Pascal Programming Guide

Document Number 007-0740-030

Contributors

Written by David Graves
Edited by Janiece Carrico
Production by Laura Cooper
Engineering contributions by David Ciemiewicz, Mike Fong, Ken Harris, Mark Libby, Claudia Lukas, Andrew Myers, Jim Terhorst, and Bill Johson

Pascal Programming Guide
Document Number 007-0740-030

Silicon Graphics, Inc.
Mountain View, California

Contents

Figures

Tables

Introduction

This guide explains how to use the Silicon Graphic Pascal compiler. Two assumptions are made of the Pascal programmer using the IRIS programming environment:

- You are fluent in the Pascal language.
- You are comfortable with the IRIX tools that exist on the IRIS.

Organization

This guide contains the following chapters and appendices:

Chapter 1, “Pascal Implementation,” lists and describes the set of extensions that are supported by the Pascal compiler on the IRIS. The *1.2 Pascal Release Notes* provide a detailed list of changes from the ANSI standard.

Chapter 2, “Compiling, Linking, and Running Pascal Programs,” lists specific commands for compiling, linking, and running Pascal programs on the IRIS-4D. It also explains the tools used in each command procedure.

Chapter 3, “Storage Mapping,” explains storage mapping of Pascal arrays, records, and variant records. Aslo discussed are alignment, size and value ranges for the various data types.

Chapter 4, “Pascal/C Interface,”describes how to use command line options to optimize PFA execution. It also describes the Pascal/C coding interface. The *Fortran 77 Programmer’s Guide* describes the Pascal/Fortran interface.

Appendix A, “Man Pages” is a listing of the *pc(1) man page*.

Pascal Programming Environment

To write Pascal graphics and other programs in the IRIS-4D programming environment, the programmer uses these tools:

- The IRIS-4D Pascal Graphics Library
- The Pascal compiler
- The IRIX debugger *dbx*

Programming Procedure

Write and run your Pascal programs according to the procedure listed below. For specific instructions about each step, refer to the chapter or document named in that step.

1. Write programs using the Pascal implementation described in Chapter 1 of this guide.
2. Use graphics routines according to the rules in the *Graphics Library Programming Guide* and the man pages for the routines.
3. Compile, link, and run the programs as described in Chapter 2 of this guide.
4. Debug the programs using *dbx*.
5. Optimize the programs as described in the *IRIS-4D Series Compiler Guide*.

There is additional information in this manual that can be useful when you write programs:

- Storage mapping is described in Chapter 3.
- Interfaces between Pascal and C are described in Chapter 4.
- For reference, the Pascal compiler manual page is included in Appendix A. You can also view it online by entering *man pc* at your system prompt.

Related Documentation

The following documents contain information relevant to PFA:

- *IRIS-4D Series Compiler Guide*, Silicon Graphics, Inc., document number 007-0905-030.

Notation and Syntax Conventions

This guide uses the following notation and syntax conventions:

italic	Indicates arguments in a command line that you must replace with a valid value. In text it is used to indicate commands, document titles, file names, glossary items, new terms, and variables.
`courier`	Indicates computer output and program listings.
`courier bold`	Indicates computer input and non-printing keys.
[ ]	Brackets enclose optional command arguments. Do not enter the brackets.
. . .	An ellipsis indicates that the preceding optional items can appear more than once in succession.
( )	Parentheses enclose items. Enter them exactly as shown.
{ }	Braces enclose items from which you must select exactly one. Do not enter the braces.
|	The vertical bar separates items from which you can choose one.

Chapter 1

Pascal Implementation

The Pascal language supported by the compiler is an implementation of ANSI Standard Pascal (ANSI/IEEE770X3.97-1983). This implementation complies with ANSI requirements except for the extensions. When this chapter refers to "SGI Pascal," it means the IRIS-4D extended implementation of Pascal.

This chapter lists and describes the set of extensions that are supported by the Pascal compiler on the IRIS. The *1.2 Pascal Release Notes* provide a detailed list of changes from the ANSI standard.

This implementation extends the ANSI definition of Pascal to provide features for application development, and to meet the following objectives:

- Provide extensions that make Pascal usable for a wide class of programs
- Anticipate the requirements of programmers
- Be consistent with the direction of the emerging extended standard
- Be consistent with the IRIX/C programming environment

Read Section 1.1 to learn how ANSI Pascal is extended in this implementation.

1.1 Names

There are three extensions for names.

1.1.1 Use of Underscores

SGI Pascal allows underscores (_) in identifiers. You can use underscores to make names that are composed of several words. You can also use an underscore as the first character of an identifier.

This feature is consistent with the use of underscores in the C language, making calls between SGI Pascal routines and C functions compatible.

Examples are shown below.

```
read_integer
_bits_per_word
```

1.1.2 Lowercase in Public Names

Pascal is not case-sensitive to variable names. Be aware that this causes problems for Pascal names intended to be linked with C functions, because case is significant in C.

SGI Pascal converts to lowercase all characters in the names of external variables, procedures, or functions.

1.1.3 Alphabetic Labels

SGI Pascal permits alphabetic labels in addition to numeric labels as specified by ANSI Pascal.

1.2 Constants

There are four extensions for constants.

1.2.1 Non-Decimal Number Constants

SGI Pascal allows the use of any radix from base 2 to 36.

It is often useful to write integer constants in a radix other than base 10. This occurs in programs that use data structures defined by the system. After base 10, base 2, 8, and 16 are the most frequently used bases.

You can specify a number in these bases with this form:

```
base#number
```

where *base* is a decimal number in the range 2 to 36, and *number* is a number in the specified radix using *a..z* (either case) to denote the values 10 through 25.

The number must specify a value in the range *0..maxint* (2147483647). The example below shows the number 42 as it would be written in several different bases:

```
2#101010
3#1120
4#222
8#52
42
10#42
11#39
16#2a
```

If the radix is a power of two (2, 4, 8, 16, or 32), the number may be negative if it specifies a 32-bit value. For example, 16#8000000 is -2147483648, which is the smallest integer contained in a 32-bit number.

1.2.2 String Padding

SGI Pascal pads a string constant with blanks on the right according to its use. That is, assigning a 3-character literal to a 6-character variable causes the string literal to be treated as being 6 characters long. Assigning a 6-character literal string to a variable containing three characters causes an error.

ANSI Pascal requires that a literal character string have the same length as the variable with which it is used. Manually adding extra blanks invites errors, and changing a Pascal type definition would require manually changing all literal strings that are used with variables of that type.

1.2.3 Non-Graphic Characters

Literal character strings cannot contain ASCII characters that have no graphic representation. Control characters are the most commonly used characters having no graphic representation.

SGI Pascal has a special form of character string, enclosed in double quotation marks, in which such characters may be included. A backslash (\ ) escape character is used to signal the use of a special character. For example, the following line rings (beeps) the bell on an ASCII terminal.

```
writeln(output, "\a")
```

Table 1-1, following, lists escape character sequences used to encode special characters.

Character	Result
\a	alert (16#07)
\b	backspace (16#08)
\f	form feed (16#0c)
\n	newline (16#0a)
\r	char (16#0d)
\t	horizontal tab (16#09)
\v	vertical feed (16#0b)
\\	backslash (16#5c)
\"	quotation mark (16#22)
\'	single quote (16#27)
nnn	character with octal value of *nnn*
xnnn	character with hexadecimal value of *nnn*.

Table 1-1 Escape Character Sequences

1.2.4 Constant Expressions

Using a constant expression in type definitions or constant definitions often makes programs easier to read and maintain. The following example shows one use of a constant expression. Changing a single definition (*array_size*) changes both the size of the array and the definition of the index type used to access it.

```
const
  array_size = 100;
type
  array_index = 0..array_size-1;
var
v : array[ array_index ] of integer;
```

SGI Pascal permits you to use a constant expression where you might ordinarily use a single integer or scalar constant. An expression can consist of any of the following operators and predefined functions, as shown in Table 1-2.

Operator	Function
+	addition
-	subtraction and unary minus
*	multiplication
div	integer division
mod	modulo
=	equality relation
<>	inequality relation
<	less than
<=	less than or equal to
>=	greater than or equal to
>	greater than
()	parentheses
bitand	bitwise AND

Table 1-2 Constant Operators and Predefined Functions

Operator	Function
bitor	bitwise OR
bitxor	bitwise exclusive-OR
lshift	logical left shift
rshift	logical right shift
lbound	low bound of array
hbound	high bound of array
first	lowest value of a scalar type
last	highest value of a scalar type
sizeof	the size (in bytes) of a data type
abs	absolute value
chr	inverse ordinal value of a scalar value
ord	the ordinal value of a scalar value
pred	the predecessor of a scalar value
succ	the successor of a scalar value
type-functions	converts from one type to another

Table 1-2 (continued) Constant Operators and Predefined Functions

SGI Pascal does not allow the use of a leading left parenthesis in constant expressions for the lower value of subrange types. That is, Pascal mistakenly assumes that:

```
subrange = (11+12)*13 .. 14+15;
```

is an enumeration instead of a subrange.

1.3 Statement Extensions

There are seven statement extensions.

