    a / 1,2,3,4,5,6,7,8,9 /
        call FixVec ( a, Sum )
        write( *, "( I3 )")  Sum
        stop
        end
```

The commands to compile and execute `FixVec.p` and `FixVecmain.f`

```
hostname% pc -c FixVec.p
hostname% f77 FixVec.o FixVecmain.f -lpfc -lpc
hostname% a.out
FixVecmain.f:
 MAIN:
    45
```

The `univ` *Arrays*

You can pass any size array to a Pascal procedure expecting a `univ` array, but there is no advantage in doing so, since there is no type or size checking for separate compilations. However, if you want to use an existing Pascal procedure that has a `univ` array, you can do so. All `univ` arrays that are `in`, `out`, `in out`, or `var` parameters pass by reference.

The Pascal procedure, `UniVec.p`, which defines a 10-element array

```
type
    VecTyp = array [0..9] of integer;

procedure univec_(in V:univ VecTyp; var Last: integer;
                  var Total: integer);

var
    i: integer;

begin
    Total := 0;
    for i := 0 to Last do
      Total := Total + V[i]
end; { univec_ }
```

The FORTRAN main program, `UniVecmain.f`, which passes a 3-element array to the Pascal procedure written to do a 10-element array

```
        integer  Sum
        integer a(0:2)
        data    a / 7, 8, 9 /
        call UniVec ( a, 2, Sum )
        write( *, "( I3 )")  Sum
        stop
        end
```

The commands to compile and execute `UniVec.p` and `UniVecmain.f`

```
hostname%  pc -c UniVec.p
hostname% f77 UniVec.o UniVecmain.f -lpfc -lpc
UniVecmain.f:
    MAIN:
hostname% a.out
 24
```

Conformant Arrays

For conformant arrays, with single-dimension array, pass upper and lower bounds, placed after the declared parameter list, as in:

```
function ip(var x:array[lb..ub:integer] of real):real;
     ...

double precision v1(10)
double precision z
z = ip ( v1, %VAL(0), %VAL(9) )
...
```

Pascal passes the bounds by value, so FORTRAN must pass them by value, too.

One bounds pair may apply to several arrays if they are declared in the same parameter group:

```
function ip(var x,y:array[lb..ub:integer] of real):real;
    ...

double precision v1(10), v2(10)
double precision z
z = ip ( v1, v2, %VAL(0), %VAL(9) )
    ...
```

Examples of single-dimension array and array of character conformant arrays follow. Conformant arrays are included here only because they are a relatively standard feature; there are usually more efficient and simpler ways to do that.

Example 1: Single-Dimension Array

The Pascal procedure, IntCA.p. Pascal passes the bounds by value.

```
procedure intca_(var a: array [lb..ub: integer] of integer);

begin
    a[1] := 1;
    a[2] := 2
end; { intca_ }
```

The FORTRAN main program, IntCAmain.f

```
        integer k
        integer s(0:2)
        data    s  / 0, 0, 0 /
        call IntCA ( s, %VAL(0), %VAL(2) )
        do k = 0, 2
        write( *, "(I1)" )  s(k)
        end do
        stop
        end
```

The commands to compile and execute IntCA.p and IntCAmain.f

```
hostname% pc -c IntCA.p
hostname% f77 IntCA.o IntCAmain.f -lpfc -lpc
IntCAmain.f:
 MAIN:
hostname% a.out
0
1
2
```

Example 2: Array of Characters

The Pascal procedure, ChrCA.p. Pascal passes the bounds by value.

```
procedure chrca_(var a: array [lb..ub: integer] of char);

begin
    a[0] := 'T';
    a[13] := 'o'
end; { chrca_ }
```

The FORTRAN main program, `ChrCAmain.f`

```
         character s*16
         data  s / "this is a string" /
         call ChrCA( s, %VAL(0), %VAL(15) )
         write( *, "(A)" )  s
         stop
         end
```

The commands to compile and execute `ChrCA.p` and `CharCAmain.f`

```
hostname% pc -c ChrCA.p
hostname% f77 ChrCA.o ChrCAmain.f -lpfc -lpc
ChrCAmain.f:
   MAIN:
hostname% a.out
This is a string
```

Records and Structures

In most cases, a Pascal record describes the same objects as its FORTRAN structure equivalent, provided that the components have compatible types and are declared in the same order. The compatibility of the types depends mostly on size and alignment.

For more information, see “Compatibility of Types for FORTRAN and Pascal” on page 168.

A Pascal record of an integer and a character string matches a FORTRAN structure of the same. Consider these examples:

The Pascal procedure, `StruChr.p`

```
type
    lenstr =
      record
          nbytes: integer;
          chrstr: array [0..25] of char
      end;

procedure struchr_(var v: lenstr);

begin
    v.chrstr := 'oyvay';
    v.nbytes := 5
end; { struchr_ }
```

The FORTRAN main program, `StruChrmain.f`

```
        structure /VarLenStr/
           integer   nbytes
           character a*25
        end structure
        record /VarLenStr/ vls
        character s25*25
        vls.nbytes = 0
        Call StruChr( vls )
        s25(1:5) = vls.a(1:vls.nbytes)
        write ( *, 1 )  s25
 1      format("s25='", A, "'" )
        stop
        end
```

The commands to compile and execute `Struchr.p` and `StruChrmain.f`

```
hostname% pc -c StruChr.p
hostname% f77 StruChr.o StruChrmain.f -lpfc -lpc
StruChrmain.f:
   MAIN:
hostname% a.out
s25='oyvay'
```

Variant Records

FORTRAN equivalents of variant records can sometimes be constructed, although there is some variation with architecture, and sometimes you need to adjust the alignment.

The Pascal procedure, VarRec.p

```
type vr = record
    case tag: char of
        'a': ( ch1, ch2: char ) ;
        'b': ( flag: boolean ) ;
        'K': ( ALIGN: integer ) ;
    end ;

procedure varrec_ ( var Rec: vr ) ;

begin
    if ( Rec.ch1 = 'a' )
        then Rec.ch2 := 'Z'
end; { VarRec.p }
```

The FORTRAN main program, VarRecmain.f. The variable ALIGN is integer*2, and is needed to match the Pascal variant record layout.

```
        structure /a_var/
                character ch1, ch2
        end structure
        structure /b_var/
                character flag
        end structure
        structure /c_var/
                integer*2 ALIGN
        end structure
        structure /var_part/
                union
                   map
                        record /a_var/ a_rec
                   end map
                   map
                        record /b_var/ b_rec
                   end map
                   map
                        record /c_var/ c_rec
                   end map
                end union
        end structure
        structure /vrnt/
                character tag
                record /var_part/ var_rec
        end structure
        record /vrnt/ VRec
        VRec.var_rec.a_rec.ch1 = 'a'
        VRec.var_rec.a_rec.ch2 = 'b'
        call varrec ( VRec )
        write ( *, * )  VRec.var_rec.a_rec.ch2
        stop
        end
```

The commands to compile and execute VarRec.p and VarRecmain.f without –xl

```
hostname% pc -c VarRec.p
hostname% f77 VarRec.o VarRecmain.f
VarRecmain.f:
   MAIN:
hostname% a.out
b
```

Pascal Set Type

The Pascal set type is incompatible with FORTRAN.

Pascal `intset` *Type*

The Pascal `intset` type is predefined as `set of [0..127]`. A variable of this type takes a minimum of 16 bytes of storage.

The Pascal procedure, `IntSetVar.p`, which has an `intset` of the elements `[1, 3, 7, 8]`

```
procedure intsetvar_(var s: intset);

begin
    s := [1, 3, 7, 8]
end; { intsetvar_ }
```

The FORTRAN main program, `IntSetVarmain.f`

```
       integer*2 s(8)
        pointer ( ps, s )
        ps = malloc(16)
        call IntSetVar ( s )
        do  i = 5, 8
           write( *, 1 )  s(i), i
        end do
 1      format(o3,1x, 'octal   (word', i2, ')')
        write( *, "('110 001 010 (binary, word 8)')")
        write( *, "('876 543 210 (bit nos, word 8)')")
        stop
```

The commands to compile and execute `IntSetVar.p` and `IntSetVarmain.f`. The output of this example depends on the architecture of your machine.

```
hostname% pc -c IntSetVar.p
hostname% f77 IntSetVar.o IntSetVarmain.f -lpfc -lpc
IntSetVarmain.f:
  MAIN:
hostname% a.out
     0 octal          (word 5)
     0 octal          (word 6)
     0 octal          (word 7)
612 octal             (word 8)
110 001 010           (binary, word 8)
876 543 210           (bit nos, word 8)
```

Value Parameters

In general, Pascal passes value parameters on the stack.

Simple Types without the –`xl` Option

Without the `-xl` option, simple types match.

See the following example:

The Pascal procedure, SimVal.p. t, f, c, i, r, and s are value parameters.

```
procedure simval_(t, f: boolean; c: char; i: integer;
                  r: real; s: integer16; var reply: integer);

begin
    reply := 0;
    { If nth arg is ok, set nth octal digit to one. }
    if t then
      reply := reply + 1;
    if not f then
      reply := reply + 8;
    if c = 'z' then
      reply := reply + 64;
    if i = 9 then
      reply := reply + 512;
    if r = 9.9 then
      reply := reply + 4096;
    if s = 9 then
      reply := reply + 32768
end; { simval_ }
```

The FORTRAN main program, SimValmain.f

```
        logical*1         t, f
        character         c
        integer*4         i
        double precision  d
        integer*2         s
        integer*4         args
        data t / .true. /,  f / .false. /,  c / 'z' /
&            i / 9 /,       d / 9.9 /,      s / 9 /

        call SimVal( %VAL(t), %VAL(f), %VAL(c),
&                        %VAL(i), %VAL(d), %VAL(s), args )
        write( *, 1 )  args
 1      format('args=', o6, '(If nth digit=1, arg n OK)')
        stop
        end
```

The commands to compile and execute `SimVal.p` and `SimValmain.f`

```
hostname% pc -c SimVal.p
hostname% f77 SimVal.o SimValmain.f -lpfc -lpc
SimValmain.f:
   MAIN:
hostname% a.out
args=111111(If nth digit=1, arg n OK)
```

Simple Types with the `–xl` *Option*

With the `-xl` option, match Pascal `real` with FORTRAN `real` and Pascal `integer` with FORTRAN `integer*2`.

You can pass by value using the `%VAL()` feature of FORTRAN.

Type `shortreal`

Unlike C, there is no problem with passing `shortreal` value parameters between Pascal and FORTRAN. They can be passed exactly as in the previous example, with the Pascal `shortreal` type matching the FORTRAN `real` type.

Arrays

Since FORTRAN cannot pass arrays by value, it cannot pass strings of characters, fixed arrays, or `univ` arrays by value.

Conformant Arrays

Although Pascal generally passes all value parameters on the stack, the exception is value-conformant array parameters, which are handled by creating a copy in the caller environment and passing a pointer to the copy. In addition, the bounds of the array must be passed. See “Conformant Arrays” on page 177.

This example is the same as the one in the earlier section, except that the `var` prefix is deleted.

Pascal procedure, ChrCAx.p

```
procedure chrca_ ( a: array [lb..ub:integer] of char) ;

begin
        a[0] := 'T' ;
        a[13] := 'o' ;
end; { chrca_ }
```

The FORTRAN main program, ChrCAmain.f

```
        character s*16
        data  s / "this is a string" /
        call ChrCA( s, %VAL(0), %VAL(15) )
        write( *, "(A)" )  s
        stop
        end
```

The commands to compile and execute ChrCAx.p and ChrCAmain.f

```
hostname% pc -c ChrCAx.p
hostname% f77 ChrCAx.o ChrCAmain.f -lpfc -lpc
ChrCAmain.f:
   MAIN:
hostname% a.out
This is a string
```

Pointers

Pointers are easy to pass, as shown in the following example:

The Pascal procedure, PassPtr.p. In the Pascal procedure statement, the name must be all in lowercase, with a trailing underscore (_).

```
type
    PtrInt  = ^integer ;
    PtrReal = ^real ;
procedure passptr_ ( var iPtr: PtrInt ;
                     var dPtr: PtrReal ) ;
begin
    iPtr^ := 9 ;
    dPtr^ := 9.9 ;
end ;
```

The FORTRAN main program, `PassPtrmain.f`. In the FORTRAN main program, the name is converted to lowercase. Uppsercase is ignored.

```
        program PassPtrmain
        integer          i
        double precision d
        integer          iptr, dptr
        pointer ( iPtr, i ), ( dPtr, d )
        iPtr = malloc( 4 )
        dPtr = malloc( 8 )
        i = 0
        d = 0.0
        call PassPtr ( iPtr, dPtr )
        write( *, "(i2, f4.1)" )  i, d
        stop
        end
```

The commands to compile and execute `PastPtr.p` and `PassPtrmain.f`

```
hostname% pc -c PassPtr.p
hostname% f77 PassPtr.o PassPtrmain.f -lpfc -lpc
PassPtrmain.f:
   MAIN passptrmain:
hostname% a.out
9 9.9
```

Function Return Values

Function return values match types the same as with parameters, and they pass in much the same way. See “Procedure Calls: FORTRAN-Pascal” on page 172.

Simple Types

The simple types pass in a straightforward way, as follows:

The Pascal function, `RetReal.p`

```
function retreal_(var x: real): real;

begin
    retreal_ := x + 1
end; { retreal_ }
```

The FORTRAN main program, `RetRealmain.f`

```
        double precision r, s, RetReal
        r = 2.0
        s = RetReal( r )
        write( *, "(2f4.1)") r, s
        stop
        end
```

The commands to compile and execute `RetReal.p` and `RetRealmain.f` without `–xl`

```
hostname% pc -c RetReal.p
hostname% f77 RetReal.o RetRealmain.f -lpfc -lpc
RetRealmain.f:
   MAIN:
hostname% a.out
 2.0 3.0
```

Type `shortreal`

There is no problem with returning a `shortreal` function value between Pascal and FORTRAN. As in the previous example, it can be passed exactly, with the Pascal `shortreal` type matching the FORTRAN `real` type (without `-xl`).

Procedure Calls: Pascal-FORTRAN

This section parallels “Procedure Calls: FORTRAN-Pascal” on page 172. The comments and restrictions given in that section apply here, also.

Variable Parameters

Pascal passes all `var` parameters by reference, the FORTRAN default.

Simple Types

Simple types pass in a straightforward manner, as follows:

The FORTRAN subroutine, `SimVar.f`

```
        subroutine SimVar ( t, f, c, i, d, si, sr )
        logical*1        t, f
        character        c
        integer          i
        double precision d
        integer*2        si
        real             sr
        t = .true.
        f = .false.
        c  = 'z'
        i  = 9
        d  = 9.9
        si = 9
        sr = 9.9
        return
        end
```

The Pascal main program, `SimVarmain.p`

```
program SimVarmain(output);

var
    t, f: boolean;
    c: char;
    i: integer;
    r: real;
    si: integer16;
    sr: shortreal;

procedure simvar(var t, f: boolean; var c: char;
                 var i: integer; var r: real;
                 var si: integer16; var sr: shortreal);
                 external fortran;

begin
    simvar(t, f, c, i, r, si, sr);
    writeln(t, f: 6, c: 2, i: 2, r: 4: 1, si: 2, sr: 4: 1)
end. { SimVarmain }
```

The commands to compile and execute `SimVar.p` and `SimVarmain.p`

```
hostname% f77 -c SimVar.f
SimVar.f:
    simvar:
hostname% pc SimVar.o SimVarmain.p -lpfc -lF77
hostname% a.out
true false z 9 9.9 9 9.9
```

Strings of Characters

The `alfa` and `string` types pass simply; varying strings are a little tricky. All pass by reference.

The FORTRAN subroutine, `StrVar.f`