1.3.1 Ranges in Case Statement Constants

SGI Pascal permits the specification of value ranges in a *case* statement. You can specify the selectors in a case statement using the SGI Pascal range notation to mean all values inclusive, as in the following example.

```
case i of
   1900..1999 : writeln('twentieth century');
   1800..1899 : writeln('nineteenth century');
   1700..1799 : writeln('eighteenth century');
   1600..1699 : writeln('seventeenth century');
otherwise :
   write(i div 100:1);
   writeln('th century');
end
```

1.3.2 Otherwise Clause in Case Statement

SGI Pascal extends the *case* statement to allow an *otherwise* clause, which is the default statement when no case clause equal to the case value exists. You can include any number of statements between *otherwise* and *end*, as in the example in the section above.

If no *otherwise* is specified, and no case clause satisfies the case selector values during execution, a case error warning message is printed.

1.3.3 Return Statement

SGI Pascal provides a *return* statement that permits a procedure or function to return to the caller without branching to the end of the procedure or function.

If used within a function, *return* may optionally supply a function result. Unless this expression is specified, the last value assigned to the function name is the function result. Using the *return* statement is equivalent to assigning the

value in the expression to the function name and executing a *goto* to the end of a routine.

```
function factorial(n:integer):integer;
   begin
      if n = 1 then return(1)
      else return(n*factorial(n-1))
 end;
```

1.3.4 Continue Statement

The *continue* statement causes the flow of control to continue at the next iteration to the nearest enclosing loop statement (*for*, *while*, or *repeat*). If the statement is a *for* statement, the control variable is incremented and the termination test is performed.

```
for J := 1 to i do begin
    if Line[J] = ' ' then continue;
    write(output, Line:J);
end (for);f1
```

1.3.5 Next Statement

The *next* statement performs the same function as the *continue* statement.

1.3.6 Break Statement

The *break* statement causes the flow of control to exit the nearest enclosing loop statement (*for*, *while*, or *repeat*). This feature permits you to end a loop without using the test specified in the loop statement.

```
while p <>  nil do begin
   if p^.flag = 0 then break;
   p := p^.next
end;
```

1.3.7 Exit Statement

The *exit* statement performs the same function as the *break* statement.

1.4 Declaration Extensions

There are six declaration extensions.

1.4.1 Separate Compilation

SGI Pascal permits breaking a program into several compilation units: one that contains the main program, and the others containing procedures or functions called by the main program. (The procedures and functions it calls need not be written in Pascal.)

SGI Pascal also permits separately compiled compilation units to share data.

The compilation unit that contains the main program (the *program compilation unit*) follows ANSI Pascal syntax for a program. Only one program compilation unit is permitted.

A compilation unit that contains separately compiled procedures and functions is called a *separate compilation unit*. In a separate compilation unit, procedure, functions, and variables are placed sequentially without any program headers or main program block.

SGI Pascal allows Pascal declarations to be placed before the program header, whereas ANSI Pascal does not. All procedures, functions, and variables preceding the program header are given external scope. This means that they may be called (as with procedures and functions) or used (as with variables) by separate compilation units.

Figure 1-1 shows the external declarations in a program compilation unit. The top box shows the code in the include file *externs.h*, and the declarations.

In this example, the *external* directives and other statements in the header file *externs.h* specify that *initialize* and *sum* (used by the main program) and their parameters are defined in a separate compilation unit. (The separate compilation unit is shown Figure 1-2.)

You can use the external directive to qualify only unnested routine names.

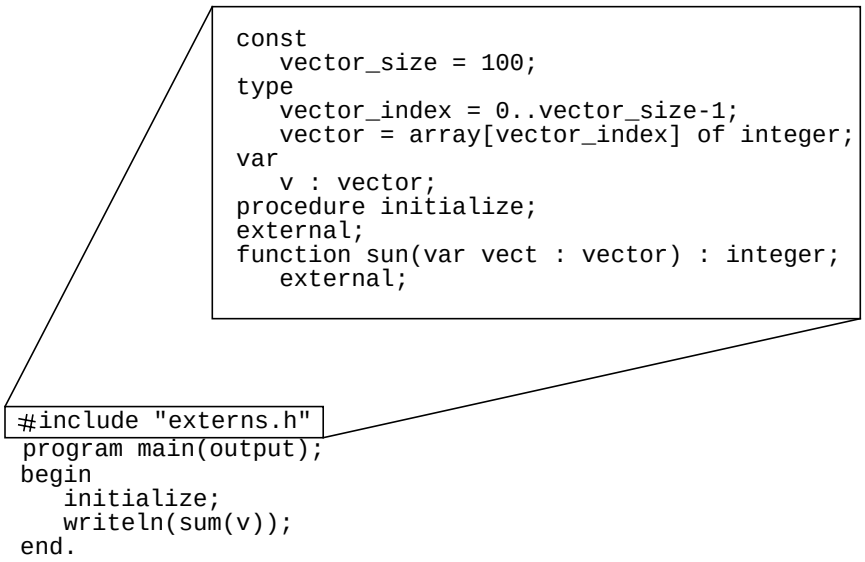


Figure 1-1 External Declarations in Header File

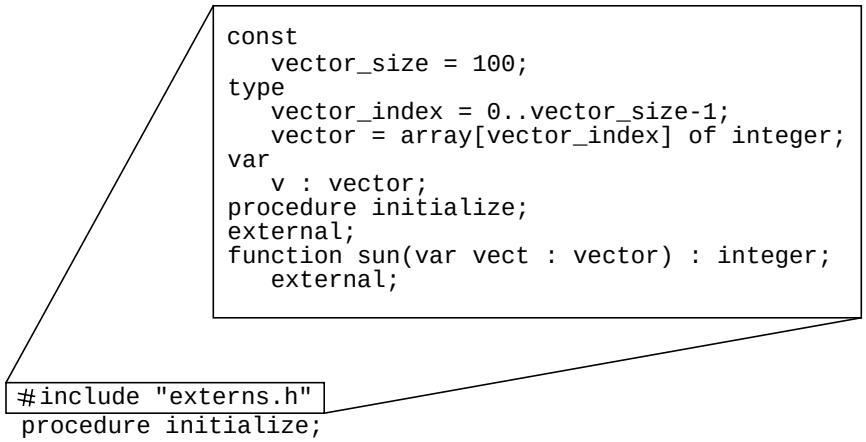
Initialize and *sum* must be defined when the main program is link edited. Consider the two commands below, in which the file *mainone.p* contains the Pascal source code for the main program, and *bodies.o* contains a previously compiled object module of *initialize* and *sum*:

`pc -c mainone.p`	(This compiles *mainone.p*)
`pc -o exec mainone.o bodies.o`	(This link edits *mainone* with *initialize* and *sum*)

The external directive is similar to the Pascal *forward* directive, because it declares a routine name and its parameters without defining the body of the routine. You can use the external directive only to qualify unnested routine names.

All procedures, functions, and variables at the outermost level have external scope.

Continuing with the example on the previous page, Figure 1-2 shows the separate compilation unit in which *initialize* and *sum* are defined with the *external* directive.



```
const
   vector_size = 100;
type
   vector_index = 0..vector_size-1;
   vector = array[vector_index] of integer;
var
   v : vector;
procedure initialize;
external;
function sun(var vect : vector) : integer;
   external;
```

```
#include "externs.h"
procedure initialize;
   var
      i : vector_index;
   begin
      for i :+ lbound(vector to hbound(vector) do v[i] :+ i;
end {procedure initialize};
function sum;
   var
      i : vector_index;
      j : integer;
   begin
      j := 0;
      for i := lbound(vector) to hbound(vector)
      do  j := j+vect[i];
      return (j);
   end {function sum};
```

Figure 1-2 Separate Compilation Unit with External Declarations

Note that, in this example, the external declarations are placed in the include file *externs.h*. This reduces the chance of errors due to inconsistent declarations. Use include files this way for statements shared by multiple compilation units.

1.4.2 Shared Variables

All variables that are declared at the outermost nesting of a compilation unit have an external scope and can be used in different compilation units.

The variable can have only one initializing clause and can be placed in only one compilation unit. Look again at the example above. It illustrates a variable named *v* that is accessed from both compilation units.

1.4.3 Initialization Clauses

SGI Pascal permits an external variable to have a clause that initializes either a scalar, a set, an array, or a pointer. The BNF syntax of the extended variable declaration is shown below.

```
var-decl ::= identifier-list ":" type-denoter
                          [" := "initial-clause]

    initial-clause ::= constant-expr |
     "[" initial-value-list "]"

    initial-value ::= constant-expr [".." constant-expr] |
           constant-expr ":" constant-expr |
           "otherwise" ":" constant-expr |
           "[" initial-value-list "]"

    initial-value-list ::= initial-value |
           initial-value-list "," initial-value
```

For scalar types, this initialization is very simple:

```
var
   a : integer := 5;
   letter : char := 'x';
   x1 : real := 6.5;
```

An initial value may also be given to a structured type (array or set). Every element of an array must be initialized or given a specific default value.

```
type
    color = (red, yellow, blue);
    hue = set of color;
    vector = array[1..100] of integer;
var
    orange : hue := [red, yellow];
    white : hue :=
            [first(color)..last(color);
    name : packed array[1..32] of char :=
            'Silicon Graphics Computer Systems':
    vect : vector :=
           [0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

```
            30 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            otherwise : 99];
pointers : array[0..127] of ^vector := [otherwise: nil];
```

The above clauses initialize two sets, a string, and two arrays. All elements of the array *vect*, except the first 10 elements and the 10 elements starting at index 30, are initialized to 99. The array *pointers* is initialized entirely to the value *nil*.

1.4.4 Relax Declaration Ordering

The declaration clauses described in this section can be written in any order and may be repeated. The ANSI requirement that a declaration must precede any use must still be observed.

1.4.5 Internal and Extern

If you declare a procedure or function with the *internal* attribute, the procedure or function becomes a local rather than global symbol. If you declare a procedure or function *extern*, you can define it in a separate file. In the following example, *x* is internal and therefore local:

```
function x (num : integer) : integer ; internal;
```

1.4.6 Function Return Types

Functions can return scalar types, records, arrays, and sets. For example:

```
program main;
type
    rec = record
    field1 : char;
    field2 : integer;
end;
var
    reca : rec;
function xx : rec;
var recb : rec;
begin
    recb.field1 := 'a';
    recb.field2 := 9;
    xx := recb;
```

```
end;
begin
reca := xx;
end.
```

1.5 Predefined Procedures

There are four extensions of predefined procedures.

1.5.1 Assert

`assert(`*`boolean-expr`* `[,`*`string`*`])`

The *assert* procedure evaluates a boolean expression and signals an execution time error (similar to a checking error) if the value is not true.

If you specify the optional string, the string is written to standard error, otherwise, this message is generated, along with the line number and file name of the *assert* statement that causes the error:

```
assertion error in Pascal program
```

1.5.2 Argv

`argv(`*`integer-expr`*`,`*`string-var`*`)`

The *argv* procedure returns the *i*-th program argument, placing the result in a string. The string can be blank, padded, or truncated, as required.

The value of the first parameter must be in the range 0..*argc-1*. Argument 0 is the name of the program.

1.5.3 Date

`date(`*`string-var`*`)`

The *date* procedure returns the current date.

The resulting string has the form *yy/mm/dd*, where *yy* is replaced with the last two digits of the year, *mm* is replaced with the number of the month (January is 01), and *dd* is replaced with the day of the month (the first day is 01).

If the string is less than 8 characters long, data is truncated on the right; if the string is longer than 8 characters, the string is padded with blanks on the right.

1.5.4 Time

`time(`*`string-var`*`)`

The *time* procedure returns the current time. The resulting string has the form *hh:mm:ss*, where *hh* is replaced with the hour (on the 24-hour clock), *mm* is replaced with minutes (00 means on the hour), and *ss* is replaced with seconds (00 means on the minute).

If the string is less than eight characters, data is truncated on the right; if the string is longer than eight characters, the string is padded with blanks on the right.

1.6 Predefined Functions

There are 20 predefined function extensions.

1.6.1 Type Functions

ANSI Pascal offers two predefined type functions, *ord* and *chr*, that convert predefined scalar types.

SGI Pascal allows all scalar types to have a conversion function that converts an integer into that scalar type. As in ANSI Pascal, the *ord* function converts from the scalar type to integer.

SGI Pascal lets all type identifiers for a scalar type be used in an expression to convert its integer argument into the corresponding scalar value. Thus, if *color* is an enumerated data type, *color(i)* is a function that returns the *i*-th element of the enumeration.

```
boolean-var := boolean (integer-expr)
char-var    := char(integer-expr)
color-var   := color(integer-expr)
```

In Pascal, the *ord* function also operates on pointers, returning the machine address of the item referenced by the pointer. A data type identifier that represents a pointer data type can also be used to convert a cardinal number into a valid pointer of that type. This feature is highly machine-dependent and should be used sparingly.

1.6.2 Min

```
scalar-var := min(scalar-expr[,scalar-exp]...)
```

The *min* function returns the smallest of its scalar arguments. For example, *min*(-6, 3, 5) returns -6.

1.6.3 Max

```
scalar-var := max(scalar-expr[,scalar-expr]...)
```

The *max* function returns the largest of its scalar arguments. For example, *max*(-2, 3, 5) returns 5.

1.6.4 Lbound

```
scalar-var := lbound(array-type[,index])
```

The *lbound* function returns the lower bound of the array type specified by the first argument. The array type must be specified by a type identifier or a variable whose type is array. If the array is multi-dimensional, then an optional second argument specifies the dimension. The default is 1, which specifies the first or outermost dimension.

1.6.5 Hbound

```
scalar-var := hbound(array-type[,index])
```

The *hbound* function returns the upper bound of the array type specified by the first argument. The array type must be specified by a type identifier or a variable whose type is array. If the array is multi-dimensional, then an optional second argument specifies the dimension. The default is 1, which specifies the first or outermost dimension.

1.6.6 First

```
scalar-var := first(type-identifier)
```

The *first* function returns the first (lowest) value of the named scalar type. For example, *first*(*integer*) returns -2147483848.

You can use *first* to find the bottom end of a range, or other scalar data type. Scalar types include integers, ranges, boolean, characters and enumerations.

1.6.7 Last

```
scalar-var := last(type-identifier)
```

The *last* function returns the last (highest) value of the named scalar type. For example, *last*(*integer*) returns 2147483847 (*maxint*). Use it to find the top end of a range.

1.6.8 Sizeof

```
sizeof(type-id[,tagfield-value]...)
```

The *sizeof* function returns the number of bytes occupied by the data type specified as an argument. If the argument specifies a record with a variant record part, then additional arguments specify the value of each tagfield.

1.6.9 Argc

```
integer-var := argc
```

The *argc* function returns the number of arguments passed to the program. The value of the function is 1 or greater.

1.6.10 Bitand

```
integer-var := bitand(integer-expr, integer-expr)
```

The *bitand* function returns the bitwise AND of the two integer-valued operands.

1.6.11 Bitor

```
integer-var := bitor(integer-expr, integer-expr)
```

The *bitor* function returns the bitwise OR of the two integer-valued operands.

1.6.12 Bitxor

```
integer-var := bitxor(integer-expr,integer-expr)
```

The *bitxor* function returns the bitwise exclusive-OR of the two integer-valued operands.

1.6.13 Bitnot

```
integer-var := bitnot(integer-expr, integer-expr)
```

The *bitnot* function returns the bitwise NOT of the two integer-valued operands.

1.6.14 Clock

```
integer-var := clockf1
```

The *clock* function returns the number of milliseconds of processor time used by the current process.

1.6.15 Lshift

```
integer-var := lshift(integer-expr, integer-expr)
```

The *lshift* function returns the left shift of the first integer-valued operand by the amount specified in the second argument. Zeros are inserted on the right.

1.6.16 Rshift

```
integer-var := rshift(integer-expr, integer-expr)
```

The *rshift* function returns the right shift of the first integer-valued operand by the amount specified in the second argument. Sign bits are inserted on the left if the operand is an integer type; zero bits are inserted if the operand is a cardinal type. You can force zero bits with the following construct:

```
rshift(cardinal(intexpr), shiftamount)
```

1.6.17 Firstof

The *firstof* function is equivalent to the *first* function.

1.6.18 Lastof

The *lastof* function is equivalent to the *last* function.

1.6.19 Addr

The *addr* function returns a pointer to the variable or function argument. The argument can be a string constant but cannot be another other type of constant. The argument can not be a nested or internal procedure, or a variable.

1.6.20 Bitsize

The *bitsize* function returns the size in bits of the data type indicated by the argument.

1.7 I/O Extensions

There are six I/O extensions.

1.7.1 Specifying Radix in the Write Statement

Pascal permits the specification of operands in a *write* statement that writes numbers in any radix from 2 through 36. The following code writes a number in each radix:

```
for i := 2 to 36 do
  writeln('x is',x:1:i,' in radix',i:1);
```

A minus sign precedes any printed number if the type of the number is integer and the value is less than zero. If the type of the number is cardinal, then the compiler interprets the number as not having a sign and prints the sign bit as part of the number (rather than with a negative sign).

1.7.2 Filename on Rewrite and Reset

SGI Pascal accepts an optional string argument that specifies the path name of the file to be opened or created. Otherwise, SGI Pascal creates a temporary file that exists only during program execution, in the directory */tmp*.

```
reset(inputfile [,string])
rewrite(outputfile [,string])
```

1.7.3 Reading Character Strings

SGI Pascal allows you to read characters into a character string array, while ANSI Pascal does not. A string is defined to be a packed array of *char* whose lower bond is 1 and whose upper bond is greater than 1.

In the following example, characters are read into the array one line at a time until the end-of-file (eof) is reached.