```
        subroutine StrVar ( s10, s80, vls )
        character   s10*10, s80*80
        structure /VarLenStr/
           integer  nbytes
           character a*25
        end structure
        record /VarLenStr/ vls
        character ax*10, sx*80, vx*5
        data ax   / "abcdefghij" /,
&          sx / "abcdefghijklmnopqrstuvwxyz" /,
&          vx / "oyvay" /
        s10(1:10) = ax(1:10)
        s80(1:80) = sx(1:80)
        vls.a(1:5) = vx(1:5)
        vls.nbytes = 5
        return
        end
```

The Pascal main program, `StrVarmain.p`

```
program StrVarmain(output);

type
    varstr = varying [25] of char;

var
    a: alfa;
    s: string;
    v: varstr;

procedure strvar(var xa: alfa; var xs: string;
                 var xv: varstr); external fortran;

begin
    strvar(a, s, v);
    writeln(a);
    writeln(s);
    writeln(v);
    writeln('length(v)= ', length(v): 2)
end. { StrVarmain }
```

The commands to compile and execute `StrVar.f` and `StrVarmain.p`

```
hostname% f77 -c StrVar.f
StrVar.f:
       strvar:
hostname% pc StrVar.o StrVarmain.p -lpfc -lF77
hostname% a.out
abcdefghij
abcdefghijklmnopqrstuvwxyz
oyvay
length(v)= 5
```

Character Dummy Arguments

When you call FORTRAN 77 routines with character dummy arguments from Pascal programs—that is, routines in which string arguments are specified as `character*(*)` in the FORTRAN source, there is no explicit analogue in Pascal.

So, if you try to simply pass an actual string and specify the FORTRAN routine as `extern fortran`, the program fails, because implementation of this type of arguments implies that the actual length of the string is implicitly passed as an extra value argument after the string pointer.

To specify this routine in Pascal, declare it as having two arguments: a VAR argument of string type for the string pointer, and an extra value argument of `integer32` type for the string length.

It is incorrect to specify the routine as `extern fortran` because Pascal passes all arguments to FORTRAN routines by reference. Consequently, to pass this type of argument, you must:

- Declare two arguments as described above, specifying the routine as simply `external` (without the `fortran` directive)
- Add a trailing underscore to the routine name in a Pascal program

The following example illustrates this method:

The Pascal program, `sun.pas`

```
program Test(input,output);
var
  s : string;

procedure mygrout_(var prompt :string; length :integer32); external;

begin
  writeln('Starting...');
  s := 'Trio Jeepy';
  mygrout_(s, 8);
  writeln('Ending...')
end.
```

The FORTRAN subroutine, `mygrout.f`

```
        subroutine MyGrout(N)
        character*(*)N
        write(6,*) N
        return
        end
```

The commands to compile and run this program

```
hostname% pc -g -c sun.pas
hostname% f77 -g sun.o mygrout.f -lpc
mygrout.f:
        mygrout:
hostname% a.out
Starting...
 Trio Jee
Ending...
```

Fixed Arrays

For a fixed-array parameter, pass the same type and size by reference:

The FORTRAN subroutine, `FixVec.f`

```
       subroutine FixVec ( V, Sum )
       integer Sum
       integer V(0:8)
       integer i
       Sum = 0
       do  2 i = 0, 8
 2     Sum = Sum + V(i)
       return
       end
```

The Pascal main program, `FixVecmain.p`

```
program FixVecmain(output);

type
    VecTyp = array [0..8] of integer;

var
    V: VecTyp := [1, 2, 3, 4, 5, 6, 7, 8, 9];
    Sum: integer;

procedure fixvec(var XV: VecTyp; var XSum: integer);
    external fortran;

begin
    fixvec(V, Sum);
    writeln(Sum: 4)
end. { FixVecmain }
```

The commands to compile and execute `FixVec.f` and `FixVecmain.p`

```
hostname% f77 -c FixVec.f
FixVec.f:
    fixvec:
hostname% pc FixVec.o FixVecmain.p -lpfc -lF77
hostname% a.out
   45
```

The univ Arrays

The `univ` arrays that are `in`, `out`, `in out`, or `var` parameters pass by reference.

The FORTRAN subroutine, `UniVec.f`

```
        subroutine UniVec ( V, Last, Sum )
        integer V(0:2), Last, Sum, i
        Sum = 0
        do i = 0, Last
                Sum = Sum + V(i)
        end do
        return
        end
```

The Pascal main program, `UniVecmain.p`

```
program UniVec;

type
    VecTyp = array [0..9] of integer;

procedure univec(var V:univ VecTyp; in Last: integer;
                 var Sum: integer); external fortran;

var
    Sum: integer;
    V: array [0..2] of integer;

begin
    V[0] := 7;
    V[1] := 8;
    V[2] := 9;
    univec(V, 2, Sum);
    writeln(Sum)
end. { UniVec }
```

The commands to compile and execute `UniVec.f` and `UniVecmain.p`

```
hostname% f77 -c UniVec.f
UniVec.f:
    univec:
hostname% pc UniVec.o UniVecmain.p -lpfc -lF77
hostname% a.out
    24
```

Conformant Arrays

Pascal-conformant array parameters are not compatible if Pascal calls FORTRAN.

Records and Structures

Records and structures pass as follows:

The FORTRAN subroutine, `StruChr.f`

```
         subroutine StruChr ( vls )
         structure /VarLenStr/
            integer   nbytes
            character a*25
         end structure
         record /VarLenStr/ vls
         vls.a(1:5) = 'oyvay'
         vls.nbytes = 5
         return
         end
```

The Pascal main program, `StruChrmain.p`

```
program StruChrmain;

type
    lenstr =
      record
          nbytes: integer;
          chrstr: array [0..25] of char
      end;

var
    v: lenstr;

procedure struchr(var v: lenstr);
    external fortran;

begin
    struchr(v);
    writeln('v.chrstr = "', v.chrstr, '"');
    writeln('v.nbytes =', v.nbytes: 2)
end. { StruChrmain }
```

The commands to compile and execute `StruChr.f` and `StruChrmain.p`

```
hostname% f77 -c StruChr.f
StruChr.f:
     struchr:
hostname% pc StruChr.o StruChrmain.p -lpfc -lF77
hostname% a.out
v.chrstr = "oyvay"
v.nbytes = 5
```

Variant Records

You can construct FORTRAN equivalents of variant records. There is some variation with architecture, and sometimes you need to adjust the alignment.

Chapter 6, “The C–Pascal Interface,” has an example that matches the following example.

The FORTRAN subroutine, `VarRec.f`. The variable `ALIGN` is `integer*2` and is needed to match the Pascal variant record layout.

```
         subroutine VarRec ( VRec )
         structure /a_var/
                 character ch1, ch2
         end structure
         structure /b_var/
                 character flag
         end structure
         structure /c_var/
                 integer*2 ALIGN
         end structure
         structure /var_part/
                 union
                    map
                         record /a_var/ a_rec
                    end map
                    map
                         record /b_var/ b_rec
                    end map
                    map
                         record /c_var/ c_rec
                    end map
                 end union
         end structure
         structure /vrnt/
                 character tag
                 record /var_part/ var_rec
         end structure
         record /vrnt/ VRec
         if ( VRec.var_rec.a_rec.ch1 .eq. 'a' )
&                VRec.var_rec.a_rec.ch2 = 'Z'
         return
         end
```

The Pascal main program, `VarRecmain.p`

```
program VarRecmain;

type
    vr =
      record
          case tag: char of
              'a': ( ch1, ch2: char );
              'b': ( flag: boolean );
              'K': ( ALIGN: integer )
      end;

var
    Rec: vr;

procedure varrec(var d: vr); external fortran;

begin
    Rec.tag := 'a';
    Rec.ch1 := 'a';
    Rec.ch2 := 'b';
    varrec(Rec);
    writeln(Rec.ch2)
end. { VarRecmain }
```

The commands to compile and execute `VarRec.f` and `VarRecmain.p` without `–xl`

```
hostname% f77 -c VarRec.f
VarRec.f:
    varrec:
hostname% pc VarRec.o VarRecmain.p -lpfc -lF77
hostname% a.out
b
```

Value Parameters

With `external fortran` on the `procedure` statement, Pascal passes value parameters as FORTRAN expects them.

Simple Types

With `external fortran`, the procedure name in the procedure statement and in the call must be in lowercase, with no underscore (_).

The FORTRAN subroutine, `SimVal.f`

```
        subroutine  SimVal( t, f, c, i, d, s, reply )
        logical*1        t, f
        character        c
        integer*4        i
        double precision d
        integer*2        s
        integer*4        reply
        reply = 0
        if ( t          ) reply = reply + 1
        if ( .not. f    ) reply = reply + 8
        if ( c .eq. 'z' ) reply = reply + 64
        if ( i .eq. 9   ) reply = reply + 512
        if ( d .eq. 9.9 ) reply = reply + 4096
        if ( s .eq. 9   ) reply = reply + 32768
        return
        end
```

The Pascal main program, `SimValmain.p`

```
program SimVal(output);

var
    t: boolean := true;
    f: boolean := false;
    c: char := 'z';
    i: integer := 9;
    r: real := 9.9;
    s: integer16 := 9;
    args: integer;

procedure simval(t, f: boolean; c: char; i: integer;
                 r: real; s: integer16; var reply: integer);
                 external fortran;

begin
    simval(t, f, c, i, r, s, args);
    writeln('args=', args: 6 oct, ' (If nth digit=1, arg n OK.)')
end. { SimVal }
```

The commands to compile and execute `SimVal.f` and `SimValmain.p`

```
hostname% f77 -c SimVal.f
SimVal.f:
     simval:
hostname% pc SimVal.o SimValmain.p -lpfc -lF77
hostname% a.out
args=111111 (If nth digit=1, arg n OK.)
```

Pointers

Pointers are easy to pass, as shown in the following example:

The FORTRAN subroutine, `PassPtr.f`. In the FORTRAN subroutine, the name is converted to lowercase. Uppsercase is ignored.

```
      subroutine PassPtr ( iPtr, dPtr )
      integer          i
      double precision d
      pointer ( iPtr, i ), ( dPtr, d )
      i  = 9
      d  = 9.9
      return
      end
```

The Pascal main program, `PassPtrmain.p`. In the Pascal program, where it calls the FORTRAN subroutine, the name must be in lowercase.

```
program PassPtrmain;

type
    PtrInt = ^ integer;
    PtrReal = ^ real;

var
    i: integer := 0;
    r: real := 0.0;
    iP: PtrInt;
    rP: PtrReal;

procedure passptr(var xiP: PtrInt; var xrP: PtrReal);
    external fortran;

begin
    iP := addr(i);
    rP := addr(r);
    passptr(iP, rP);
    writeln(i: 2, r: 4: 1)
end. { PassPtrmain }
```

The commands to compile and execute `PassPtr.f` and `PassPtrmain.p`

```
hostname% f77 -c PassPtr.f
PassPtr.f:
        passptr:
hostname% pc PassPtr.o PassPtrmain.p -lpfc -lF77
hostname% a.out
 9 9.9
```

Function Return Values

Function return values match types the same as with parameters, and they pass in much the same way.

Simple Types

The simple types pass in a straightforward way, as in this example:

The FORTRAN function, `RetReal.f`

```
double precision function retreal ( x )
 retreal = x + 1.0
 return
 end
```

The Pascal main program, `RetRealmain.p`

```
program retrealmain;

var
    r, s: real;

function retreal(x: real): real; external fortran;

begin
    r := 2.0;
    s := retreal(r);
    writeln(r: 4: 1, s: 4: 1)
end. { retrealmain }
```

The commands to compile and execute `RetReal.f` and `RetRealmain.p`

```
hostname% f77 -c RetReal.f
RetReal.f
        retreal:
hostname% pc RetReal.o RetRealmain.p -lpfc -lF77
hostname% a.out
 2.0 3.0
```

Type `shortreal`

You can return a `shortreal` function value between Pascal and FORTRAN. Pass it exactly as in the previous example, with the Pascal `shortreal` type matching the FORTRAN `real` type (without `-xl`).

Routines as Parameters

If the passed procedure is a top-level procedure, write it as follows:

The FORTRAN subroutine, `PassProc.f`

```
    subroutine PassProc ( r, s, prcdr )
      real r, s
      external prcdr
      call prcdr ( r, s )
      return
    end
```

The Pascal main program, `PassProcmain.p`

```
program PassProcmain;

var
    a, b: real;

procedure passproc(var u: real; var v: real;
    procedure p(var r: real; var s: real));
    external fortran;

procedure AddOne(var x: real; var y: real);

begin
    y := x + 1
end; { AddOne }

begin
    a := 8.0;
    b := 0.0;
    passproc(a, b, AddOne);
    writeln(a: 4: 1, b: 4: 1)
end. { PassProcmain }
```

The commands to compile and execute `PassProc.f` and `PassProcmain.p`

```
hostname% f77 -c PassProc.f
PassProc.f
       passproc
hostname% pc PassProc.o PassProcmain.p -lpfc -lF77
hostname% a.out
 8.0 9.0
```

If the procedure is *not* a top-level procedure, then you do not deal with how to pass it, because that requires allowing for the static link. A procedure or function passed as an argument is associated with a static link to its lexical parent’s activation record.

When an outer block procedure or function is passed as an argument, Pascal passes a null pointer in the position normally occupied by the static link of the passed routine. So that procedures and functions can be passed to other languages as arguments, the static links for all procedure or function arguments are placed after the end of the conformant array bounds pairs, if any.

Routines in other languages can be passed to Pascal; a dummy argument must be passed in the position normally occupied by the static link of the passed routine. If the passed routine is not a Pascal routine, the argument is used only as a placeholder.

Error Diagnostics 9

This chapter discusses the errors you may come across while writing software programs with Pascal. It contains the following sections:

Note – Appendix B, "Error Messages," lists in numerical order all the error messages generated by Pascal.

Compiler Syntax Errors

Here are some common syntax errors in Pascal programs and the ways that the compiler handles them.

Illegal Characters

Characters such as `@` are not part of Pascal. If they are found in the source program and are not part of a string constant, a character constant, or a comment, they are considered to be illegal characters. This error can happen if you leave off a closing string quotation mark (`'`).

Most nonprinting characters in your input are also illegal, except in character constants and character strings. Except for the tab and formfeed characters, which are used to format the program, nonprinting characters in the input file print as the character ? in your listing.

String Errors

Encountering an end-of-line after an opening string quotation mark (`'`) without first encountering the matching closing quote yields the diagnostic:

```
Unmatched ' for string.
```

Also, anything enclosed in double quotes (for example, `"hello"`) is treated as a comment and is, therefore, ignored.

Programs containing # characters (other than in column 1 or in arbitrary-based integers) can produce this diagnostic, because early implementations of Pascal use # as a string delimiter. In this version, # is used for `#include` and preprocessor directives, and must be in column 1.

Digits in Real Numbers

Pascal requires digits in `real` numbers before the decimal point. Thus, the statements `b := .075;` and `c := 05e-10;` generate the following diagnostics in Pascal:

```
Mon Feb 13 10:46:44 1995 digerr.p:
           5  b:= .075;
e 18740-------------------^---  Digits required before decimal
point
           6  c:= .05e-10
e 18740-------------------^---  Digits required before decimal
point
```

These constructs are also illegal as data input to variables in `read` statements whose arguments are variables of type `real`, `single`, `shortreal`, `double`, and `longreal`.

Replacements, Insertions, and Deletions

When Pascal encounters a syntax error in the input text, the compiler invokes an error recovery procedure. This procedure examines the input text immediately after the point of error and uses a set of simple corrections to determine whether or not to allow the analysis to continue. These corrections involve replacing an input token with a different token or inserting a token. Most of these changes do not cause fatal syntax errors.

The exception is the insertion of or replacement with a symbol, such as an identifier or a number; in these cases, the recovery makes no attempt to determine which identifier or what number should be inserted. Thus, these are considered fatal syntax errors.

The Pascal program, `synerr.p`, which uses `**` as an exponentiation operator

```
program synerr_example(output);

var i, j are integer;

begin
    for j :* 1 to 20 begin
        write(j);
        i = 2 ** j;
        writeln(i))
    end
end. { synerr_example }
```

`synerr.p` produces a fatal syntax error when you compile it because Pascal does not have an exponentiation operator.

```
hostname% pc synerr.p
Mon Feb 13 10:56:19 1995 synerr.p:
           3  var i, j are integer;
e 18460----------------^---  Replaced identifier with a ':'
           6      for j :* 1 to 20 begin
E 18490-----------------^---  Expected keyword (null)
E 18460-----------------^---  Replaced ':' with a identifier
e 18480----------------------------^---  Inserted keyword do
           8          i = 2 ** j;
e 18480---------------^---  Inserted keyword if
E 18480----------------------^---  Inserted identifier
e 18480-------------------------^---  Inserted keyword then
           9          writeln(i))
e 18450-------------------------^---  Deleted ')'
```

Undefined or Improper Identifiers

If an identifier is encountered in the input but is undeclared, the error recovery mechanism replaces it with an identifier of the appropriate class.

Further references to this identifier are summarized at the end of the containing procedure, function, or at the end of the program. This is the case if the reference occurs in the main program.

Similarly, if you use an identifier in an inappropriate way, for example, if a `type` identifier is used in an assignment statement, `pc` produces a diagnostic and inserts an identifier of the appropriate class. Further incorrect references to this identifier are flagged only if they involve incorrect use in a different way. `pc` summarizes all incorrect uses in the same way it summarizes uses of undeclared variables.

Expected Symbols and Malformed Constructs

If none of the corrections mentioned previously appears reasonable, the error recovery routine examines the input to the left of the point of error to see if there is only one symbol that can follow this input. If so, the recovery prints a diagnostic which indicates that the given symbol is expected.

In cases where none of these corrections resolve the problems in the input, the recovery may issue a diagnostic that indicates "malformed" input. If necessary, `pc` can then skip forward in the input to a place where analysis can continue. This process may cause some errors in the missed text to be skipped.

See this example:

The Pascal program, `synerr2.p`. Here `output` is misspelled, and a is given a FORTRAN-style variable declaration.

```
program synerr2_example(input,outpu);

integer a(10)

begin
    read(b);
    for c := 1 to 10 do
        a(c) := b * c
end. { synerr2_example }
```

These are the error messages you receive when you compile `synerr2.p`. On line 6, parentheses are used for subscripting (as in FORTRAN), rather than the square brackets that are used in Pascal.

The compiler noted that a was not defined as a procedure (delimited by parentheses in Pascal). Since you cannot assign values to procedure calls, pc diagnosed a malformed statement at the point of assignment.

```
hostname% pc synerr2.p
Mon Feb 13 11:02:04 1995 synerr2.p:
           3  integer a(10)
e 18480-------^---  Inserted '['
E 18490-----------------^---  Expected identifier
           6      read(b);
E 18420----------------^---  Undefined variable
           7      for c := 1 to 10 do
E 18420---------------^---  Undefined variable
           8          a(c) := b * c
E 18420---------------^---  Undefined procedure
E 15010 line 1 -  File output listed in program statement but not
declared
In program synerr2_example:
E 18240 a undefined on line 8
E 18240 b undefined on lines 6 8
E 18240 c undefined on lines 7 8
```

Expected and Unexpected End-of-file

If `pc` finds a complete program, but there is more (noncomment) text in the input file, then it indicates that an end-of-file is expected. This situation may occur after a bracketing error, or if too many `end`s are present in the input. The message may appear after the recovery says that it `Expected '.'` because a period (`.`) is the symbol that terminates a program.

If severe errors in the input prohibit further processing, `pc` may produce a diagnostic message followed by `QUIT`. Examples include unterminated comments and lines longer than 1,024 characters.

The Pascal program, `mism.p`

```
program mismatch_example(output);

begin
    writeln('***');
    { The next line is the last line in the file. }
    writeln
```

When you compile `mism.p`, the end-of-file is reached before an end delimiter

```
hostname% pc mism.p
E 26020-------^---  Malformed declaration
  15130-------^---  Unexpected end-of-file - QUIT
```

Compiler Semantic Errors

The following sections explain the typical formats and terminology used in Pascal error messages.

Format of the Error Diagnostics

In the example program above, the error diagnostics from the Pascal compiler include the line number in the text of the program, as well as the text of the error message. While this number is most often the line where the error occurred, it can refer to the line number containing a bracketing keyword like `end` or `until`. If so, the diagnostic may refer to the previous statement. This diagnostic occurs because of the method the compiler uses for sampling line numbers. The absence of a trailing semicolon (`;`) in the previous statement causes the line number corresponding to the `end` or `until` to become associated with the statement.

As Pascal is a free-format language, the line number associations can only be approximate and may seem arbitrary in some cases.

Incompatible Types

Since Pascal is a strongly-typed language, many type errors can occur, which are called type clashes by the compiler.

The Pascal compiler distinguishes among the following type classes in its diagnostics:

`array`	`integer`	`scalar`
`boolean`	`pointer`	`string`
`char`	`real`	`varying`
`file`	`record`	

Thus, if you try to assign an `integer` value to a `char` variable, you receive a diagnostic as follows:

```
Mon Feb 13 13:16:20 1995 inchar.p:
E 25190 line 6 -  Type clash: integer is incompatible with char
  ... 25560: Type of expression clashed with type of variable in assignment
```

In this case, one error produces a two-line error message. If the same error occurs more than once, the same explanatory diagnostic is given each time.

The `scalar` *Class*

The only class whose meaning is not self-explanatory is `scalar`. It has a precise meaning in the Pascal standard where it refers to `char`, `integer`, and `boolean` types as well as the enumerated types. For the purposes of the Pascal compiler, `scalar` in an error message refers to a user-defined enumerated type, such as `color` in:

```
type color = (red, green, blue)
```

For integers, the more precise denotation `integer` is used.

Procedure and Function Type Errors

For built-in procedures and functions, two kinds of errors may occur. If a routine is called with the wrong number of arguments, a message similar to the following is displayed:

```
Mon Feb 13 13:21:26 1995 sin.p:
E 10010 line 6 -  Builtin function SIN takes exactly 1 argument
```

If the type of an argument is wrong, you receive a message similar to the following:

```
Mon Feb 13 13:31:14 1995 abs.p:
E 10180 line 8 -  Argument to ABS must be of type integer or real, not char
```

Scalar Error Messages

Error messages stating that scalar (user-defined) types cannot be read from and written to files are often difficult to interpret. In fact, if you define:

```
type color = (red, green, blue)
```

standard Pascal does not associate these constants with the strings `red`, `green`, and `blue` in any way. Pascal adds an extension so that enumerated types can be read and written; however, if the program is to be portable, you must write your own routines to perform these functions.

Standard Pascal only allows the reading of characters, integers, and `real` numbers from text files, including `input` (not strings or `booleans`). You can make the following declaration:

```
file of color
```

However, the representation is binary rather than as strings, and it is impossible to define `input` as other than a text file.

Expression Diagnostics

The diagnostics for semantically ill-formed expressions are explicit, as the following program shows. This program, `expr.p`, is admittedly far-fetched, but illustrates that the error messages are clear enough so you can easily determine the problem in the expressions.

```
program expr_example(output);

var
    a: set of char;
    b: Boolean;
    c: (red, green, blue);
    p: ^ integer;
    A: alfa;
    B: packed array [1..5] of char;

begin
    b := true;
    c := red;
    new(p);
    a := [];
    A := 'Hello, yellow';
    b := a and b;
    a := a * 3;
    if input < 2 then writeln('boo');
    if p <= 2 then writeln('sure nuff');
    if A = B then writeln('same');
    if c = true then writeln('hue''s and color''s')
end. { expr_example }
```

This program generates the following error messages:

```
hostname% pc expr.p
Mon Feb 13 13:36:51 1995 expr.p:
E 13050 line 16 -  Constant string too long
E 20070 line 17 -  Left operand of and must be Boolean, not set
E 25550 line 18 -  expression has invalid type
E 20030 line 18 -  Cannot mix sets with integers and reals as operands of *
E 20240 line 19 -  files may not participate in comparisons
E 20230 line 20 -  pointers and integers cannot be compared - operator was <=
E 20220 line 21 -  Strings not same length in = comparison
E 20230 line 22 -  scalars and Booleans cannot be compared - operator was =
```

Type Equivalence

The Pascal compiler produces several diagnostics that generate the following message:

```
non-equivalent  types
```

In general, Pascal considers types to be the same only if they derive from the same type identifier. Therefore, the following two variables have different types, even though the types look the same and have the same characteristics.

```
x : record
       a: integer;
       b: char;
    end;
y :  record
       a: integer;
       b: char;
    end;
```

The assignment:

```
x := y
```

produces the following diagnostic messages:

```
Mon Feb 13 14:22:46 1995 inchar.p:
E 25170 line 12 -  Type clash: non-identical record types
   ... 25560: Type of expression clashed with type of variable in assignment
```

To make the assignment statement work, you must declare a type and use it to declare the variables, as follows:

```
type
    r = record
        a: integer;
        b: char;
    end;
var
    x: r;
    y: r;
```

Alternatively, you could use the declaration:

```
x, y : record
        a: integer;
        b: char;
    end;
```

The assignment statement then works.

Unreachable Statements

Pascal flags unreachable statements. Such statements usually correspond to errors in the program logic, as shown in the following example:

The Pascal program, `unreached.p`

```
program unreached_example(output);

label
    1;

begin
    goto 1;
    writeln('Unreachable.');
    1:
    writeln('Reached this.');
end. { unreached_example }
```

This error message is generated when you compile `unreached.p`.

```
hostname% pc unreached.p
Tue Feb 14 14:21:03 1995 unreached.p:
w 18630 line 8 -  Unreachable statement
```

A statement is considered to be reachable if there is a potential path of control, even if it cannot be taken. Thus, no diagnostic is produced for the statement:

```
if false then
writeln('Impossible!')
```

The `goto` Statement

Pascal detects and produces an error message about `goto` statements that transfer control into structured statements—for example, `for` and `while`. It does not allow such jumps, nor does it allow branching from the `then` part of an `if` statement into the `else` part. Such checks are made only within the body of a single procedure or function.

Uninitialized Variables

Pascal does not necessarily set variables to an initial value unless you explicitly request that with the `–Z` option. The exception is `static` variables, which are guaranteed to be initialized with zero values.

Because variable use is not tracked across separate compilation units, `pc` does nothing about uninitialized or unused variables in global scope, that is, in the main program. However, `pc` checks variables with local scope—those declared in procedures and functions, to make sure they are initialized before being used. `pc` flags uninitialized variables with a warning message.

Unused Variables, Constants, Types, Labels, and Routines

If you declare a variable, constant, type, procedure, or function in local scope but never use it, Pascal gives you a warning message. It does not do this for items declared in global scope because you can use the items in a separately compiled unit.

If you declare a label but never use it, Pascal gives you a warning. This is true even for a label declared in global scope.

Compiler Panics, I/O Errors

One class of error that rarely occurs, but which causes termination of all processing when it does, is a panic.

A panic indicates a compiler-detected internal inconsistency. A typical panic message follows:

```
pc0 internal error line=110 yyline=109
```

If you receive such a message, compilation is terminated. Save a copy of your program, then contact Sun Customer Support. If you were making changes to an existing program when the problem occurred, you may be able to work around the problem by determining which change caused the internal error and making a different change or error correction to your program.

The only other error that aborts compilation when no source errors are detected is running out of memory. Most tables in Pascal, with the exception of the parse stack, are dynamically allocated, and can grow to take up a good deal of process space. In general, you can get around this problem with large programs by using the separate compilation facility. See Chapter 5, "Separate Compilation," for details.

If you receive an `out of space` message during translation of a large procedure or function, or one containing a large number of string constants, either break the procedure or function into smaller pieces or increase the maximum data segment size using the `limit` command of `csh`(1).

Runtime Errors

When Pascal detects an error in a program during runtime, it prints an error message and aborts the program, producing a core image.

Following is a list of runtime errors that Pascal generates:

<*filename*> `: Attempt to read from a file open for writing`

<*filename*> `: Attempt to write to a file open for reading`

<*filename*> : `Bad data found on enumerated read`

<*filename*> : `Bad data found on integer read`

<*filename*> : `Bad data found on real read`

<*filename*> : `Bad data found on string read`

<*filename*> : `Bad data found on varying of char read`

<*filename*> : `Cannot close file`

<*filename*> : `Could not create file`

<*filename*> : `Could not open file`

<*filename*> : `Could not remove file`

<*filename*> : `Could not reset file`

<*filename*> : `Could not seek file`

<*filename*> : `Could not write to file`

<*filename*> : `File name too long (maximum of` <*number*> `exceeded`

<*filename*> : `File table overflow (maximum of` <*number*> `exceeded)`

<*filename*> : `Line limit exceeded`

<*filename*> : `Non-positive format widths are non-standard`

<*filename*> : `Overflow on integer read`

<*filename*> : `Tried to read past end of file`

<*filename*> : `eoln is undefined when eof is true`

`Argument` <*number*> `is out of the domain of atan`

`Argument to argv of` <*number*> `is out of range`

`Assertion #`<*number*>` failed:` <*assertion message*>

`Cannot close null file`

`Cannot compute cos(`<*number*>`)`

`Cannot compute sin(`<*number*>`)`

`Cannot open null file`

`Enumerated type value of` <*number*> `is out of range on output`

`Floating point division by zero`

`Floating point operand is signaling Not-A-Number`

`Floating point overflow`

`Floating point underflow`

`Illegal argument of` <*number*> `to trunc`

`Inexact floating point result`

`Integer divide by zero`

`Integer overflow`

`Internal error in enumerated type read`

`Invalid floating point operand`

`Label of` <*number*> `not found in case`

`Negative argument of` <*number*> `to sqrt`

`Non-positive argument of` <*number*> `to ln`

`Overflow/Subrange/Array Subscript Error`

`Pointer value (`*<number>*`) out of legal range`

`Ran out of memory`

`Range lower bound of `*<number>*` out of set bounds`

`Range upper bound of `*<number>*` out of set bounds`

`Reference to an inactive file`

`Second operand to MOD (`*<number>*`) must be greater than zero`

`Statement count limit of `*<number>*` exceeded`

`Subrange or array subscript is out of range`

`Sun FPA not enabled`

`Unknown SIGFPE code`

`Unordered floating point comparison`

`Value of `*<number>*` out of set bounds`

`Varying length string index `*<number>*` is out of range`

`exp(`*<number>*`) yields a result that is out of the range of reals`

`i = `*<number>*`: Bad i to pack(a,i,z)`

`i = `*<number>*`: Bad i to unpack(z,a,i)`

`pack(a,i,z): z has `*<number>*` more elements than a`

`substring outside of bounds of string`

`unpack(z,a,i): z has `*<number>*` more elements than a`

Math Libraries 10

This chapter describes how to use the `libm` and `libsunmath` functions in Pascal programs. The math libraries are always accessible from a Pascal program because the Pascal compiler driver `pc` calls `ld`, the linker and loader, with the `-lsunmath -lm` options. If you compile Pascal program with the `-v` option, it prints the command-line used to call each of the compilation passes.

For convenience, Pascal supports special Pascal header files, `math_p.h` and `sunmath_p.h`, which contain prototypes of math functions, procedures, and some useful constants. The `math_p.h` file refers to `sunmath_p.h` by an `#include` directive.

This chapter contains the following sections:

10

Contents of the Math Libraries

Altogether, there are three math libraries:

- `libm.a`—A set of functions required by the various standards to which the operating system conforms
- `libm.so`—The shared version of `libm.a`
- `libsunmath.a`—A set of functions not required by the standards, but are of common use

Table 10-1 lists the contents of the math libraries.

Table 10-1 Contents of Math Libraries

Algebraic functions			
	Roots^{m+}		
	Euclidean distance$^{m+, s}$		
Transcendental functions			
	Elementary transcendental functions		
		Trigonometric functions	
			Trigonometric functions of radian arguments^{m+}
			Trigonometric functions of degree arguments ^{s}
			Trigonometric functions (scaled in Pi)^{s}
			Trigonometric functions (with double precision Pi) ^{s}
		Hyperbolic functions^{m+}	
		Exponential, logarithm, power$^{m+, s}$	
		Financial functions^{s}	
	Higher transcendental functions		
		Bessel^{m+}	
		Gamma^{m+}	
		Error function^{m+}	
Integral rounding functions$^{m+, s}$			

Table 10-1 Contents of Math Libraries *(Continued)*

Random number generators
Additive pseudo-random generators^{s}
Linear pseudo-random generators^{s}
Random number shufflers^{s}
IEEE support functions
IEEE functions^{m+}
IEEE test^{m+}
IEEE values^{s}
IEEE sun^{s}
Control flags^{s}
Floating-point trap handling
IEEE handling^{s}
Handling for specific SIGFPE codes (in `libc`)
Error handling function^{m}
Data conversion^{s}
BSD miscellaneous^{s}
Base conversion routines (in `libc`)
FORTRAN intrinsic functions^{s}

Legend:

^{m}	Functions available in bundled `libm`
$^{m+}$	Functions available in bundled `libm` and as single-precision version only in `libsunmath`
^{s}	Functions available in unbundled `libm` (`libsunmath`)

`libm` *Functions*

Most numerical functions are available in double- and single-precision version. In general, the names of the single-precision version are formed by adding `f` to the names of the double-precision version.

The following Pascal program is an example of how to use math functions.