```
program CountLines(input, output);
type
    string80 = packed array[1..80] of char;
var
    Line : string80;
 begin
    .
    .
   while not eof(input) do begin
          readln(input, Line)
       i := i+1;
   end; {while}
   write ('The number of lines is ', I:1);f1
```

1.7.4 Reading and Writing Enumeration Types

SGI Pascal provides an extension to permit any enumerated scalar type to be specified as the operand of a *read* or a *write* to a text file.

Reading an enumeration value interprets the programmer-defined name, preceded optionally by blank, tab, or new-line characters. The end of the name is delimited by any character that is not a valid character of an identifier. The delimiting character is skipped. Good choices for delimiting characters are blanks, tabs, commas, or new-lines.

Writing an enumerated value causes the programmer defined name to be written out to the text file.

The following piece of code is an example of using enumerated values:

```
program testenum(input, output);
type
color = (red, orange, yellow, green, blue, violet,
           black, white);
var
   vcolor : color;
begin
   repeat
       writeln('enter color');
       read(vcolor);
       writeln('The color is ', vcolor : 0);
   until eof;
end.
```

If any color other than the one specified in the *color* enumeration is entered, the following message is displayed:

```
enumerated value string not within type
          The color is red
```

where *string* represents the incorrect value entered.

When an incorrect value is entered, the first value in the enumeration (*red* in the example above) is written to the screen.

1.7.5 Lazy I/O

SGI Pascal provides an extension to ANSI Pascal I/O that simplifies terminal-oriented I/O. Standard Pascal defines the file pointer of a text file to point to the first character of a line after a *reset* or *readln* operation. This makes it difficult to issue a prompt message because the physical I/O operation for the next line occurs at the end of the *readln* procedure.

SGI Pascal follows ANSI Pascal conventions, except that it does not perform physical I/O until the program user actually uses the file pointer.

In effect, it is lazy about performing the I/O operation. This allows the user to issue a prompt message after the *readln* (or reset) prior to the time when the user's terminal attempts to read the next line.

1.7.6 Standard Error

SGI Pascal provides an additional predefined text file called *err*, which is mapped to IRIX standard error. Also, the files *input* and *output* are mapped to the IRIX file standard input and standard output.

1.8 Predefined Data Type Extensions

There are six predefined data type extensions.

1.8.1 Cardinal

SGI Pascal accepts the *cardinal* data type that represents an unsigned integer in the range 0..4294967295 ($2^{32}-1$). If either operand of an expression is cardinal, then the compiler uses unsigned arithmetic.

1.8.2 Double

SGI Pascal accepts a new predefined data type called *double* that represents a double-precision floating point number. If either operand of an expression is double, then the compiler uses double-precision.

1.8.3 Pointer

SGI Pascal defines a new predefined data type called *pointer* that is compatible with any pointer type. This can be thought of as the type of the Pascal value *nil*. Use *pointer* to write procedures and functions that accept an arbitrary pointer type as an operand.

Note that you cannot directly de-reference a variable of this type because it is not bound to a base type. To de-reference it, you must first assign the variable to a normally typed pointer.

Procedure and Function Pointers

You can use the *addr* function to assign a value to a variable defined to be a procedure or function pointer. This is useful in passing the address of a procedure or function as a parameter. For example:

```
var proc_ptr : ^procedure (i:integer);
func_ptr : ^function (a,b:integer) : real;
```

1.8.4 String

This new SGI Pascal predefined array type is defined as

```
type string = array[1..80] of char;
```

1.8.5 INTEGER16

A variable defined as *INTEGER16* represents a signed 16-bit integer, and occupies a halfword.

1.8.6 INTEGER32

A variable defined as *INTEGER32* represents a signed 32-bit integer, and occupies a full word.

1.9 Predefined Data Type Attributes

SGI Pascal supports two new data type attributes: *static* and *volatile*.

1.9.1 static

The compiler will allocate space for a static variable and keep the name local to the procedure. The variable retains its value from one invocation of the procedure to the next. In the following example, *k* is a static integer.

```
var first : boolean := true;
j : integer;
function show_static: integer;
var k : static integer;
i : integer;
begin
    if first then begin
       k := 9;
       first := false;
    end else
    k := k + 3;
    show_static := k;
end;
program main;
begin
    for j := 1 to 2 do
        writeln('Show_Static ;+ ',show_static:3);
end.
```

1.9.2 volatile

The compiler will not perform certain optimizations for a volatile variable. This is useful for retaining pointer references that might appear to the optimizer to be redundant.

1.9.3 Static Arrays

SGI Pascal permits you to initialize a *static* variable of type *array* (single or multi-dimensional). For example:

```
type foo = (one,two);
var x : array [foo,foo] of foo := [[one,two], [two,one]];
```

1.9.4 Static Variables of Type Record

SGI Pascal permits you to initialize *static* variables of type *record*. For example:

```
type rec = record;
field1 : integer;
field2 : char;
end

var rec1 : rec := [1,'a'];
rec2 : rec := [Field2 := 'M' ; Field1 := 3]'
```

1.9.5 Packed Records

In packed records, fields of subrange data types based on integers and enumerated types are assumed by default to have word alignment. Thus, in allocating the field, the compiler skips bits as necessary to avoid crossing a word boundary. In extracting the field, the compiler always loads words. The base type of a subrange can now be specified using the *of* keyword, as in the following example:

```
i: 0..32767 of INTEGER16;
```

This causes the field *i* to have halfword alignment. Similarly,

```
i: 0..255 of char;
```

causes the field *i* to have the alignment of the data type char, that is, byte alignment.

1.10 Parameter Extensions

1.10.1 Univ

The compiler will not perform type checking for a parameter with the universal attribute Univ. The compiler will still perform size checking. For example:

```
var y : real;
procedure x (j: univ integer);
…
x(y);
```

1.10.2 Conformant Array Parameters

Conformant array parameters allow array type parameters of functions and procedures to have variable dimensions. You can use any array that conforms to the parameter definition as an actual parameter. For example:

```
procedure print (msg : array[min..max: integer] of char);
var i : integer;
begin
    for i := min to max do
        write(msg[i]);
    writeln;
end;
var
    large : array [-10..10] of char
    small : array [2..7] of char;

begin
    … {Load some values into the arrays}
    print(large);
    print(small);
end.
```

1.10.3 In, Out, and In Out

These keywords specify the direction of parameter passing between routines. Parameters declared as type *in* can not be changed within the scope of the routine. Parameters declared as *out* are not expected to have a value when they enter the procedure, but are expected to have a value when they leave it. Flagging a parameter as *in out* tells the compiler that it contains a value and that the procedure is permitted to modify that value.

1.11 Compiler Notes

These notes describe extensions that affect the compile process.

1.11.1 Macro Preprocessor

SGI Pascal invokes the C preprocessor (*cpp*) before each compilation, allowing you to use *cpp* syntax in the program. The *cpp* variables *LANGUAGE_PASCAL*, *LANGUAGE_C*, *LANGUAGE_FORTRAN*, and *LANGUAGE_ASSEMBLY* are defined automatically, allowing you to build header files that can be used by different languages.

The following example shows two conditional statements, one written in Pascal and the other written in C.

```
#ifdef LANGUAGE_PASCAL
type
    pair =
        record
             high, low : integer;
        end; {record}
#end

#ifdef LANGUAGE_C
typedef struct {
    int high, low;
    } pair
#end
```

You can also use the full conditional expression syntax of *cpp*, as well as C style comments (/* ...*/), which are stripped during compilation.

1.11.2 Short Circuiting

SGI Pascal always short circuits boolean expressions. Short circuiting is a technique where only a portion of a boolean expression is actually executed.

For example, in this code:

```
if (P<>nil) and (P^.Count > 0) then
```

the expression involving *P^.Count* is not evaluated if the first expression is false. This extension is permitted by ANSI Pascal. A program that relies on this feature, as does this example, would not be portable.

1.11.3 Translation Limits

The following table shows the maximum limits imposed on certain items by the Pascal compiler. Chapter 3 discusses set sizing rules in greater detail; see Chapter 3 for more information.

Pascal Specification	Maximum
Literal string length	288
Procedure nesting levels	20
Set size	451 - 512 (see Section 3.4 for rules)
Significant characters	32

Table 1-3 Maximum Limits of Data Items

1.11.4 The -apc Option

This option enables some of the Apollo Pascal extensions that are implemented in the SGI Pascal compiler. This option must be passed to the SGI Pascal compiler by using *-Wf,-apc*.

These extensions include:

- *IN_RANGE*: This built-in function determines whether a specified scalar variable is within the defined range of an integer subrange or enumerated type.

```
var i : -7..7;
begin
write(' enter i : ');
readln(i);
if (in_range(i)) then
....
```

- *DISCARD*: This procedure explicitly discards an expression value, which could include a value returned by a function call.
- Non-standard *MOD* operator: Under this flag the *MOD* operator returns a negative result when the dividend is negative.
- *DEFINE*: This attribute tells the compiler to allocate the variable in the static data area and to make its name accessible externally.

```
var j : define integer := 9;
```

- *EXTERN*: This attribute tells the compiler not to allocate a space for the variable since it may be allocated in a separate compiled procedure.
- infix bit operators: These bitwise operators are:
  - & equivalent to *BITAND*
  - ! equivalent to *BITOR*
  - ~ equivalent to *BITNOT*

  Examples:

```
var i,j : integer;
 .  .  .
(i & j)
(i ! j )
(~ i)
```

- Type coercion: *TYPE*(*var*). Type coercions are a bitwise transfer from a variable of one data type to a variable of another data type.

```
var i : integer;
x : real;
begin  i := 456;
x := real(i);
```

The variable *x* now contains the same bits that *i* does, not the same number value. Note that the SGI Pascal compiler currently implements type conversion using the above syntax. Under the *-apc* flag the compiler interprets this as a type coercion rather than type conversion.

- <*type-id*> pseudo function: SGI Pascal permits the use of the transfer function on the left hand side of an assignment statement.