```
program TestLibm(output);
#include <math_p.h>

var
  d0,d1,d2: double;
  f0,f1,f2: single;

begin
  d0 := 0.0; d1 := 1.0; d2 := 2.0;
  f0 := 0.0; f1 := 1.0; f2 := 2.0;

  writeln('Trigonometric functions');

  writeln(sin(d0));
  writeln(sinf(f0));

  sincos(M_PI_2, d1, d2);
  writeln(d1, d2);
  sincosf(M_PI_2, f1, f2);
  writeln(f1, f2);

  writeln('Exponential, logarithm, power');

  writeln(exp(d1));
  writeln(log(d1));
  writeln(pow(d1, d1));
  writeln(expf(f1));
  writeln(logf(f1));
  writeln(powf(f1, f1));
end.
```

IEEE Support Functions

This section describes the IEEE support functions, including `ieee_functions()`, `ieee_values()`, and `ieeee_retrospective()`.

ieee_functions()

The functions described in `ieee_functions`(3M) provide capabilities either required by the IEEE standard or recommended in its appendix. Example:

```
program TestIEEEFunctions(output);

#include "math_p.h"

var
  d1: double := 1.0;
  d2: double := 2.0;
  i1: integer := 1;

begin
  writeln('IEEE functions');

  writeln(ilogb(d1));
  writeln(isnan(d1));
  writeln(copysign(d1, d2));
  writeln(fabs(d1));
  writeln(fmod(d1, d1));
  writeln(nextafter(d1, d1));
  writeln(remainder(d1, d1));
  writeln(scalbn(d1, i1));
end.
```

ieee_values()

IEEE values, such as infinity, NaN, minimum and maximum positive floating-point numbers, are provided by special functions described in the `ieee_values`(3M) man page. Another example follows.

```
program TestIEEEValues(output);

#include "math_p.h"

var
  l0: integer32 := 0;
begin
  writeln('IEEE values');

  writeln(infinity);
  writeln(signaling_nan(l0));
  writeln(quiet_nan(l0));
  writeln(max_normal);
  writeln(max_subnormal);
  writeln(min_normal);
  writeln(min_subnormal);
end.
```

`ieee_retrospective()`

The `libm` function `ieee_retrospective()` prints to `stderr` information about unrequited exceptions and nonstandard IEEE modes. Pascal programs call `ieee_retrospective()` on exit by default.

SPARC Libraries

The `libm` and `libsunmath` libraries also contain:

- Argument reduction functions, using infinitely precise Pi and trigonometric functions scaled in Pi
- Data conversion routines for converting floating-point data between IEEE and non-IEEE formats
- Random number generators

There are two facilities for generating uniform pseudo-random numbers, `addrans`(3M) and `lcrans`(3M). `addrans` is an additive random number generator; `lcrans` is a linear congruential random number generator. In addition, `shufrans`(3M) shuffles a set of pseudo-random numbers to provide even more randomness for applications that need it.

```
program TestRandom(output);

#include "math_p.h"

var
  n: integer := 100;
  i: integer;
  ilb,           { Lower bound }
  iub: integer; { Upper bound }
  ia: array [1..100] of integer;

begin
 writeln('Integer linear congruential random number generator');
  ilb := I_LCRAN_LB;
  iub := I_LCRAN_UB;
  i_lcrans_(ia, n, ilb, iub);
  for i := 1 to n do
    writeln(ia[i]);

  writeln('Integer additive random number generator');
  ilb := minint;
  iub := maxint;
  i_addrans_(ia, n, ilb, iub);
  for i := 1 to n do
    writeln(ia[i]);

  writeln('Integer random number shufflers');
  i_shufrans_(ia, n, ilb, iub);
  for i := 1 to n do
    writeln(ia[i]);
end.
```

Arithmetic Exceptions

An arithmetic exception arises when an attempted atomic arithmetic operation does not produce an acceptable result. The meaning of the terms “atomic” and “acceptable” may vary, depending on the context.

Following are the five types of IEEE floating-point exceptions:

- **Invalid operation**—An operand is invalid for the operation about to be performed.

- **Division by zero**—The divisor is zero, and the dividend is a finite non-zero number; or, more generally, an exact infinite result is delivered by an operation on finite operands.
- **Overflow**—The correctly rounded result is larger than the largest number in the required precision.
- **Underflow**—The number is too small, or precision is lost, and no signal handler is established for underflow.
- **Inexact**—The rounded result of a valid operation is different from the infinitely precise result. This exception occurs whenever there is untrapped overflow or untrapped underflow.

Math Library Exception-Handling Function: `matherr()`

Some `libm` functions are specified to call `matherr()` when an exception is detected. You can redefine `matherr()` by including a function named `matherr()` in the program. When an exception occurs, a pointer to the exception structure, `exc`, is passed to the user-supplied `matherr()` function. This structure, defined in the `math_p.h` header file, is as follows:

```
type
  exception = record
    kind: integer;
    name: ^string;
    arg1: double;
    arg2: double;
    retval: double;
  end;
```

The element kind is an integer constant that describes the type of exception that occurred, and is one of the following constants. These constants are defined in the header file.

`DOMAIN`	Argument domain exception
`SING`	Argument singularity
`OVERFLOW`	Overflow range exception
`UNDERFLOW`	Underflow range exception
`TLOSS`	Total loss of significance
`PLOSS`	Partial loss of significance

If your `matherr()` function returns a non-zero result, no exception message is printed, and `errno` is not set.

```
program TestMatherr(output);

#include <math_p.h>

function matherr(var info: exception): integer;
begin
  case info.kind of
    DOMAIN: begin
      { change sqrt to return sqrt(-arg1), not NaN }
     if substr(info.name^, 1, length('sqrt')) = 'sqrt' then begin
        info.retval := sqrt(-info.arg1);
        matherr := 1; { No exception message will be printed }
      end;
    end;
    otherwise
      matherr := 0;
  end;
end;

begin
  writeln('Error handling function');
  writeln('sqrt(-1)= ', sqrt(-1));
end.
```

libsunmath Support for IEEE Modes and Exceptions

`ieee_handler()` is used primarily to establish a signal handler for a particular floating-point exception or group of exceptions.

The syntax of this function is described in the `ieee_handler`(3M) man page.

This following Pascal program demonstrates how to abort on division by zero.

```
program TestIEEEHandler(output);

#include <math_p.h>

procedure DivisionHandler(
  sig: integer;
  sip: univ_ptr;
  uap: univ_ptr);
begin
  writeln('Bad data - division by zero.');
end; { DivisionHandler }

var
  FpAction, FpException: string;
  Zero: integer := 0;

begin
  FpAction := 'set';
  FpException := 'division';

  writeln(ieee_handler(FpAction, FpException,
    addr(DivisionHandler)));
  writeln('1/0 = ', 1 / Zero);

  writeln(ieee_handler(FpAction, FpException, SIGFPE_DEFAULT));
  writeln('1/0 = ', 1 / Zero);
end.
```

`ieee_flags()` is the recommended interface to:

- Query or set rounding direction mode
- Query or set rounding precision mode
- Examine, clear, or set accrued exception flags

The syntax of this function is described in the `ieee_flags`(3M) man page.

If an exception is raised at any time during program execution, then its flag is set, unless it is explicitly cleared. Clearing accrued exceptions is done by a call, as shown in the following Pascal program.

```
program TestIEEEFlags(output);

#include "math_p.h"

var
  FlAction, FlMode, FlIn: string;
  FlOut: string_pointer;
  Zero: integer := 0;

begin
  writeln(sqr(-1));     { Invalid operation }
  writeln(1 / Zero);    { Division by zero }
  writeln(exp(709.8));  { Overflow }
  writeln(exp(-708.5)); { Underflow }
  writeln(log(1.1));    { Inexact }

  FlAction := 'clear';
  FlMode := 'exception';
  FlIn := 'all';
  writeln(ieee_flags(FlAction, FlMode, FlIn, FlOut));
end.
```

The Multithread Library 11

This chapter describes how to use the multithread library, `libthread`, in Pascal programs. This multithread library interface is defined in the *SunOS 5.x Reference Manual*. The *SunOS 5.x Guide to Multi-Thread Programming* describes concurrent programming concepts and practices suitable to Solaris threads.

The features described in this chapter apply only to the Solaris 2.x environment.

This chapter contains the following sections:

Multithread Environment for Pascal

Compiling and binding a multithreaded Pascal program requires the following:

- Pascal 4.2 compiler (`pc`)
- Standard Solaris linker (`ld`)
- Multithread library (`libthread`)
- Multithreading-safe libraries (`libc`, `libpc`, possibly others as needed)

- Pascal program including the two `#include <file>` statements for `thread_p.h` and for `synch_p.h`, part of Pascal 4.2 providing `libthread` library binding for Pascal (as `math_p.h` does for the mathematical library)

Compiling Multithreaded Programs

Compile multithreaded programs by using the `-mt` flag to ensure that the `-D_REENTRANT` option is passed to `cpp` and that the `-lpc_mt` and `-lthread` options are passed to `ld`.

Introduction to Multithreading

A thread is a sequence of execution steps performed by a program. A program without multithreading operates on a single thread of control. Control in a single-threaded program is always synchronous, operating sometimes from the program and sometimes from the operating system. Figure 11-1 schematically depicts the basic elements of multithreading as discussed in this chapter.

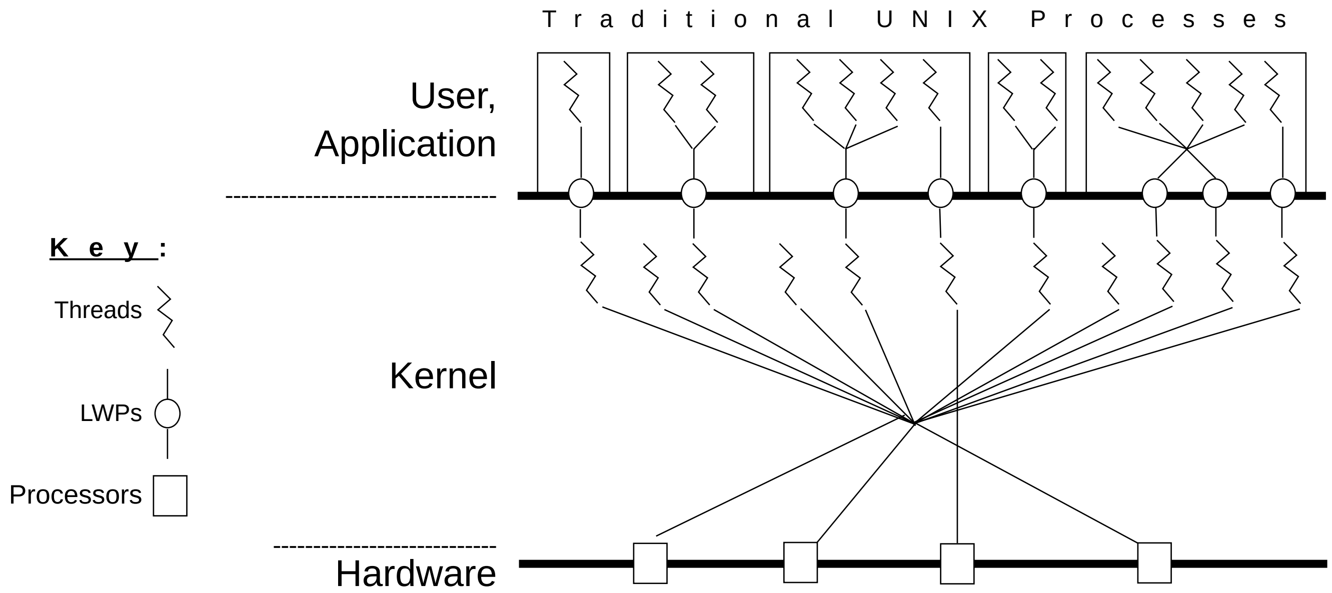


Figure 11-1 Thread Interface Architecture

The multithreading capabilities of Solaris threads allow many threads of control to share a single UNIX process and use its address space, as shown in Figure 11-1. A multithreaded UNIX process is not a single thread of control in itself, but it contains one or more threads of control. A single thread can start other threads, and each thread can execute independently and asynchronously. Multithreading can:

- Take advantage of multiprocessing hardware
- Enhance application responsiveness
- Enhance application throughput

Thread Resources

A thread can be created within an existing process a thousand times faster than a new process can be created. Switching between threads within a process is especially fast because it does not involve switching between address spaces. Each thread includes the following resources:

- thread identifier
- registers (set of registers)
- signal mask
- execution priority
- stack
- thread-specific data

Thread Creation Functions

Solaris threads provides the following thread creation functions:

- `thr_create`(3T), which creates and starts a new thread that executes the function specified in the call.
- `thr_join`(3T), which blocks until the specified thread exits.
- `thr_self`(3T), which returns the thread identifier structure of the caller.
- `thr_exit`(3T), which terminates the invoking thread and sets the exit status to the specified value.

Lightweight Processes

The multithread library uses underlying threads of control called *lightweight processes* (LWPs) that are supported by the kernel. LWPs work like virtual

CPUs that execute code or system calls. LWPs act as bridges between the user or application level and the kernel. Each traditional UNIX process contains one or more LWPs, each running one or more user threads. Programming thread creation involves creating a user context, usually without also creating an LWP.

Process Control

Your program can tell the `libthread` multithread support subroutine library how many threads should be able to run at the same time. You should make sure at least that number of LWPs is available.

The `libthread` multithread library schedules LWP usage for user threads. When a user thread blocks due to synchronization, the LWP transfers to another runnable thread through co-routine linkage and not by system call. If all LWPs block, the multithread library makes another LWP available.

Each LWP is independently dispatched by the kernel, performs independent system calls, and may incur independent page faults. On multiprocessor systems, LWPs can run in parallel, one at a time per processor. The operating system multiplexes the LWPs onto the available processors, deciding which LWP will run on which processor and when. The kernel schedules CPU resources for the LWPs according to their scheduling classes and priorities; the kernel has no information about the user threads active behind each process.

Synchronization

The operating system schedules LWP usage of the processors. Threads scheduling is influenced by the `thr_setconcurrency` and `thr_setprio` multithread library functions; see their man pages for more information.

Threads share access to the process address space, and therefore their accesses to shared data must be synchronized. Solaris threads provide a variety of synchronization facilities that use various semantics and support different styles of synchronization interaction:

- Mutual exclusion locks
- Condition variables
- Counting semaphores
- Multiple-readers / single-writer locks

Mutual exclusion locks let one thread at a time hold the lock. They are typically used to ensure that only one thread at a time executes a section of code (called a *critical section*) that accesses or modifies some shared data. Use mutex locks to limit access to a resource to one thread at a time. Refer to the `mutex`(3T) man pages for more details.

The following two Solaris threads routines are the most commonly used routines for mutual exclusion:

- `mutex_lock`(3T) acquires the lock or blocks the calling thread if the lock is already held. Blocked threads wait on a prioritized queue.
- `mutex_unlock`(3T) unlocks the mutex lock. This routine must be called by the thread that holds the mutex lock, which must be locked. If other threads are waiting for the lock, the thread at the head of the queue is unblocked.

Conditional Variables

Use conditional variables to make threads wait until a particular condition is true. A conditional variable must be used in conjunction with a mutex lock. Refer to the `condition`(3T) man page for more details.

The following three Solaris threads routines are the most common routines for handling conditional variables:

- `cond_wait`(3T) blocks until the condition is signaled. It atomically releases the associated mutex lock before blocking and atomically reacquires the lock before returning. In typical use, a conditional expression is evaluated under the protection of a mutex lock. If the conditional expression is false, the thread blocks on the conditional variable. When another thread changes the condition, it signals the conditional variable. One (or all) of the threads waiting on the condition then unblock and try to reacquire the mutex lock. Since reacquiring the mutex lock can be blocked by other waiting threads, the condition that caused the wait must be retested after the mutex lock has been acquired.
- `cond_signal`(3T) signals one thread blocked in `cond_wait`.
- `cond_broadcast`(3T) wakes all threads blocked in `cond_wait`.

Semaphores

Solaris threads provide conventional counting semaphores. A semaphore is a non-negative integer count that can be atomically incremented and decremented by special routines. Semaphores must be initialized before use. Refer to the `semaphore`(3T) man page for more details.

The following three Solaris thread routines are the most common routines for handling semaphores:

`sema_init`(3T) initializes the semaphore variable.

`sema_wait`(3T) blocks the thread until the semaphore becomes greater than zero, then decrements it—the P operation on Dijkstra semaphores.

`sema_post`(3T) increments the semaphore, potentially unblocking a waiting thread—the V operation on Dijkstra semaphores.

Readers/Writer Lock

A multiple-readers/single-writer lock gives threads simultaneous read-only access to a protected object. such a lock also gives write access to the object to a single thread while excluding any readers. This type of lock is usually used to protect data that is read more often than written. Refer to the `rwlock`(3T) man pages for more details.

The following routines are the most commonly used for readers/writer locks:

- `rwlock_init`(3T) initializes the readers/writer lock and sets its state to unlocked.
- `rw_rdlock`(3T) acquires the read lock and blocks if a writer holds the lock.
- `rw_wrlock`(3T) acquires the write lock and blocks if a writer or any readers holds the lock.
- `rw_unlock`(3T) unlocks the readers/writer lock or returns an error.

Parallel Matrix-Multiplication Example

Computationally intensive applications can benefit from using all available processors. Matrix multiplication, for example, is an operation that could be speeded up by using multiple processors, as shown in the following program, `MatrixMultiply`.

When a matrix-multiply operation is called, it acquires a mutex lock to ensure that only one matrix-multiply operation is in progress. The MatrixMultiply program uses mutex locks that are statically initialized to zero. A requesting thread checks whether its worker threads have been created. If its worker threads have not been created, the requesting thread creates them.

In the MatrixMultiply program, once its worker threads are created, the requesting thread sets up a to_do counter for the work and then signals the worker procedure via a conditional variable. Each worker procedure picks off a row and column from the input matrix, then the next worker procedure gets the next item, and so on.

The matrix-multiply operation then releases the mutex lock so computation of the vector product can proceed in parallel, with each processor running one thread at a time.

When the vector product results are ready, the worker procedure reacquires the mutex lock and updates the not_done counter of work not yet completed. At the end of the matrix-multiply operation, the worker procedure that completes the last bit of work then signals the requesting thread. Each iteration computes the result of one entry in the result matrix. In some cases, the amount of computation could be insufficient to justify the overhead of synchronizing multiple worker procedures. In such cases, more work per synchronization should be given to each worker. For example, each worker could compute an entire row of the output matrix before synchronization.

The Pascal program, MatrixMultiply.p (including the next 3 pages)

```
program MatrixMultiply:
#include <thread_p.h>
#include <synch_p.h>
const
    THR = 2;{ level of parallelism }
    DIM = 400;{ array dimension }
type
    arr_elem_t = double;

    arr_t = array[0..DIM-1, 0..DIM-1] of arr_elem_t;
    arr_p = ^arr_t;

{continued on next page}
```

```
{continued from previous page}
    { totality data for parallel work }
    work_data_t = record
        { synchronization primitives }
            lock: mutex_t;
        start_cond, done_cond: cond_t;
        { counters of work }
            to_do, not_done: integer;
        { level of parallelism }
            workers: integer;
        { shared data }
            m1: arr_p;
            m2: arr_p;
            m3: arr_p;
            row, col: integer;
    end;
var
    only_one_matrix_multiply_is_in_progress: mutex_t;
    work_data: work_data_t;
    m1: arr_t;
    m2: arr_t;
    m3: arr_t;
    i, j: integer;
    x: integer;
    elems_sum: arr_elem_t;
    start, stop: integer;
procedure worker; forward;
procedure matmul(m1, m2, m3: arr_p); { requesting thread }
var
    i: integer;
    cr: integer;
begin
cr := mutex_lock(addr(only_one_matrix_multiply_is_in_progress));
    cr := mutex_lock(addr(work_data.lock));
    { create worker threads }
    if (work_data.workers = 0) then begin
        for i := 0 to THR - 1 do
            cr := thr_create(nil, 0, addr(worker), nil,
                THR_NEW_LWP, nil);
        work_data.workers := THR;
    end;

{continued on next page}
```

```
{continued from previous page}

    { initialization data for parallel work }
    work_data.m1 := m1;
    work_data.m2 := m2;
    work_data.m3 := m3;
    work_data.row := 0;
    work_data.col := 0;
    work_data.to_do := DIM*DIM;
    work_data.not_done := DIM*DIM;
    { signals the worker to start via a condition variable }
    cr := cond_broadcast(addr(work_data.start_cond));

    { waiting for signal from worker that completes
      the latest bit of work }
    while (work_data.not_done > 0) do
        cr := cond_wait(addr(work_data.done_cond),
            addr(work_data.lock));
    cr := mutex_unlock(addr(work_data.lock));
    cr :=
mutex_unlock(addr(only_one_matrix_multiply_is_in_progress));
end;
procedure worker;
var
    wm1, wm2, wm3: arr_p;
    row, col: integer;
    i: integer;
    result: arr_elem_t;
    cr: integer;
begin
    while true do begin
        { critical region 1 }
        cr := mutex_lock(addr(work_data.lock));
        while work_data.to_do = 0 do { wait for signal to start }
            cr := cond_wait(addr(work_data.start_cond),
                addr(work_data.lock));
        work_data.to_do := work_data.to_do - 1;
        wm1 := work_data.m1;
        wm2 := work_data.m2;
        wm3 := work_data.m3;
        row := work_data.row;

{continued on next page}
```

```
{concluding from previous page}

        col := work_data.col;
        work_data.col := work_data.col + 1;
        if work_data.col = DIM then begin
            work_data.col := 0;
            work_data.row := work_data.row + 1;
        if work_data.row = DIM then
            work_data.row := 0;
        end;
        cr := mutex_unlock(addr(work_data.lock));
        { end of critical region 1 }
        { computing the vector product in parallel }
        result := 0;
        for i := 0 to DIM - 1 do
            result := result + wm1^[row,i] * wm2^[i,col];
        wm3^[row,col] := result;
        { end of computing the vector product in parallel }
        { critical region 2 }
        cr := mutex_lock(addr(work_data.lock));
        work_data.not_done := work_data.not_done - 1;
        if work_data.not_done = 0 then {work is complete}
            cr := cond_signal(addr(work_data.done_cond));
            cr := mutex_unlock(addr(work_data.lock));
        { end of critical region 2 }
    end;
end;
begin
    writeln('Matrix size: ', DIM :1);
    writeln('Number of worker threads: ', THR :1);
    for i := 0 to DIM - 1 do
        for j := 0 to DIM - 1 do begin
            m1[i,j] := random(x);
            m2[i,j] := random(x);
        end;
    start := wallclock;
    matmul(addr(m1), addr(m2), addr(m3));
    stop := wallclock;
    writeln('Matrix multiplication time: ', stop - start :1, '
seconds.');
end.
```

Improving Time Efficiency With Two Threads

To save time, the preceding program could be run with a different number of threads and matrices of different sizes. The following examples show the results of testing two different thread / matrix combinations on a SPARCstation 10 with two 50MHz TMS390Z55 CPUs.

Results of running `MatrixMultiply.p` with a single thread:

```
> matr_mult_1
Matrix size: 400
Number of worker threads: 1
Matrix multiplication time: 68 seconds.
```

Using two threads to run `MatrixMultiply.p` cuts the time for the matrix multiplication almost in half:

```
> matr_mult_2
Matrix size: 400
Number of worker threads: 2
Matrix multiplication time: 35 seconds.
```

Use of Many Threads

The following example Pascal program, `many_threads.p`, is based on a similar C example in the *Threads Primer (A Guide to Multithreaded Programming)* by Bill Lewis and Daniel J. Berg. This example shows how to easily create many threads of execution in a Solaris environment.

Because of the lightweight nature of threads, it is possible to create thousands of threads. After its creation, each thread is blocked by waiting on a mutex variable. (This prevents the thread from continuing execution independently.) After the main thread has created all other threads, it waits for user input and then tries to join all the threads.

Pascal program, many_threads.p

```
program many_threads;
#include <thread_p.h>
#include <synch_p.h>
const
    THR_COUNT = 100;{ the number of threads }
var
    lock: mutex_t;
    cr: integer;
    i: integer;
procedure thr_sub;
var
    thread_id: thread_t;
begin
    { try to lock the mutex variable - since the main thread
    has locked the mutex before the threads were created, this
    thread will block until the main thread unlock the mutex }
    cr := mutex_lock(addr(lock));
    thread_id := thr_self;
    writeln('Thread ', thread_id:1, ' is exiting...');
     {unlock the mutex variable, to allow another thread to
proceed}
    cr := mutex_unlock(addr(lock));
end;
begin
    writeln('Creating ', THR_COUNT:1, ' threads...');
    { lock the mutex variable - this mutex is being used to keep
    all the other threads created from proceeding }
    cr := mutex_lock(addr(lock));
    { creates all the threads }
    for i := 0 to THR_COUNT - 1 do
        cr := thr_create(nil, 2048, addr(thr_sub), nil,0, nil);
    writeln(i+1:1, ' threads have been created and are running!');
    writeln('Press <Return> to join all the threads...');
    { wait till user presses return, then join all the threads }
    readln;
    writeln('Joining ', THR_COUNT:1, ' threads...');
   {now unlock the mutex variable, to let all the threads proceed}
    cr := mutex_unlock(addr(lock));
    { join the threads }
    for i := 0 to THR_COUNT - 1 do
        cr := thr_join(0, nil, nil);
end.
```

Debugging Multithreaded Pascal Programs

Using the `dbx` utility you can debug and execute programs written in Pascal. Both `dbx` and the SPARCworks Debugger support debugging multithreaded programs. Table 11-1 lists `dbx` options that support multithreaded programs.

Table 11-1 `dbx` Options That Support Multithreaded Programs

dbx Option	**Explanation**
`cont [[at "`*prog_file*`":`*line*`] [`*sig*`] [`*id*`]]`	Continue execution of program "*prog_file*" at line number *line* with signal number *sig*. The *id*, if present, specifies which thread ID (*tid*) or LWP ID (*lid*) to continue. If *id* is absent, the default is for all *tid*s and *lid*s continue. (For more information, refer to the `dbx` command discussion of using `continue` for loop control.)
`lwp`	Display the current LWP.
`lwp` *lid*	Switch to the LWP identified lid *lid*.
`lwps`	List all LWPs in the current process.
`next...` *tid*	Step the given thread. When a function call is skipped over, all LWPs are implicitly resumed for the duration of that function call. Non-active threads cannot be stepped.
`next...` *lid*	Step the given LWP. Will not implicitly resume all LWPs when skipping a function.
`step...` *tid*	Step the given thread. When a function call is skipped over, all LWPs are implicitly resumed for the duration of that function call. Non-active threads cannot be stepped.
`step...` *lid*	Step the given LWP; will not implicitly resume all LWPs when skipping a function.
`thread`	Display current thread.
`thread` *tid*	Switch to thread *tid*. In the following variations, the lack of the optional *tid* means the current thread.
`thread -info [`*tid*`]`	Display everything known about the current [or given] thread.
`thread -locks [`*tid*`]`	Display all locks held by the current [or given] thread.
`thread -suspend [`*tid*`]`	Put the current [or given] thread into suspended state.
`thread -continue [`*tid*`]`	Unsuspend the current [or given] thread.
`thread -hide [`*tid*`]`	"Hide" the current [or given] thread; will not show in the `threads` listing.
`thread -unhide [`*tid*`]`	"Unhide" the current [or given] thread.

Table 11-1 `dbx` Options That Support Multithreaded Programs

dbx Option	Explanation
`thread -unhide all`	"Unhide" all threads.
`threads`	Display a list of all known threads.
`threads -all`	Display threads normally not printed (zombies).
`threads -mode all|filter`	Control whether `threads` by default lists all threads or filters them.
`threads -mode`	Display a list of the current mode of each thread.

Sample `dbx` *Session*

The following examples use the program `many_threads.p`.

1. **Compile - To use** `dbx` **or** `debugger`**, compile and link with** `-g` **flag, as shown in the following command line:**

```
> pc many_threads.p -o many_threads -mt -g
```

2. **Start - To start** `dbx`**, enter** `dbx` **and the name of the executable file, as shown in the following command line and screen output display:**