  ```
  real(i) := x + y;
  ```

  The above means that the assignment is done in floating point mode and that the bits of *i* contain the same number as *x+y*.

- Constant strings may be assigned to non-packed arrays of char.
- Correctly aligned objects within a packed array are passed as *VAR* parameters.
- implicit null otherwise: If no *CASE* is taken and *OTHERWISE* is left undefined, the *CASE* statement has no effect and executes without an error.
- array initialization [*N* of *C*]: This form tells the compiler to initialize *N* elements of the array to the value *C*.
- Wild card array initialization: When initializing an array in the *var* part of a routine, the compiler can compute the first dimension of the array.

  ```
  var arr: array[1..*] of char := 'This is a test';
  ```

- Under the *-apc* flag *integer* is equivalent to *INTEGER16* and occupies two bytes of storage. Therefore, *MAXINT* = 32767.
- Real constants are permitted to be of the form “123.”

1.11.5 The -casesense Option

Assert case sensitivity on variable names. The default is case insensitive, that is, variable names *ABC* and *abc* are considered to be the same identifier.

Chapter 2

Compiling, Linking, and Running Pascal Programs

This chapter lists specific commands for compiling, linking, and running Pascal programs on the IRIS-4D. It also explains the tools used in each command procedure.

You can find general information about compiling, linking, and running programs on the IRIS-4D in the *IRIS-4D Series Compiler Guide*.

2.1 Compiling and Linking Programs

Compile and link programs using the Pascal driver command *pc* as described in this section.

This is the format of the *pc* command:

```
pc [options] filename
```

These are the parts of the command and their tasks:

- *pc* invokes the Pascal driver processes that compile, optimize, assemble, and link edit the program. (To compile programs without linking, see Section 2.1.2, "Pascal Driver Options" below.)
- *option* represents a driver option. The number of options allowed on a command line by the compiler varies according to the task.

- *filename* may include Pascal source code, object code, and libraries:
  - *filename.p* for Pascal source code
  - *filename.o* for object code
  - *-l libraryname* for libraries; libraries follow files

The output of the *pc* command is a program executable module.

2.1.1 The Pascal Driver

The Pascal driver invoked by the *pc* command is an intelligent program that in turn invokes the five major components of the compiler system.

These are the five components, listed below in processing order:

1. Pascal compiler
2. optimizer
3. code generator
4. assembler
5. link editor

Figure 2-1 illustrates the compile process.

In this illustration, the primary driver phases are in the boxes. The principal input is the source file *more.p*. The principal output is the executable file *a.out*. The file name *a.out* is the default name for executables.

See Section 2.1.2 section below to rename the executable file during the compile process.

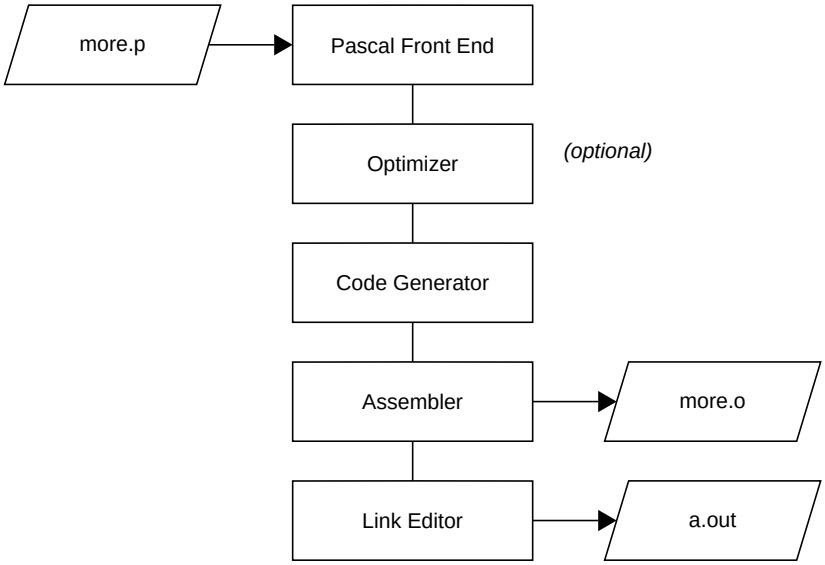


Figure 2-1 Pascal Compile Process

2.1.2 Pascal Driver Options

Three driver output options are listed in this section according to the task each one performs. Additional options are discussed briefly by category. The complete set of options is listed in reference format in the *pc*(1) man page.

Produce a Linkable Object File

Use the *-c* driver option to produce a linkable object file instead of an executable. This option stops the driver immediately after the assembler phase.

The linkable object file has the suffix *.o.*

Here is an example of the driver command with this option:

```
pc -c more.p
```

The object file produced is *more.o*.

Note: When you want to place more than one object file on the command line, specify the file containing the main program first. The link editor requires this information.

Rename the Executable

Use the *-o* option to rename the executable during the compile process. If this option is not used, the executable will be named *a.out*.

You can use this option with a source file or an object module.

Here is an example command line that takes the source file *myprog.p*, compiles it, links it, and names the executable *exec.myprog*:

```
pc myprog.p -o exec.myprog
```

Here is an example command that takes the object module *more.o*, links it, and names the executable *exec.myprog*:

```
pc more.o -o exec.myprog
```

Graphics Option

If your program contains graphics routines from the Graphics Library, you need to use the graphics option. The graphics option, *-Zg*, automatically links to your program two libraries and a conversion file: *libpgl.a*, *libgl.a*, and *p2cstr.o*.

If you do not use the *-Zg* option, you can explicitly link the necessary routines as follows:

```
pc sourcefile.p -lpgl -lgl /usr/lib/p2cstr.o
```

Debugging Options

When you are debugging, use the *-g* option with the output option discussed above. The *-g* option creates an expanded symbol table in the object file and links the file using the debugging flag. An example of compiling for debugging is:

```
pc -g sourcefile.p -o executefile
```

There are other forms of the *-g* option, plus different options for profiling (*-pn*), optimizing (*-On*), and other specialized driver functions. These are discussed and listed in the release notes for your version of Pascal.

2.1.3 Compiling Multi-Language Programs

The compiler system provides drivers for other languages in addition to Pascal, including C (standard), Fortran 77 (optional), and PL/1 (optional).

This section describes general multi-language compiling conventions.

When your application has two or more source programs written in different languages, you should compile each program separately with the appropriate driver, and then separately link them in another step. Use the *-c* option to create objects suitable for link editing.

Driver Command Syntax

For each language below, see the man page listed with it for correct driver command syntax:

```
C              cc(1)
Fortran 77     f77(1)
PL/1           pl1(1)
Pascal         pc(1)
```

Here is a command example with a C program and a Pascal program:

```
cc -c main.c
pc -c rest.p
```

main.c contains the main routine and is executed first, as required. Then the Pascal source file *rest.p* is compiled.

The compile processes and output for this example are shown in Figure 2-2.

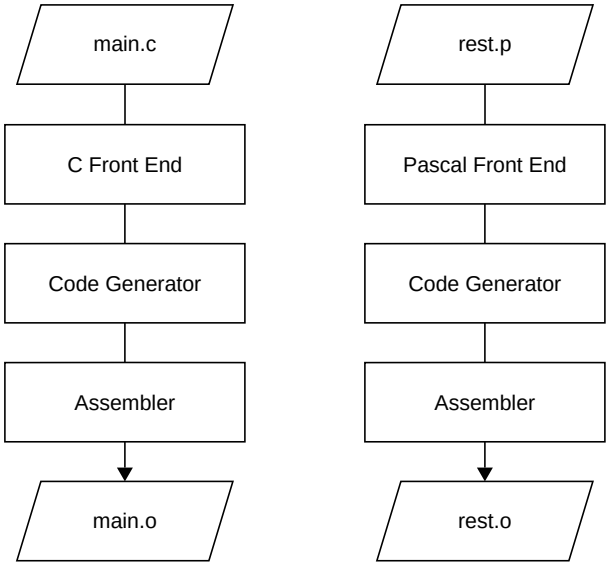


Figure 2-2 Multi-Language Compile Process

2.1.4 Linking Separate Language Objects

The driver recognizes the *.o* suffix as the name of a file containing object code suitable for link editing, and immediately invokes the link editor.

Here is an example command that link edits the object created in the last example above:

```
pc -o all main.o rest.o
```

The command produces the executable program object *all*. You can achieve the same results using the *cc* driver command with the additional option *-l*, as shown below:

```
cc -o all main.o rest.o -lp -lm
```

Both *f77* and *cc* use the C link library by default. However, the *cc* driver command does not know the names of the link libraries required by Pascal

objects. Therefore, you must specify them explicitly to the link editor, using the *-l* option with the symbols for the other libraries.

The *-l* option with *p* links the Pascal library */usr/lib/libp.a*.

The *-l* option with *m* links the Math library */usr/lib/libm.a*.

See the *FILES* section of the Pascal *pc*(1) manual pages, for a complete list of files available. See the *ld* man page for additional information about the *-l* option.

2.1.5 Making Inter-Language Calls

In addition to compiling separate language source files into one executable, you can make inter-language calls between C, Pascal, and Fortran 77.

An interface between C and Pascal is described in Chapter 4 of this manual. An interface between Pascal and Fortran 77 is described in the *Fortran 77 Programmer's Guide*.

2.2 Running Programs

Run Pascal programs using the name of the executable object module produced by the *pc* command with the *-o* option.

If you do not rename the program during the compile process, it is named *a.out* by default. Invoke it with this command in the directory where the executable resides:

```
a.out
```

If you do rename the program during the compile process, invoke it with its assigned name in the directory where the executable resides.

The *IRIS-4D Series Compiler Guide* contains more information about run time considerations.

Chapter 3

Storage Mapping

Storage mapping of Pascal arrays, records, and variant records is described in the first part of this chapter. Ranges are discussed in the second part. Alignment, size and value ranges for the various data types are described in the third part. The last section discusses rules for set sizing.

3.1 Arrays, Records, and Variant Records

Pascal maps arrays and records into storage like C maps arrays and structures.

3.1.1 Arrays

An array has the same boundary requirements as the data type specified for the array. The size of an array is the size of the data type, multiplied by the number of elements.

For example, for the following declaration,

```
x : array [1..2, 1..3] of double;
```

the size of the resulting array is 48 bytes (2*3*8, where 8 is the size of the double-floating point type in bytes).

3.1.2 Records

Each member of a record begins at an offset from the record base. The offset corresponds to the order in which a member is declared; the first member is at offset 0.

The size of a record in the object file is the size of its combined members plus padding added, where necessary, by the compiler.

The following rules apply to records:

- Records must align on the same boundary as that required by the member with the most restrictive boundary requirement. These are the boundary requirements, listed by increasing degree of restrictiveness:
  - byte
  - halfword
  - doubleword
- The compiler terminates the record on the same alignment boundary on which it begins. For example, if a record begins on an even-byte boundary, it also ends on an even-byte boundary.

Example: Record

This example shows a record in code:

```
type S = record
     v : integer;
     n : array [1..10] of char;
end;
```

Figure 3-1 shows how it is mapped in storage.

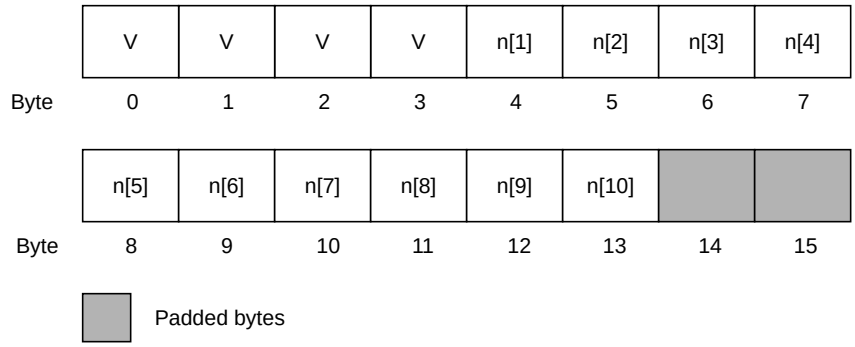


Figure 3-1 Record in Storage, End Padded

Note that the length of the record is 16 bytes, even though the byte count as defined by the *v:integer* and the *n:array[1..10] of char* components is only 14. Because *integer* has a stricter boundary requirement (word boundary) than *char* (byte boundary), the record must end on a word boundary (a byte offset divisible by four). The compiler therefore adds two bytes of padding to meet this requirement.

An array of data records illustrates the reason for this requirement. For example, if the above record were the element-type of an array, some of the *v:integer* components would not be aligned properly without the two-byte pad.

Example: Record with Different Alignment

This example shows a record with different alignment in code:

```
type S = record
     n : packed array [1..10] of char;
     v : integer;
end;
```

Figure 3-2 shows how it is mapped in storage.

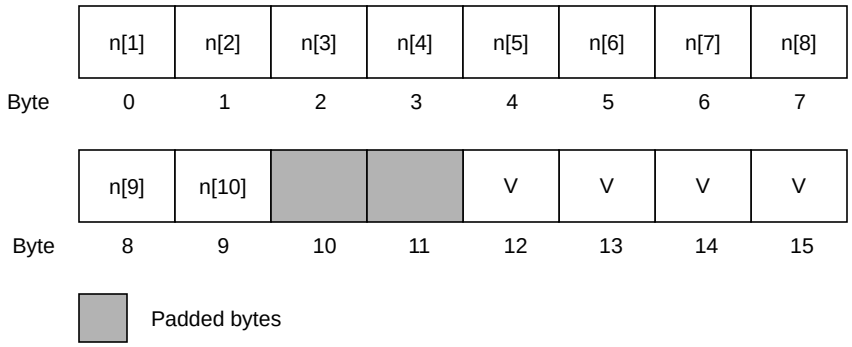


Figure 3-2 Record in Storage, Middle Padding

In the example, the alignment requirements cause padding to appear in the middle of the record. Note that the size of the record remains 16 bytes, but two bytes of padding follow the *n* component to align *v* on a word boundary.

3.1.3 Variant Records

A variant record must align on the same boundary as the member with the most restrictive boundary requirement.

These are the boundary requirements, listed by increasing degree of restrictiveness:

- byte
- halfword
- word
- doubleword

For example, a variant record containing *integer*, *char*, and *double* data types must align on a doubleword boundary, as required by the *double* data type.

3.2 Ranges

Ranges in a packed record are packed from the most-significant bit to the least-significant bit in a word.

3.2.1 Ranges in a Packed Record

This example shows a packed record in code:

```
type virtual_address = packed record
    offset : 0..4095;              (* 12 bits *)
    page : 0..1023;                (* 10 bits *)
    segment : 0..511;              (* 9 bits *)
    supervisor : 0..1;             (* 1 bit *)
end;
```

Figure 3-3 shows how it is mapped in storage.

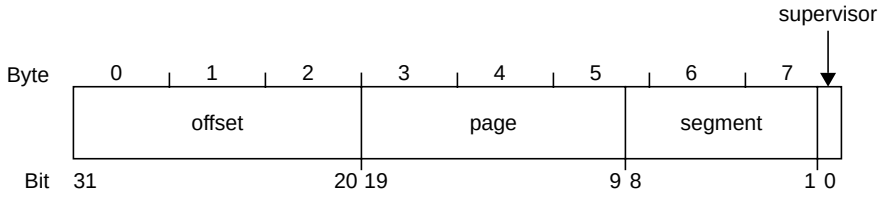


Figure 3-3 Ranges in a Packed Record

3.2.2 Ranges in an Unpacked Record

Ranges in an unpacked record are packed from the most-significant bit to the least-significant bit, but each range is aligned to the appropriate boundary. This example shows an unpacked record in code:

```
type virtual_address = record
    offset : 0..4095;        (* 12 bits *)
    page : 0..1023;          (* 10 bits *)
    segment : 0..511;        (* 9 bits *)
    supervisor : 0..1;       (* 1 bit *)
end;
```

Figure 3-4 shows how it is mapped in storage.

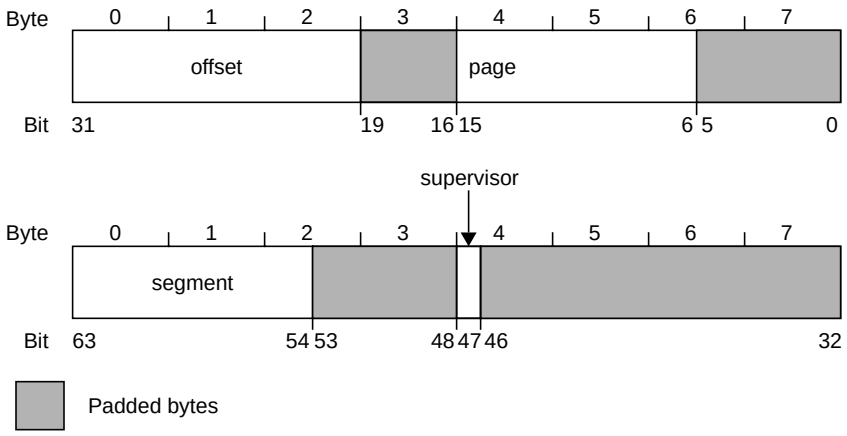


Figure 3-4 Ranges in an Unpacked Record

3.2.3 Non-Ranges Following Ranges in Unpacked Records

For unpacked records, the compiler aligns a non-range element that follows a range declaration to the next boundary appropriate for its type.

This example shows another unpacked record in code:

```
var x : record
    a : 0..7;                (* 3 bits packed *)
    b : char;                (* 8 bits *)
    c : -32768..32767;       (* 16 bits *)
end;
```

Figure 3-5 shows how it is mapped in storage.

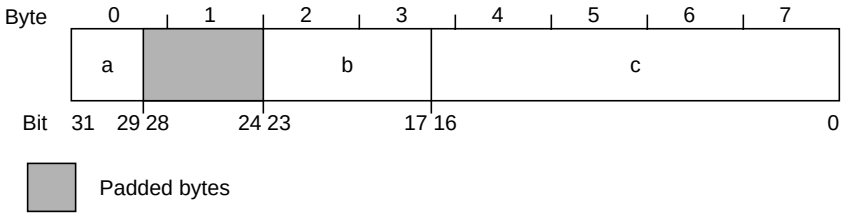


Figure 3-5 Non-Range Elements in an Unpacked Record

3.2.4 Non-Range Elements in a Packed Record

For a packed record, the computer bit-aligns *booleans*, *chars*, and *ranges*. All other types are word or double-word aligned as appropriate for the type. This example shows the same code as above in a packed record and its mapping in storage:

```
var x : packed record
    a : 0..7;            (* 3 bits *)
    b : char;            (* 8 bits *)
    c : -32768..32767    (* 16 bits *)
end;
```

Figure 3-6 shows how it is mapped in storage.

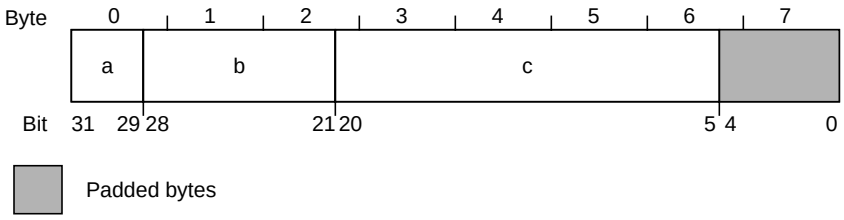


Figure 3-6 Non-Range Elements in a Packed Record

3.3 Alignment, Size, and Value by Data Type

This section describes how the Pascal compiler implements size, alignment, and value ranges for the various data types.

Table 3-1 shows the value ranges for the Pascal scalar types.

Scalar Type	Value Ranges
boolean	0 or 1
char	0..127
integer	$-2^{31}..2^{31}-1$
cardinal	$0..2^{32}-1$
real	See Note
double	See Note

Table 3-1 Value Ranges by Data Type

Note: See Table 3-2 and Table 3-3 for more data.

Table 3-2 and Table 3-3 show the approximate valid ranges for *real* and *double*. Enumerated types with n elements are treated in the same manner as the integer subrange 0..*n*-1.

Type	Maximum Value
real	$3.40282356*10^{38}$
double	$1.7976931348623158*10^{308}$

Table 3-2 Real and Double Maximum Values

	Minimum Value	
Type	Denormalized	Normalized
real	$1.40129846 * 10^{-46}$	$1.17\ 549429 * 10^{-38}$
double	$4.9406564584124654*10^{-324}$	$2.2250738585072012*10^{-308}$

Table 3-3 Real and Double Minimum Values

3.3.1 Set Sizing for Unpacked Records

Table 3-4, following, lists size and alignment parameters for unpacked records. See Section 3.4, "Rules for Set Sizes," for rules about specifying the upper and lower bounds of sets.

	Unpacked Records or Arrays (variables or fields)	
Type	**Size**	**Alignment**
boolean	8	byte
char	8	byte
integer	32	word
cardinal	32	word
pointer	32	word
file	32	word
real	32	word
double	64	doubleword
subrange of:		
0..255 or -128..127	8	byte
0..65535 or -32768..32767	16	halfword
$0..2^{32} - 1$		
$-231..-2^{31} - 1$	32	word
set of char	128	word
set of a..b	See note	word

Table 3-4 Size and Alignment of Data Types in Unpacked Records or Arrays

Note: For unpacked records, the compiler uses the following formula for determining the size of the *set of a..b:*

$$size = \lfloor b/32 \rfloor - \lfloor a/32 \rfloor + 1 \text{ words}$$

The notation $\lfloor x \rfloor$ indicates the floor of x, which is the largest integer not greater than x.

3.3.2 Set Sizing for Packed Arrays by Type

The compiler uses the rules shown in Table 3-5 for aligning packed arrays.

	Packed Arrays	
Scalar Type	**Size**	**Alignment**
boolean	8	byte
char	8	byte
integer	32	word
cardinal	32	word
pointer	32	word
file	32	word
real	32	word
double	64	doubleword
subrange of:		
0..1 or -1..0	1	bit
0..3 or -2..1	2	2-bit
0..15 or -8..7	4	4-bit
0..255 or -128..127	8	byte
0..65535 or -32768..32767	16	halfword
$0..2^{32} - 1$		
$-231..-2^{31} - 1$	32	word
set of char	128	word
set of a..b	See note	

Table 3-5 Size and Alignment of Pascal Packed Arrays

Note: For packed arrays, the compiler uses the minimum number of bits possible in creating the *set of a..b*.

The following formula is used:

If
$(b - 32\lfloor a/32 \rfloor + 1) \leq 32$
then
$size = b - 32\lfloor a/32 \rfloor + 1$ bits
else
$size = \lfloor b/32 \rfloor - \lfloor a/32 \rfloor + 1$ words

Note that the *set of a..b* is aligned on an *n*-bit boundary where *n* is a power of 2. The value of n is computed as follows:

$n = 2^{\lceil \log2(size) \rceil}$

For example, the *set of 0..2* has a size of 3 bits as computed above and will align on a 4-bit boundary.

See Section 3.4 at the end of this chapter for rules about specifying the upper and lower bounds of sets.

3.3.3 Packed Record Alignment

The compiler uses the rules shown in Table 3-6 for aligning packed records.

	Packed Arrays	
Scalar Type	**Size**	**Alignment**
boolean	8	byte
char	8	byte
integer	32	word
cardinal	32	word
pointer	32	word
file	32	word
real	32	word
double	64	doubleword
subrange of a..b	See note	bit/word

Table 3-6 Size and Alignment of Pascal Packed Records

Note: For packed records, the compiler uses the minimum number of bits possible in creating a subrange field.

For the subrange *a..b*, this formula is used:

```
If  a >= 0  then  size =⌈log2(b + 1) ⌉ bits
If a > 0    then  size = max(⌈log2(b + 1)⌉,⌈log2(-a ) ⌉)+1  bits
```

The notation ⌈ x ⌉ indicates the ceiling of *x*, which is the smallest integer not less than *x*.

To avoid crossing a word boundary, the compiler moves data types aligned to bit boundaries in a packed record to the next word.

3.4 Rules for Set Sizes

The maximum number of elements permitted in a set ranges between 481 and 512. This variance is due to the way Pascal implements sets. For efficient accessing of set elements, Pascal expects the lower-bound of a set to be a multiple of 32. If for the set specified:

```
set of a..b
```

a is not a multiple of 32, Pascal adds elements to the set from a down to the next multiple of 32 less than *a*.

For example, the set:

```
set of 5..31
```

would have internal padding elements *0..4* added. These padding elements are inaccessible to the program. This implementation sacrifices some space for a fast, consistent method of accessing set elements.

The padding required to pad the lower bound down to a multiple of 32 varies between 0 and 31 elements.

For the *set of a..b* to be a valid set in Pascal, the following conditions must be met:

```
size = ( b – 32⌊a / 32⌋+ 1 ) ≤ 512
```

Table 3-7 shows some example sets and whether each set is valid by the above equation.

Specification	Lower	Upper	Set Size	Valid Size
set of 1..511	0 (padded down to value by Pascal)	511	512	Yes
set of 0..511	0	511	512	Yes
set of 1..512	0*	512	513	No
set of 31..512	0*	512	513	No
set of 32..512	32	512	481	Yes
set of 32..543	32	543	512	Yes

Table 3-7 Set Specifications

Chapter 4

Pascal/C Interface

This chapter describes the coding interface between Pascal and C. The *Fortran 77 Programmer's Guide* describes the Pascal/Fortran interface.

Section 4.1, "Guidelines for Using the Interface," makes some comparisons between Pascal and C, and lists guidelines for dealing with the differences noted.

Section 4.2, "Calling Pascal from C," lists rules and examples for calling Pascal from C.

Section 4.3, "Calling C from Pascal," lists rules and examples for calling C from Pascal.

4.1 Guidelines for Using the Interface

In general, calling C from Pascal and Pascal from C is fairly simple. Most data types in one language have natural counterparts in the other.

Differences exist in eight areas. Section 4.1.1 through Section 4.1.10 describe these differences with some guidelines.

4.1.1 Single-precision Floating Point

In function calls, C automatically converts single-precision floating point values to double-precision. Pascal passes single-precision floating by-value arguments directly.

Follow these guidelines for passing double-precision values between a C routine and a Pascal routine:

- If possible, write the Pascal routine so that it receives and returns double-precision values.
- If the Pascal routine cannot receive a double-precision value, write a Pascal routine to accept double-precision values from C. Then have that routine call the single-precision Pascal routine.

The compiler has no problem passing single-precision values by reference between C and Pascal.

4.1.2 Procedure and Function Parameters

C function variables and parameters consist of a single pointer to machine code.

Pascal procedure and function parameters consist of a pointer to the machine code, and a pointer to the stack frame of the lexical parent of the function.

Such values can be declared as structures in C. To create such a structure, put the C function pointer in the first word and 0 in the second. C functions cannot be nested, and have no lexical parent; therefore, the second word is irrelevant.

You cannot call a C routine with a function parameter from Pascal.

4.1.3 Pascal By-Value Arrays

C never passes arrays by value. In C, an array is actually a sort of pointer, and so passing an array actually passes its address, which corresponds to Pascal by-reference (VAR) array passing.

In practice, this is not a serious problem because passing Pascal arrays by value is not efficient, and so most Pascal array parameters are VAR anyway. When it is necessary to call a Pascal routine with a by-value array parameter from C, pass a C structure containing the corresponding array declaration.

Check your copy of the *Pascal Release Notes* for detailed instructions to pass packed array integer subranges by value.

4.1.4 File Variables

The Pascal text type and the C *stdio* package *FILE** are compatible. However, Pascal passes file variables only by reference. A Pascal routine cannot pass a file variable by value to a C routine. C routines that pass files to Pascal routines should pass the address of the *FILE** variable, as with any reference parameter.

4.1.5 Passing String Data Between C and Pascal

C and Pascal handle strings differently. Pascal defines a string as a packed array of characters, where the lower bound of the array is 1 and the upper bound is an integer greater than 1. C indexes arrays from 0 to *max*-1. For example, Pascal string parameters are typically declared so that the upper bound is large enough to efficiently handle most processing requirements:

```
var s:packed array[1..100] of char
```

In passing an array, Pascal passes the entire array as specified, padding to the end of the array with spaces. Most C programs treat strings as pointers to a single character and use pointer arithmetic to step through the string. A null character (\0 in C) terminates a string in C.

The following example shows a Pascal routine that calls the C routine *strtol* and passes the string *s*. Note that the routine ensures that the string terminates with a null character.