```
> dbx many_threads
  Reading symbolic information for many_threads
  Reading symbolic information for rtld /usr/lib/ld.so.1
  Reading symbolic information for libthread.so.1
  Reading symbolic information for libc.so.1
  Reading symbolic information for libdl.so.1
  detected a multithreaded program
```

3. **Set breakpoints - To set a breakpoint, enter a** `stop at "`*file*`":`*N* **command, where** *file* **is the program file name and** *N* **is a program string number in that program. The following two commands, for example, set two breakpoints in the** `many_threads.p` **program:**

```
> stop at "many_threads.p":46
> stop at "many_threads.p":58
```

4. **Run program - To run the executable file, enter the** `run` **command as shown in the following command line and screen output display:**

```
> run
  Running: many_threads
 (process id 12452)
 t@1 (l@1) stopped in program at line 46 in file "many_threads.p"
 46 writeln(i+1:1, ' threads have been created and are running!');
```

5. **Print threads - To print a list of all known threads, enter the** `threads` **command as shown in the following command line and screen output display:**

```
> threads
      t@1   a l@1   ?()   breakpoint               in program()
      t@2           ?()   sleep on (unknown)       in _swtch()
      t@3   b l@2   ?()   running                  in __sigwait()
      t@4           thr_sub()      runnable        in _setpsr()
      t@5           thr_sub()      runnable        in _setpsr()
...
    t@102           thr_sub()      runnable        in _setpsr()
    t@103           thr_sub()      runnable        in _setpsr()
```

6. **Continue program - To continue program execution after the** `stop at "many_threads":46` **command, enter the** `cont` **command as shown in the following command line and screen output display:**

```
> cont
  continuing all LWPs
  Creating 100 threads...
  100 threads have been created and are running!
  Press <Return> to join all the threads...
  Joining 100 threads...
 t@1 (l@1) stopped in program at line 58 in file "many_threads.p"
  58   cr := mutex_unlock(addr(lock));
```

7. **List LWPs - To list all LWPs in the current process, enter the** `lwps` **command as shown in the following command line and screen output display:**

```
> lwps
  l@1 breakpoint       in program()
  l@2 running          in __sigwait()
  l@3 running          in _lwp_sema_wait()
  l@4 running          in ___lwp_cond_wait()
```

8. **Continue program - To continue program execution after the** `stop at "many_threads":58` **command, enter the** `cont` **command as shown in the following command line and screen output display:**

```
> cont
  continuing all LWPs
  Thread 4 is exiting...
  Thread 5 is exiting...
...
  Thread 102 is exiting...
  Thread 103 is exiting...
  execution completed, exit code is 0
```

9. **Quit - Exit** `dbx`**:**

```
> quit
```

Pascal Preprocessor A

This appendix describes the preprocessors, `cpp`(1) and `cppas`.

`cpp`

`cpp`(1) is the C language preprocessor. Pascal runs your source program through `cpp`(1) when you compile it without the `–xl` option. For a complete description of `cpp`(1), see the Solaris documentation.

`cppas`

The `cppas` preprocessor handles the Pascal conditional variables and compiler directives. You call `cppas` using the `–xl` option.

Conditional Variables

A conditional variable is defined when it appears in a `%var` directive; otherwise, it is undefined. In addition, we predefine:

```
__sun                  __SVR4                  sparc
__sparc                __SUNPRO_PC=0x400       unix
__unix                 sun
```

These variables are not predefined when you use `-s0`, `-s1`, `-V0`, or `-V1`.

A defined conditional variable is enabled (`true`) when it appears in either the `%enable` directive or in the `–config` option; otherwise, it is disabled (`false`), as in:

```
%var one two
%enable two
```

The following section describes `%var` and `%enable`. Programs that contain conditional variables must be compiled with the `–xl` option.

Compiler Directives

A directive indicates some action for the compiler to take. You can use a directive anywhere in your program.

Each directive consists of a percent sign (%) followed by the directive name. Programs that contain compiler directives must be compiled with the `–xl` option.

Table A-1 summarizes the compiler directives.

Table A-1 `cppas` Compiler Directives

Compiler Directive	**Description**
`%config`	Sets a special predefined conditional variable with a value of either `true` or `false`.
`%debug`	Instructs `pc` to compile this line of code when you use the `-cond` compiler directive.
`%else`	If *expression* in `%if` *expression* `%then` is `false`, the compiler skips over the `%then` part and executes the `%else` part instead.
`%elseif`	If *expression* in `%if` *expression* `%then` is `false`, the compiler skips over the `%then` part and executes the `%elseif` part instead. Similar to `%else`.
`%elseifdef`	If *expression* in `%ifdef` *expression* `%then` is false, the compiler skips over the `%then` part and executes the `%elseifdef` part instead.
`%enable`	Sets a conditional variable to `true`.
`%endif`	Indicates the end of an `%if` or `%ifdef` directive.
`%error`	Prints a string on the standard output and treats it as an error.

Table A-1 `cppas` Compiler Directives *(Continued)*

Compiler Directive	Description
`%exit`	Stops processing the current Pascal source file.
`%if`	When the compiler encounters a `%if` *expression* `%then` directive, it evaluates *expression*. If *expression* is `true`, `pc` executes the statements after `%then`. If *expression* is `false`, `pc` skips over `%then`.
`%ifdef`	Determines whether or not you previously defined a conditional variable in a `%var` directive.
`%include`	Inserts the lines from the specified file into the input stream.
`%list`	Enables a listing of the program.
`%nolist`	Disables the program listing.
`%slibrary`	Inserts the lines from the specified file into the input stream. Same as `%include`.
`%var`	Defines conditional variables.
`%warning`	Prints a warning string on the standard output.

The rest of this appendix contains detailed descriptions and examples of each directive.

The `%config` *Directive*

The `%config` directive is a predefined conditional variable with a value of either `true` or `false`.

Syntax

`%config`

Comments

`%config` is `true` when you compile your program with the `-config` option; otherwise, `%config` is `false`.

Use `%config` in an `%if`, `%ifdef`, `%elseif`, or `%elseifdef` directive to catch any undefined values specified with `-config`. Do not define `%config` in the `%var` directive.

Example

The Pascal program, `config.p`, which defines the conditional variables one and `two`

```
program config_example(output);

{ This program demonstrates the use of the
  %config compiler directive. }

var
    a: integer := maxint;
    b: integer := minint;

%var one two

begin
    writeln('Begin program.');
    %if one %then
        writeln('One is defined as ', a:2, '.');
    %elseif two %then
        writeln('Two is defined as ', b:2, '.');
    %elseif %config %then
        writeln('Nothing is defined.');
    %endif
    writeln('End program.')
end. { config_example }
```

The output when you compile `config.p` without the `-config` option

```
hostname% pc -xl config.p
hostname% a.out
Begin program.
End program.
```

The output when you define the variable one

```
hostname% pc -xl -config one config.p
hostname% a.out
Begin program.
One is defined as 32767.
End program.
```

The output when you define `two`

```
hostname% pc -xl -config two config.p
hostname% a.out
Begin program.
Two is defined as -32768.
End program.
```

The output when you define `foo`

```
hostname% pc -xl -config foo config.p
Fri Mar 3 15:22 1995 config.p

Error: –CONFIG command argument foo was never declared.
Compilation failed
```

The `%debug` *Directive*

The `%debug` directive instructs `pc` to compile this line of code when you use the `–cond` compiler directive.

Syntax

```
%debug;
```

Comments

The `%debug` directive works in conjunction with the `–cond` compiler option. `–cond` causes `pc` to compile the lines in your program that begin with `%debug`. Without `–cond`, `pc` treats lines with `%debug` as comments.

Example

The Pascal program, debug.p

```
program debug_example(output);

{ This program demonstrates the use of the
  %debug compiler directive. }

begin
    writeln ('Hello, how are you?');
    %debug;  writeln ('Fine, thank you.');
end. { debug example }
```

The output when you compile debug.p without the –cond option

```
hostname% pc -xl debug.p
hostname% a.out
Hello, how are you?
```

The output when you use –cond

```
hostname% pc -xl –cond debug.p
hostname% a.out
Hello, how are you?
Fine, thank you.
```

The `%else` *Directive*

The `%else` directive provides an alternative action to the `%if` directive.

Syntax

```
%if expression %then
    .
    .
%else
    .
    .
%endif
```

Example

The Pascal program, `if_then_else.p`

```
program if_then_else (output);
%var red
begin
%if red %then
    writeln ('It is red.');
%else
    writeln ('It is not red.')
%endif
end.
```

The output when you compile `if_then_else.p` without the `-config`

```
hostname% pc -xl if_then_else.p
hostname% a.out
It is not red.
```

The output when you supply `-config` with the argument `red`

```
hostname% pc -xl -config red if_then_else.p
hostname% a.out
It is red.
```

The `%elseif` *Directive*

The `%elseif` directive provides an alternative action to the `%if` directive.

Syntax

`%if` *expression* `%then`

```
       .
       .
```

`%elseif` *expression* `%then`

```
       .
       .
```

`%endif`

Comments

If the expression in `%if` *expression* `%then` is `false`, `pc` skips over the `%then` part and executes the `%elseif` part instead. *expression* consists of a conditional variable and the optional `boolean` operators, `and`, `or`, and `not`. See the `%else` listing for examples of *expression*.

Example

The Pascal program, `elseif.p`

```
program elseif_example(output);

{ This program demonstrates the use of the
   %if, %then, and %elseif directives. }

%var blue red

begin
    %if blue %then
        writeln('The color is blue.');
    %elseif red %then
        writeln('The color is red.');
    %endif
end. { elseif_example }
```

The output when you supply `-config` with the argument `blue`

```
hostname% pc -xl -config blue elseif.p
hostname% a.out
The color is blue.
```

The output when you supply `-config` with the argument `red`

```
hostname% pc -xl -config red elseif.p
hostname% a.out
The color is red.
```

The `%elseifdef` *Directive*

The `%elseifdef` directive provides an alternative action to the `%ifdef` directive.

Syntax

```
%ifdef expression %then
       .
       .
%elseifdef expression %then
       .
       .
%endif
```

Comments

If the expression in `%ifdef` *expression* `%then` is `false`, `pc` skips over the `%then` part and executes the `%elseifdef` part instead. *expression* consists of a conditional variable and the optional `boolean` operators, `and`, `or`, and `not`. See the `%else` listing for examples of *expression*.

Example

The Pascal program, `ifdef.p`, which first checks if `bird1` has been defined. If not, it defines it with a `%var` directive. If `bird1` has been defined, the program checks whether or not it needs to define `bird2`.

```
program ifdef_example(output);

%include 'bird.h';

begin
    %ifdef not(bird1) %then
          %var bird1
    %elseifdef not(bird2) %then
          %var bird2
    %endif;

    %if bird1 %then
        writeln('Bird one is a ', a, '.');
    %elseif bird2 %then
        writeln('Bird two is a ', b, '.')
    %endif
end.  { ifdef_example }
```

The `include` file, `bird.h`

```
var
    a: array[1..7] of char := 'penguin';
    b: array[1..6] of char := 'toucan';

%var bird1
```

The output when you enable `bird1` with the `-config` option

```
hostname% pc -xl -config bird1 ifdef.p
hostname% a.out
Bird two is a penguin.
```

The output when you enable `bird2` with the `-config` option

```
hostname% pc -xl -config bird2 ifdef.p
hostname% a.out
Bird two is a toucan.
```

The `%enable` *Directive*

The `%enable` directive sets a conditional variable to `true`.

Syntax

`%enable` *var1 ..., varN*

Comments

A defined conditional variable is `enable` (`true`) when it appears in either the `%enable` directive or in the `-config` option. Conditional variables are `false` by default.

Example

The Pascal program, `enable.p`. This example sets the conditional variable `two` to `true`, which is equivalent to setting the `-config` option to `two` on the command-line.

```
program enable_example(output);

{ This program demonstrates the use of
   the %enable compiler directive. }

var
    a: integer;
    b: integer;

%var one, two
%enable two

begin
    %if one %then
        a := maxint;
        writeln('One is defined as ', a:2, '.');
    %endif
    %if two %then
        b := minint;
        writeln('Two is defined as ', b:2, '.');
    %endif
end. { enable_example }
```

The commands to compile and output `enable.p`

```
hostname% pc -xl enable.p
hostname% a.out
Two is defined as -32768.
```

The `%endif` *Directive*

The `%endif` directive indicates the end of a `%if` or `%ifdef` directive. See the sections on `%if` and `%ifdef` for more information on this directive.

The `%error` *Directive*

The `%error` directive causes the compiler to print a string on the standard output and treat it as an error.

Syntax

`%error '`*string*`'`

Comments

`pc` does not produce an object file.

Example

The Pascal program, `error.p`

```
program error_example(output);

{ This program demonstrates the use of the
  %error compiler directive. }

%var arch

begin
    %if arch %then
        writeln('This is a SPARC computer.');
    %else
        %error 'Unknown architecture.'
    %endif
end. { error_example }
```

`error.p` produces this error if you compile it without the `–config sparc` option.

```
hostname% pc -xl error.p
Tue Feb 28 17:10 1995 error.p

Line 12 :  %error 'Unknown architecture.'
E --------------------^---'Unknown architecture.'
Compilation failed
```

The `%exit` Directive

The `%exit` directive instructs the compiler to stop processing the current Pascal source file.

Syntax

`%exit`

Comments

If the compiler encounters an `%exit` directive within an `include` file, it stops processing the `include` file, but continues processing the source file in which it is included. In effect, `%exit` is equivalent to an end-of-file marker.

When the compiler processes an `%exit` directive within an `%if` or `%ifdef` construct, it closes all `%if` or `%ifdef`s before it stops processing the current file.

Example

The Pascal program, `exit_directive.p`

```
program exit_directive(output);

begin
    writeln('Hello, world!')
end. { exit_directive }
%exit
Everything after the %exit is ignored.
So you can put anything here.
```

The commands to compile and execute `exit_directive.p`

```
hostname% pc -xl exit_directive.p
hostname% a.out
Hello, world!
```

The `%if` Directive

The `%if` directive is a conditional branching directive.

Syntax

```
%if expression %then
       .
       .
%end if
```

Comments

When `pc` encounters a `%if` directive, it evaluates *expression*. If *expression* is `true`, `pc` executes the statements in the `%then` part. If *expression* is `false`, `pc` skips over the `%then` part and executes the `%else`, `%elseif`, or `%endif` directive. If no such directive exists, `pc` proceeds to the next statement.

The *expression* consists of a conditional variable and the optional `boolean` operators `and`, `or`, and `not`. You can set a conditional variable on the command-line by using the `-config` option. See "–config" on page 26 for information on this option.

Assuming `one` and `two` are conditional variables, *expression* can be any of the following:

```
one
two
one and two
one or two
not one
not two
```

Example

See the example in the `%else` listing on page 258.

The `%ifdef` Directive

The `%ifdef` directive determines whether or not you previously defined a conditional variable in a `%var` directive.

Syntax

```
%ifdef expression %then
       .
       .
%elseifdef expression %then
       .
       .
%endif
```

Comments

expression consists of a conditional variable and the optional `boolean` operators `and`, `or`, and `not`. See the `%else` listing for examples of *expression*.

`%ifdef` is especially useful for determining whether or not a conditional variable has been declared in an `include` file.

Example

See the example in "The %elseifdef Directive."

The `%include` *Directive*

The `%include` directive inserts lines from the specified file in the input stream.

Syntax

`%include '`*filename*`';`

Comments

When `cppas` encounters the `%include` directive, it inserts the lines from the file name into the input stream.

Example

The program unit, `include_prog.p`

```
program include_prog;

%include 'extern.h';

begin
    global := 1;
    writeln('From MAIN, before PROC: ',global);
    proc;
    writeln('From MAIN,  after PROC: ',global);
end. { include_prog }
```

The module unit, include_mod.p

```
module include_mod;

define
    global, proc;

%include 'extern.h';

procedure proc;

begin
    writeln('From PROC              : ',global);
    global := global + 1;
end; { proc }
```

The include file, include.h

```
var
    global : integer;

procedure proc; extern;
```

The commands to compile and execute ext_prog.p and ext_mod.p

```
hostname% pc -xl include_prog.p include_mod.p
include_prog.p:
include_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC:1
From PROC       :  1
From MAIN,  after PROC:2
```

The %list Directive

The %list directive enables a listing of the program.

Syntax

%list;

Comments

The `%list` directive and the `-l` compiler option perform the same function.

Example

The Pascal program, `list.p`

```
program list_example(output);

{ This program demonstrates the use of the %list
  and %nolist directives. }

%list;
%include 'types.h';
%nolist;

begin
    pri := [red, yellow, blue];
    pos := [true, false];
    cap := ['A'..'Z'];
    dig := [0..100];
    writeln('There are ',card(pri): 4, 'primary colors.');
    writeln('There are ',card(pos): 4, 'possibilities.');
    writeln('There are ',card(cap): 4, 'capital letters.'');
    writeln('There are ',card(dig): 4, 'digits.')
end. { list_example }
```

The `include` file, `types.h`

```
type
    lowints = 0..100;
    primary_colors = set of (red, yellow, blue);
    possibilities = set of boolean;
    capital_letters = set of 'A'..'Z';
    digits = set of lowints;

var
    pri: primary_colors;
    pos: possibilities;
    cap: capital_letters;
    dig: digits;
```

The listing includes the time each unit was compiled and the name of each unit compiled.

```
hostname% pc -xl list.p
Tue Feb 28 15:48 1995  list.p:
    6  %list;
Tue Feb 28 15:50 1995  ./types.h:
    1  type
    2   lowints = 0..100;
    3   primary_colors = set of (red, yellow, blue);
    4   possibilities = set of boolean;
    5   capital_letters = set of 'A'..'Z';
    6   digits = set of lowints;

    8  var
    9   pri: primary_colors;
    10  pos: possibilities;
    11  cap: capital_letters;
    12  dig: digits;
Tue Feb 28 15:52 1995  list.p:
    7  %include 'types.h';

hostname% a.out
    There are    3 primary colors
    There are    2 possibilities
    There are    26 capital letters
    There are    101 digits
```

The `%nolist` *Directive*

The `%nolist` directive disables the program listing.

Syntax

`%nolist;`

Comments

`%nolist` is the default.

Example

See the example under “The %list Directive.”

The `%slibrary` *Directive*

`cppas` treats `%slibrary` in the same manner as the `%include` directive. See "The %include Directive" on page 267.

The `%var` *Directive*

The `%var` directive defines conditional variables for the preprocessor.

Syntax

`%var` *var1 ..., varN*

Comments

A conditional variable is defined when it appears in a `%var` directive; otherwise, it is undefined.

Example

See the example under "The %config Directive" on page 255.

The `%warning` *Directive*

The `%warning` directive instructs `pc` to print a string on the standard output as a compiler warning.

Syntax

`%warning '`*string*`'`

Comments

`pc` produces an object file.

Example

The Pascal program, `warning.p`

```
program warning_example(output);

{ This program demonstrates the use of the
  %warning compiler directives. }

%var blue

begin
    %if blue %then
        writeln('The color is blue.');
    %else
        %warning 'Color not defined'
    %endif
end. { warning_example }
```

The output when you compile `warning.p` without the `–config` option

```
hostname% pc -xl warning.p
Fri Mar 3 15:03 1995 warning.p

Line 12:%warning 'Color not defined'
w -------------------^----- 'Architecture not defined'
```

Error Messages B

The following is a list of the error messages produced by Pascal, arranged by message number.

`10010: Builtin` *<function>* `takes exactly` *<number>* `argumen`*ts*

`10020: Builtin` *<function>* `takes at least` *<number>* `arguments`

`10030: Builtin` *<function>* `takes at least` *<number>* `arguments and at most` *<number>*

`10040: Built-in` *<function>* `cannot be passed as a parameter`

`10050: argv takes two arguments`

`10060: argv's first argument must be an integer, not` *<type>*

`10070: argv's second argument must be a string, not` *<type>*

`10080: Argument to card must be a set, not` *<type>*

`10090: flush takes at most one argument`

`10100: flush's argument must be a file, not` *<type>*

`10110: Not enough arguments to` *<function>*

10120: Too many arguments to *<function>*

10130: Actual argument cannot be conformant array

10140: Actual argument is incompatible with formal var parameter *<identifier>* of *<function>*

10150: Actual argument is incompatible with formal *<paramtype>* parameter *<identifier>* of *<function>*

10160: Actual argument to NONPASCAL procedure cannot be conformant array

10170: Extra arguments to ADDR ignored

10180: Argument to *<function>* must be of type *<type>*, not *<type>*

10190: Second argument to *<function>* must be of type *<type>*, not *<type>*

10200: First argument to *<function>* must be of type *<type>*, not *<type>*

10210: Illegal argument to IN_RANGE

10220: Type clash in argument to IN_RANGE

10230: Illegal argument to ADDR

10240: Third argument to *<function>* must be of type *<type>*, not *<type>*

10250: Argument to ADDR is a dynamically allocated variable *<identifier>*

10260: Argument to ADDR is an internal variable *<identifier>*

10270: Argument to ADDR is a nested function *<function>*

10280: Argument to ADDR is an internal procedure *<function>*

10290: Fourth argument to *<function>* must be of type *<type>*, not *<type>*

10300: First argument to *<function>* cannot be a univ_ptr

10310: *<number>* argument to *<function>* must be of type *<type>*, not *<type>*

10320: *<number>* argument to *<function>* must be unpacked

10330: *<number>* argument to *<function>* must be packed

10340: *<function>* (line *<number>*) has *<number>* arguments

10350: Transfer functions take exactly one argument

10360: sizeof takes at least 1 argument

10370: Formal arguments should be given only in forward declaration

10380: Types must be specified for arguments

10390: Each procedure/function argument must be declared separately

10400: *<function>* takes no arguments

10410: *<function>* takes either zero or one argument

10420: *<function>* takes exactly one argument

10430: *<function>*'s argument must be integer or real, not *<type>*

10440: seed's argument must be an integer, not *<type>*

10450: *<function>*'s argument must be a real, not *<type>*

10460: *<function>*'s argument must be an integer or real, not *<type>*

10470: ord's argument must be of scalar type, not *<type>*

10480: *<function>*'s argument must be of scalar type, not *<type>*

10490: odd's argument must be an integer, not *<type>*

10500: chr's argument must be an integer, not *<type>*

10510: Argument to eoln must be a text file, not *<type>*

10520: Argument to eof must be file, not *<type>*

10530: Transfer functions take only one argument

10540: Arguments to *<function>* must be variables, not expressions

10550: Read requires an argument

10560: Write requires an argument

10570: Message requires an argument

10580: null takes no arguments

10590: *<function>* expects one argument

10600: Argument to *<function>* must be a file, not *<type>*

10610: *<function>* expects one or two arguments

10620: First argument to *<function>* must be a file, not *<type>*

10630: Second argument to *<function>* must be a string, not *<type>*

10640: *<function>* expects at least one argument

10650: (First) argument to *<function>* must be a pointer, not *<type>*

10660: Second and successive arguments to <*function*> must be constants

10670: Argument to <*function*> must be a alfa, not <*type*>

10680: halt takes no arguments

10690: stlimit requires one argument

10700: stlimit's argument must be an integer, not <*type*>

10710: remove expects one argument

10720: remove's argument must be a string, not <*type*>

10730: linelimit expects two arguments

10740: linelimit's second argument must be an integer, not <*type*>

10750: linelimit's first argument must be a text file, not <*type*>

10760: page expects one argument

10770: Argument to page must be a text file, not <*type*>

10780: Assert expects one or two arguments

10790: Second argument to assert must be a string, not <*type*>

10800: pack expects three arguments

10810: unpack expects three arguments

10820: Illegal transfer function argument

10830: constant argument expected

10840: Illegal argument with format radix specification; probably a comma missing

11010: have incompatible conformant array schemas

11020: sizeof(conformant array) is undefined

11030: Conformant arrays are not allowed at ANSI Level 0

11040: Subscripting allowed only on arrays, not on *<type>*s

11050: subrange value or array subscript (*<integer>*) is out of range

11060: compiler takes size of array

11070: Elements of a packed array cannot be passed by reference

11080: Subscripting allowed only on arrays and varying, not on *<type>*s

11090: Varying size must be a constant

11100: Size of *<identifier>* is zero

11110: Character array lower bound <> 1

11120: Character array upper bound of 1

11130: *<function>* requires a to be an unpacked array, not *<type>*

11140: *<function>* requires z to be a packed array, not *<type>*

11150: *<operation>* not allowed on arrays - only allow = and <>

11160: Packed multidimensional conformant arrays are not permitted

11170: For-statement variable *<identifier>* cannot be an element of a record

11180: . allowed only on records, not on *<type>*s

11190: *<identifier>* is not a field in this record

11200: Record required when specifying variant tags

11210: Missing case constants in variant record

11220: *<identifier>* is a duplicate field name in this record

11230: Duplicate variant case label in record

11240: *<operation>* not allowed on records - only allow = and <>

11250: Variable in with statement refers to *<identifier>*, not to a record

11260: Too many tag fields

11270: No variant case label value equals specified constant value

11280: Tag fields cannot be *<type>*s

11290: Bad variant constant

11300: variant required after case

12010: Assert expression must be Boolean, not *<type>*s

12020: illegal transfer function with *<identifier>*

12030: Illegal transfer function

12040: Transfer functions on bit fields not implemented

12050: Oct/hex allowed only on writeln/write calls

12060: Width expressions allowed only in writeln/write calls

12070: Cannot write *<type>*s with two write widths

12080: Can't write *<identifier>*s with oct/hex

12090: First write width must be integer, not *<type>*

12100: Second write width must be integer, not *<type>*

12110: Negative widths are not allowed

12120: Cannot write unpacked array to textfile

12130: Can't read *<type>*s from a text file

12140: Can't 'readln' a non text file

12150: Oct/hex allowed only on text files

12160: Write widths allowed only on text files

12170: Cannot write unpacked array to textfile

12180: Can't write *<type>*s to a text file

12190: Can't 'writeln' a non text file

13010: constant identifier required

13020: Constant value cannot be evaluated at compile-time

13030: Constant too large for this implementation

13040: *<identifier>* is a constant and cannot be qualified

13050: Constant string too long

13060: Constant expression is required

13070: constant argument expected

13080: newline in string or char constant

13090: empty character constant

13100: too many characters in character constant

14010: Constant declarations should precede type, var and routine declarations

14020: Label declarations should precede const, type, var and routine declarations

14030: Type declarations should precede var and routine declarations

14040: All types should be declared in one type part

14050: All constants should be declared in one const part

14060: INTERNAL ignored, procedure was previously declared PUBLIC or as EXTERNAL

14070: *<function>* has already been declared forward

14080: Unknown language *<identifier>* in EXTERN procedure declaration ignored

14090: Unresolved forward declaration of *<function>*

14100: *<identifier>* DEFINED, but not declared

14110: label *<identifier>* was declared but not defined

14120: PUBLIC procedures must be declared at outer block level

14130: PRIVATE ignored, procedure was declared DEFINED previously

`14140: PUBLIC ignored, procedure was declared INTERNAL or PRIVATE previously`

`14150: PRIVATE ignored, procedure was declared PUBLIC or as EXTERNAL previously`

`14160: For-statement variable` *<identifier>* `must be declared in the block in which it is used`

`14170: All labels should be declared in one label part`

`14180: Expected identifier VARYING in declaration`

`14190: Variable declarations should precede routine declarations`

`14200: All variables should be declared in one var part`

`14210: public vars must be declared at outer block level`

`14220: Declarations may not be both STATIC and PUBLIC, STATIC is ignored`

`14230: Declarations may not be both DEFINE and PRIVATE, DEFINE is ignored`

`14240: Declarations may not be both EXTERN and PRIVATE, EXTERN is ignored`

`14250:` *<identifier>* `was declared in a DEFINE, cannot be STATIC`

`14260:` *<identifier>* `was declared in a DEFINE, cannot be PRIVATE`

`14270:` *<identifier>* `used as both a field and a type name in a record definition`

`14280: cannot DEFINE` *<identifier>*`, variable was not previously declared as EXTERN`

`14290: Declaration found when statement expected`

`14300: Expected keyword begin after declarations, before statements`

`14310: Improper initialization for variable` *<identifier>*

`14320:` *<identifier>* `is already defined globally`

`14330: Definition of name` *<identifier>* `after applied use in` *<identifier>*

`14340: Definition of name` *<identifier>* `after applied use`

`14350:` *<identifier>* `is already defined in this block`

`14360: Range lower bound exceeds upper bound`

`14370: '*' subrange descriptor used in illegal context`

`14380: Cannot initialize dynamic local variables`

`15010: File` *<identifier>* `listed in program statement but not declared`

`15020: File` *<identifier>* `listed in program statement but declared as a` *<identifier>*

`15030: File` *<identifier>* `listed in program statement but defined as` *<identifier>*

`15040: Files cannot be passed by value`

`15050: Files cannot be a component of` *<identifier>* `passed by value`

`15060: Pre-defined files input and output redefined`

`15070: Files cannot be members of files`

`15080: Missing closing` *<quote or bracket>* `for include file name`

`15090: Include filename must end in .i or .h`

15100: Cannot open #include file <*filename*>

15110: line <*number*> does not exist in file <*identifier*>

15120: End-of-file expected - QUIT

15130: Unexpected end-of-file - QUIT

16010: <*identifier*> is not a function

16020: Too many levels of function/procedure nesting

16030: No assignment to the function variable

16040: Functions should not return <*type*>s

16050: Procedure/function nesting too deep

16060: pcc_fvar(): no result for this function

16070: <*identifier*> is a <*class*>, not a function

16080: Illegal function qualification

16090: compiler takes size of function

16100: Can't call <*identifier*>, its not a pointer to a procedure or function

16110: Can't qualify a function result value

16120: Cannot assign value to built in function

16130: INTERNAL option illegal for procedure pointer - option ignored

16140: Unknown option for procedure pointer ignored : <*option*>

16150: Too many levels of function/procedure nesting

16160: Procedure/function nesting too deep

16170: Can't call *<identifier>*, its not a pointer to a procedure or function

16180: Can't call *<identifier>*, it's *<class>* not a procedure

16190: Procedure *<identifier>* found where expression required

16200: Illegal procedure call, expecting value

16210: Non-pascal routine *<identifier>* will fail if called indirectly from pascal

16220: Passing nested routine *<identifier>* to non-pascal routine *<identifier>*

16230: Can't call *<identifier>*, its not a pointer to a procedure or function

17010: Case label out of range

17020: Real constant out of range for this implementation

17030: Short real out of range for this implementation

17040: Short real *<number>* out of range for this implementation

17050: Implementation restriction: sets must be indexed by 16 bit quantities

17060: Subscript of *<identifier>* is out of range

17070: Subscript value of *<number>* is out of range

17080: subrange value or array subscript (*<integer>*) is out of range

17090: Successor of *<integer>* is out of range

`17100: Predecessor of` *<integer>* `is out of range`

`17110: Value of` *<integer>* `is out of range`

`17120: Range upper bound of` *<integer>* `out of set bounds`

`17130: Range lower bound of` *<integer>* `out of set bounds`

`17140: Value of` *<integer>* `out of set bounds`

`17150: value of` *<integer>* `(initial assign. to for loop variable) is out of range`

`17160: Value of` *<integer>* `(terminal assign. to for loop variable) is out of range`

`17170: Tag type is out of range`

`17180: Base out of range (2..36)`

`18020: (* in a (* ... *) comment`

`18030: { in a { ... } comment`

`18040: Comment does not terminate - QUIT`

`18050: Illegal character, use the -xl option to process % compiler directives`

`18060: Illegal character`

`18070: unexpected EOF`

`18080: Point of error`

`18090: Parsing resumes`

`18100: Parse stack overflow`

`18110: Expression passed to UNIV formal` *<identifier>* `was converted to` *<type>* `of size` *<number>* `bytes`

18120: 8 or 9 in octal number

18130: Number too large for this implementation

18140: *<operation>* is undefined for reals

18150: *<identifier>* cannot have more elements in z (*<integer>*) than in a (*<integer>*)

18160: *<identifier>* is an unimplemented extension

18180: Initializer must be a string

18190: Initialization string *<string>* is too long

18200: Expected 'options', not *<identifier>*

18210: Unknown option ignored : *<identifier>*

18220: Duplicate options specification : *<identifier>*

18230: Unimplemented option ignored : *<identifier>*

18240: *<identifier>* undefined on line *<number>*

18250: *<identifier>* improperly used on line *<number>*

18260: *<identifier>* is neither used nor set

18270: *<identifier>* is never used

18280: *<identifier>* is used but never set

18290: Length of external label for identifier *<identifier>* exceeds implementation limit

18300: Applied use of *<identifier>* before definition in this block

18320: *<operation>* is forbidden for reals

18420: Undefined *<identifier>*

18430: Undefined identifier

18440: Improper *<identifier>* identifier

18450: Deleted *<token>*

18460: Replaced *<token>* with a *<token>*

18470: Replaced *<class>* id with a *<class>* id

18480: Inserted *<token>*

18490: Expected *<token>*

18500: Label *<identifier>* not defined in correct block

18510: Label *<identifier>* redefined

18520: *<identifier>* is undefined

18530: ^ allowed only on files and pointers, not on *<type>*s

18540: Pascal uses [] for subscripting, not ()

18550: Error occurred on qualification of *<identifier>*

18560: division by 0

18580: Variable required in this context

18590: Universal pointers may not be dereferenced

18600: Illegal format

18610: Unknown format radix *<identifier>* ignored, expecting hex or oct

18620: Expression required