```
type
   astrindex = 1..20;
   astrindex = packed array [astrindex] of char;
   function atoi(var c: astring): integer; external;
program ptest(output);
   var
      s: astring;
      i: astrindex;
   begin
   argv(1, s); {This predefined Pascal function is a MIPS
extension}
   writeln(output, s);
   {  Guarantee that the string is null-terminated (but might
      bash the last character if the argument is too long).
      "lbound" and "hbound" are MIPS extensions. }
   s[hbound(s)] := chr(0);
   for i := lbound(s) to hbound(s) do
       if s[i] = ' ' then
       begin
          s[i] := chr(0);
          break;
       end;
   writeln (output, atoi(s));
   end.
```

For more information about *strtol*, see the *strtol*(3c) man page.

4.1.6 Graphics Strings

Using Graphics Library routines requires conversion for Pascal format strings. You will need to convert Pascal strings to C formats. The procedure file */usr/lib/p2cstr.o* will perform this conversion for you. If you compile your program with the *-Zg* option, this conversion will be done automatically.

The compile option *-Zg* will link *p2cstr.o* to perform the conversion. If you do not use the *-Zg* option, you must explicitly link *p2cstr.o*. See Section 2.1.2, "Pascal Driver Options," for details.

4.1.7 Passing Variable Arguments

You can define C functions that take a variable number of arguments (*printf* and its variants are examples). You cannot call such functions from Pascal.

4.1.8 Passing Pointers

A Pascal routine cannot pass a function pointer to a C routine.

You can pass a C pointer to a function to Pascal as a structure by value. The first word of the structure must contain the function pointer and the second word must contain a zero. Pascal requires this format because it expects an environment specification in the second word.

4.1.9 Type Checking

Pascal checks certain variables for errors at execution time, whereas C does not. For example, in a Pascal program, when a reference to an array exceeds its bounds, the error is flagged (if run time checks are not suppressed). You can not expect a C program to detect similar errors when you pass data to it from a Pascal program.

4.1.10 One Main Routine

Only one main routine is allowed per program. You can write the main routine in either Pascal or C. Here are examples of C and Pascal routines:

Pascal:

```
program p (output);
begin
   writeln ("hi!");
end.
```

C:

```
main() {
   printf("hi!\n");
}
```

4.2 Calling Pascal from C

To call a Pascal function from C, perform these two tasks:

1. Write a C *extern* declaration to describe the return value type of the main routine.
2. Write the call itself with the return value type and argument types as required by the Pascal routine.

See Section 4.2.3, "Calling a Pascal Function from C" below, for an example.

4.2.1 C Return Values

Use Table 4-1 below as a guide to declaring the return value type.

If Pascal function returns:	Declare C function as:
integer (also applies to subranges with lower bounds < 0)	int
cardinal (also applies to subranges with lower bounds > 0)	unsigned int
char	char
boolean	char
enumeration	unsigned, or corresponding enum (recall that C enums are signed)
real	none
double	double
pointer type	corresponding pointer type
record type	corresponding structure or union type
array type	corresponding array type

Table 4-1 Declaration of Return Value Types

To call a Pascal procedure from C, write an *extern* declaration in this form:

```
extern void name();
```

Then call it with actual arguments with appropriate types.

Use Table 4-2 as a guide for values to pass that correspond to the Pascal declarations listed.

C does not permit declaration of formal parameter types, but instead infers them from the types of the actual arguments passed. (See Section 4.2.3 for an example.)

4.2.2 C to Pascal Arguments

Table 4-2 shows the C argument types to declare in order to match those expected by the Pascal routine.

If Pascal expects:	C argument should be:
integer	integer or char value $-2^{31}..2^{31}-1$
cardinal	integer or char value $0..2^{32}-1$
subrange	integer or char value in subrange
char	integer or char (0..255)
boolean	integer or char (0 or 1 only)
enumeration	integer or char (0..N-1)
real	none
double	float or double
procedure	struct {void *p(); int *l}
function	struct {function-type *f(); int *l}
pointer types[1]	pointer type, under 0 := lbound(s)
reference parameter	pointer to the appropriate type
record types	structure or union type

Table 4-2 C Argument Types

If Pascal expects:	C argument should be:
by-reference array parameters	corresponding array type
by-reference text	FILE**
by-value array parameters	structure containing the corresponding array

[1]To pass a pointer to a function in a C-to-Pascal call, you must pass a structure by value. The first word of the structure must contain the function pointer and the second word a zero. Pascal requires this format because it expects an environment specification in the second word.

Table 4-2 (continued) C Argument Types

4.2.3 Calling a Pascal Function from C

This example shows a C routine calling a Pascal function.

Pascal:

```
function bah (
    var f: text;
    i: integer
    ) double;
begin
    ...
end {bah};
```

C declaration of bah:

```
extern double bah();
```

C call:

```
int i; double d;
FILE *f;
d = bah(&f, i);
```

4.2.4 Calling a Pascal Procedure from C

This example shows a C routine calling a Pascal procedure.

Pascal:

```
type
  int_array = array[1..100] of integer;
procedure humbug (
    var a: int_array;
    n: integer
    ): integer;
  begin
    ...
  end {humbug};
```

C declaration:

```
extern void humbug();
```

C call:

```
int a[100];
int n;
humbug(a, n);
```

4.2.5 Passing Strings to a Pascal Procedure

The following example is a C routine that passes strings to a Pascal procedure, which then prints the strings. The example illustrates two points:

- The Pascal routine must check for the null [*chr*(0)] character, which indicates the end of the string passed by the C routine.
- The Pascal routine must not write to *output*, but instead uses the *stdout* file-stream descriptor passed by the C routine.

C routine:

```
/* Send the last command-line argument to the Pascal routine */

#include <stdio.h>
main(int argc, char **argv)
{
   FILE *temp = stdout;
   if (argc != 0)
      p_routine(&temp, argv[argc-1]);
}
```

Pascal routine:

```
{ We assume the string passed to us by the C program will not
exceed 100 bytes in length. }
type
  astring = packed array [1..100] of char;
procedure p_routine(var f: text; var c: astring);
  var
    i: integer;
  begin
    i := lbound(c);
    { check for null character }
    while (i > hbound(c)) and (c[i] <> chr(0)) do
    begin
      write (f, c[i])  { write to file stream descriptor
passed from C }
      i := i+1;
    end;
    writeln(f);
  end; {p_routine}
```

4.3 Calling C from Pascal

To call a C routine from Pascal, you must write:

1. A Pascal declaration describing the C routine.
2. A procedure declaration or, if the C routine returns a value, a function declaration.
3. Parameter and return value declarations corresponding to the C parameter types, using Table 4-3 below as a guide.

If C expects:	Pascal parameter should be:
int (same as *signed int, long, signed long*, and *signed*)	integer
unsigned int (same as *unsigned* and *unsigned long*)	cardinal
short (same as *signed short*)	integer (or -32768..32767)
unsigned short	cardinal (or 0..65535)
char (same as *unsigned char)*	char
signed char	integer (or -128..127)
float	double
double	double
enum type	corresponding enumeration type
string (char *)	packed character array passed by reference (VAR)
pointer to function	none
FILE*	none
FILE**	text, passed by reference (VAR)
pointer type	corresponding pointer type or corresponding type passed by reference
struct or union type	corresponding record type
array type	corresponding array type passed by reference (VAR)

Table 4-3 Pascal Parameter Data Types Expected by C

4.3.1 Calling a C Procedure

The following example shows code calling a C procedure from Pascal.

Note: A Pascal routine cannot pass a function pointer to a C routine.

C routine:

```
void bah (int i, float f, char *s)
{
    ...
}
```

Pascal declaration:

```
procedure bah (
    i: integer;
    f: double;
    var a: packed array[1..100] of char);
external;
```

Pascal call:

```
str := "abc\0";
bah(i, 1.0, str);
```

4.3.2 Calling a C Function

The following example shows code calling a C function from Pascal.

C routine:

```
float humbug(FILE **f, struct scrooge *x)
{
    …
}
```

Pascal declaration:

```
type
  scrooge_ptr = ^scrooge;
function humbug (
    var f: text;
    x: scrooge_ptr
    ): double;
  external;
```

Pascal call:

```
x := humbug(input, sp);
```

4.3.3 Passing Arrays

The following example shows code calling a C array from Pascal.

C routine:

```
int sum(int a[], unsigned n)
{
    …
}
```

Pascal declaration:

```
type
  int_array = array[0..100] of integer;
function sum (
    var a: int_array;
    n: cardinal
    ): integer;
  external;
avg := sum(samples, hbound(samples) + 1) /
        (hbound(samples)+1);
```

Appendix A

Man Pages

This appendix is a listing of the *pc*(1) man page.

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-0740-030.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389