```
18630: Unreachable statement

18640: Unrecoverable syntax error - QUIT

18650: Too many syntax errors - QUIT

18660: Input line too long - QUIT

18670: Unrecognizable # statement

18680: Include syntax error - expected ' or \" not found

18690: Absurdly deep include nesting - QUIT

18700: Too many levels of include nesting - QUIT

18710: Bad arbitrary base integer

18720: Bad base <number> number, <number>

18730: Digits required after decimal point

18740: Digits required before decimal point

18750: Digits required in exponent

18760: Space required between number and word-symbol

18770: Character/string delimiter is '

18780: Unmatched ' for string

18790: Null string not allowed

18800: Invalid preprocessed file (probably renamed as
'.p').  Compile the original .p file with -sb again

19010: Module name expected after keyword MODULE

19020: Identifiers after module name ignored
```

19030: Missing program statement

19040: Input is used but not defined in the program statement

19050: Output is used but not defined in the program statement

19060: Program parameter *<identifier>* is repeated

20010: Left operand of *<operator>* must be integer, real or set, not *<type>*

20020: Right operand of *<operator>* must be integer, real or set, not *<type>*

20030: Cannot mix sets with integers and reals as operands of *<type>*

20040: Operands of *<operator>* must be Boolean or Integer

20050: Left operand of / must be integer or real, not *<type>*

20060: Right operand of / must be integer or real, not *<type>*

20070: Left operand of *<operator>* must be Boolean, not *<type>*

20080: Right operand of *<operator>* must be Boolean, not *<type>*

20090: Left operand of *<operator>* must be integer, not *<type>*

20100: Right operand of *<operator>* must be integer, not *<type>*

20110: Right operand of 'in' must be a set, not *<type>*

20120: Operands of + are of incompatible types

20130: Operand of *<operator>* must be integer or real, not *<type>*

20140: Operands of *<operator>* must both be sets

20150: Set types of operands of *<operator>* must be compatible

20160: Incorrect statement usage const. with operand

20170: ^ allowed only on files and pointers, not on *<type>*s

20180: *<operation>* is an illegal operation on strings

20190: not must operate on a Boolean or Integer, not *<type>*

20200: not must operate on a Boolean, not *<type>*

20210: *<operation>* not allowed on pointers - only allow = and <>

20220: Strings not same length in *<operator>* comparison

20230: *<identifier>*s and *<identifier>*s cannot be compared - operator was *<operator>*

20240: *<identifier>*s may not participate in comparisons

21010: Attempt to pass IN parameter to formal reference parameter

21020: Expression type clashed with type of value parameter *<identifier>* of *<identifier>*

21030: Expression given (variable required) for var parameter *<identifier>* of *<identifier>*

21040: Parenthesis not allowed for var parameter *<identifier>* of *<identifier>*

21050: Parameter type not identical to type of var parameter *<identifier>* of *<identifier>*

21060: Packed/unpacked type conflict between formal and actual parameters

21070: Conformant array parameters in the same specification must be the same type

21080: actual parameter is not an array

21090: array index type is incompatible with conformant array parameter *<identifier>*

21100: array index type is not within of index type of conformant array parameter *<identifier>*

21110: array index type is not within range[*<number>*..*<number>*] of index type of conformant array parameter *<identifier>*

21130: *<class>* *<identifier>* given for formal *<class>* parameter *<identifier>*

21140: does not match type of formal *<class>* parameter *<identifier>* (line *<number>*)

21150: *<class>* parameter *<identifier>* of *<identifier>* (line *<number>*)

21160: does not match *<class>* parameter *<identifier>* of *<identifier>* (line *<number>*)

21170: Type of *<class>* parameter *<identifier>* of *<identifier>* (line *<number>*)

21180: does not match type of *<class>* parameter *<identifier>* of *<identifier>* (line *<number>*)

21190: Parameter congruity error: incompatible groupings

21200: Packed/unpacked type conflict between parameters of *<identifier>* and *<identifier>*

21210: than formal *<class>* parameter *<identifier>* (line *<number>*)

21220: Conformant array parameter bound symbol table entry is NULL

21230: Program parameter *<identifier>* is repeated

21240: Previous declaration for formal parameter '*<identifier>*' has different type

21250: Previous declaration for formal parameter '*<identifier>*' has different name '*<identifier>*'

21260: Previous declaration for procedure '*<identifier>*' had different number of formal parameters

21270: Formal *<class>* *<name>* cannot be qualified

21280: *<class>* *<name>* cannot be qualified

21290: Expression given, *<class>* required for *<class>* parameter *<identifier>*

21300: Variable given, *<class>* required for *<class>* parameter *<identifier>*

21310: Cannot take value of OUT parameter

21320: Invalid assignment to a parameter marked as IN

21330: Parameter congruity error: incompatible groupings

21340: UNIV parameter *<identifier>* should be passed as a var parameter

21350: Fields which select variant parts may not be passed by reference

21360: Cannot pass components of packed structures by reference

21370: Cannot pass components of packed structures by VAR, OUT, or IN/OUT

22010: Ran out of memory (gentypind)

```
22020: Ran out of memory (case)

22030: Ran out of memory (hash)

22040: Ran out of memory (TRIM)

22050: Ran out of memory (defnl)

22060: Ran out of memory (buffer_ir_pass)

22070: Ran out of memory (string)

22080: Ran out of memory (tralloc)

22090: out of memory (tstr)

22100: out of memory

22110: Ran out of hash tables

22120: Ran out of tree tables

22130: Out of space (put_on_idlist)

22140: out of tree space; try simplifying

22150: out of temporary string space
```

`23010:` *<identifier>* `is a non-standard function`

`23020:` *<identifier>* `is a non-standard procedure`

```
23030: Two argument forms of reset and rewrite are non-
standard

23040: NIL values in const declaration are non-standard

23050: Set constants in const declaration are non-standard

23060: Expressions in const declaration are non-standard
```

23070: Routine Options are not standard

23080: External procedures and functions are not standard

23090: Separately compiled routine segments are not standard

23100: UNIV parameters are non-standard

23110: IN parameters are non-standard

23120: OUT parameters are non-standard

23130: IN OUT parameters are non-standard

23140: *<identifier>* is a nonstandard function

23150: OTHERWISE clause in case statements is non-standard

23160: Ranges in case statements are non-standard

23170: Transfer functions are non-standard

23180: Reading scalars from text files is non-standard

23190: Writing *<type>*s with two write widths is non-standard

23200: Zero widths are non-standard

23210: Oct and hex are non-standard

23220: Writing *<type>*s to text files is non-standard

23230: *<identifier>* is a nonstandard procedure

23240: Short-circuit operators are non-standard

23250: *<operator>* comparison on sets is non-standard

23260: record comparison is non-standard

`23270: Storage Class non-standard`

`23280: Initialization in declaration part is non-standard`

`23290: UNIV_PTR types are non-standard`

`23300: '_' in an identifier is nonstandard`

`23310: Octal constants are non-standard`

`23320:` *<operator>* `is non-standard`

`24010: Cannot exit -- not within a loop`

`24020: For-statement variable` *<identifier>* `must be unqualified`

`24030: For-statement variable` *<identifier>* `cannot be an element of a record`

`24040: For-statement variable` *<identifier>* `may be illegally changed at line` *<number>*

`24050: For-statement variable` *<identifier>* `must be declared in the block in which it is used`

`24060: For-statement variable` *<identifier>* `cannot be` *<type>*`s`

`24070: Can't modify the for-statement variable` *<identifier>* `in the range of the loop`

`24080: Incorrect control variable`

`24090: Type of initial expression clashed with index type in 'for' statement`

`24100: Type of limit expression clashed with index type in 'for' statement`

`24110: Can't modify the for variable` *<identifier>* `in the range of the loop`

24120: Case selectors cannot be *<type>*s

24130: Duplicate otherwise clause in case statement

24140: Case label type clashed with case selector expression type

24150: Multiply defined label in case, lines *<number>* and *<number>*

24160: No case-list-elements

24170: Bad case constant

24180: Range in case label must be in ascending order

24190: Maximum number of case selectors exceeded

24200: Cannot next -- not within a loop

24210: Goto *<label>* is into a structured statement

24220: Goto *<label>* from line *<number>* is into a structured statement

24230: Variable in with statement refers to *<type>*, not to a record

24240: Maximum WITH nesting depth exceeded, ignoring WITH variables

24250: Variable in with statement not correct

24260: cannot assign to lhs

24270: Variable required

24280: *<class>* *<identifier>* found where variable required

24290: univ_ptr variable cannot be dereferenced in assignment

`24300: Improper use of the DEFINE statement`

`24310: End matched` *<keyword>* `on line` *<number>*

`24320: Inserted keyword end matching {begin|record|class} on line` *<number>*

`25010: {pack|unpack}: elements of a and z must have the same type`

`25020: component types of formal and actual arrays are not conformable`

`25030: Type of function` *<name>* `(line` *<number>*`)`

`25040: Case label type clashed with case selector expression type`

`25050: Type clash: initializer must be a pointer`

`25060: Type clash: type of initializer does not match type of array`

`25070: Type clash: Integer is incompatible with real`

`25080: Type clash:  Initializer out of range`

`25090: Type clash: real is incompatible with integer`

`25100: This resulted because you used '/' which always returns real rather`

`25110: than 'div' which divides integers and returns integers`

`25120: Type clash: non-identical scalar types`

`25130: Type clash: unequal length strings`

`25140: Type clash: varying char and unknown`

25150: Type clash: packed and unpacked set

25160: Type clash: files not allowed in this context

25170: Type clash: non-identical *<class>* types

25180: Type clash: *<type>*s with file components not allowed in this context

25190: Type clash: *<type>* is incompatible with *<type>*

25200: Type clash: string and unpacked array

25210: Type clash: string and packed array with lower bound <> 1

25220: Type clash: Non-identical string types must be packed

25230: Type clash: packed array with lower bound <> 1

25240: Set default type 'intset' is not a set

25250: Upper bound of element type clashed with set type in constant set

25260: Lower bound of element type clashed with set type in constant set

25270: Element type clashed with set type in constant set

25280: Set elements must be scalars, not *<type>*s

25290: Set of real is not allowed

25300: Procedures cannot have types

25310: Function type can be specified only by using a type identifier

25320: Function type should be given only in forward declaration

25330: Different type declared previously for function return *<name>*

25340: Function type must be specified

25350: Type of expression in while statement must be Boolean, not *<type>*

25360: Until expression type must be Boolean, not *<type>*, in repeat statement

25370: Array index type incompatible with declared index type

25380: Too many subscripts

25390: Bad type of variable used for call

25400: Transfer functions only work on types of the same size

25410: Only type char allowed in varying array

25420: Type mismatch in read from non-text file

25430: Type mismatch in write to non-text file

25440: Specified tag constant type clashed with variant case selector type

25460: Tag type is out of range

25470: Variant label type incompatible with selector type

25480: set types must be compatible in comparisons - operator was *<operator>*

25490: *<class>* types must be identical in comparisons - operator was *<operator>*

25500: Index type clashed with set component type for 'in'

25510: Operands of + are of incompatible types

25520: Type names (e.g. *<type>*) allowed only in declarations

25530: Type name (e.g. *<type>*) found in illegal context

25540: Set types of operands of *<operator>* must be compatible

25550: expression has invalid type

25560: Type of expression clashed with type of variable in assignment

25570: Type of expression in if statement must be Boolean, not *<type>*

25580: Type of transfer function does not match type of right side

25590: *<identifier>* is a *<class>*, not a type as required

25600: Set type must be range or scalar, not *<type>*

25610: Sets must have basetype with range *<number>*..*<number>*

25620: Subrange type cannot contain a expression

25630: Scalar types must be identical in subranges

25640: Can't mix *<identifier>*s and *<identifier>*s in subranges

25650: Subrange of real is not allowed

25660: Subrange bounds must be Boolean, character, integer or scalar, not *<type>*

25670: Index type for arrays cannot be real

25680: Array index type is a *<type>*, not a range or scalar as required

25690: Packed conformant array schema requires type identifier as element type

25700: Illegal type in VARYING array, only type CHAR allowed

25710: Size of *<type>* type exceeds implementation limits

25720: Initializer must be a constant value of same type

26010: Malformed program statement

26020: Malformed declaration

26030: Malformed const declaration

26040: Malformed type declaration

26050: Malformed var declaration

26060: Malformed record declaration

26070: Malformed statement in case

26080: Malformed statement

26090: Missing/malformed expression

26100: Label restricted to 4 digits

26110: Label declared non-integer

26120: Replaced unknown parameter specifier *<identifier>* with OUT

26130: Deleted ';' before keyword else

`26140: Extension to WITH statement not allowed`

`27010: Integer overflow in constant expression`

The following are internal error messages. If any of them is displayed, contact Sun Customer Support.

`18330: NAME with no symbol table entry`

`18340: ir_symbol: unknown class`

`18350: ir_leaf: cannot take address of op` *<identifier>*

`18360: ir_leaf: illegal leaf node, op =` *<identifier>*

`18370: ir_starg: illegal STARG`

`18380: ir_exit: no return label`

`18390: cannot take address of op` *<identifier>*

`18400: op '`*<identifier>*`' not implemented yet`

`18410: goal` *<integer>* `not defined for op` *<operation>*

`18570: wasted space:` *<number>*

Index