UPDATE PACKAGE

UPD4303-192

for

PASCAL REFERENCE GUIDE, DOC4303-191

June 1983

This Update Package, UPD4303-192, is Update 1 for the December 1982 Edition of the Pascal Reference Guide, DOC4303-191. This package contains 264 pages. A list of effective pages appears on the next page.

Changes made to the text since the last printing are identified by vertical bars in the margin. Change bars with numbers identify new Pascal features of Software Release 19.2. Change bars without numbers identify documentation corrections and clarifications.

Technical Publications Department
500 Old Connecticut Path
Framingham, MA 01701

(Pages with changes, enclosed with this package, are underlined.)

Effective Pages for the Pascal Reference Guide at Software Release 19.2.

Pages

ii to v
vi to ix
x to xiii

1-1 to 1-2
1-3
1-4 to 1-6

2-1 to 2-2
2-3
2-4 to 2-6
2-7
2-8 to 2-10
2-11
2-12 to 2-13
2-14
2-15 to 2-17

3-1
3-2
3-3 to 3-8

4-1 to 4-4
4-5
4-6 to 4-8
4-9
4-10
4-11
4-12

5-1 to 5-3
5-4
5-5 to 5-15

6-1 to 6-2
6-3 to 6-8
6-9
6-10 to 6-13
6-14 to 6-14L
6-15 to 6-16
6-17 to 6-17A
6-18 to 6-33

7-1
7-2 to 7-4
7-5 to 7-6
7-7 to 7-7A
7-8

Pages

8-1 to 8-2
8-3
8-4 to 8-16

9-1 to 9-8
9-9 to 9-9A
9-10 to 9-11
9-12
9-13 to 9-17
9-18 to 9-18A
9-19 to 9-22

10-1
10-2 to 10-3
10-4
10-5 to 10-5A
10-6 to 10-10
10-11 to 10-12
10-13 to 10-23
10-24

11-1
11-2 to 11-4
11-5

A-1 to A-3
A-4 to A-4A
A-5

B-1
B-2 to B-7
B-8 to B-9

D-1
D-2
D-3 to D-8
D-9

X-1 to X-16

Pascal Reference Guide

DOC4303-191

Second Edition

by

A. Paul Cioto

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 19.1 (Rev. 19.1).

Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

COPYRIGHT INFORMATION

HOW TO ORDER TECHNICAL DOCUMENTS

U.S. Customers

Software Distribution
Prime Computer, Inc.
1 New York Ave.
Framingham, MA 01701
(617) 879-2960 X2053, 2054

Customers Outside U.S.

Contact your local Prime subsidiary or distributor.

Prime Employees

Communications Services
MS 15-13, Prime Park
Natick, MA 01760
(617) 655-8000 X4837

PRIME INFORMATION

Contact your Prime INFORMATION dealer.

PRINTING HISTORY — PASCAL REFERENCE GUIDE

Edition	Date	Number	Software Release
First Edition	October 1980	IDR4303	17.6
Update 1	December 1980	PTU2600-080	18.1
Update 2	July 1982	PTU2600-086	19.0
Second Edition	December 1982	DOC4303-191	19.1

This edition is a complete revision of IDR4303. It incorporates update material up to and including software release 19.1, corrects all known errors, and has been revised for clarity.

Changes made to the text since the last printing have been indicated with change bars in the margin. Change bars with numbers indicate technical changes. Those without numbers indicate rewrites for clarification or additional information. Appendixes A and D are new.

SUGGESTION BOX

All correspondence on suggested changes to this document should be directed to:

A. Paul Cioto
Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

Contents

Part II -- Compiling, Loading, and Executing Programs

- Chapter 2 provides information on the use of Prime's Pascal compiler, including compiler options.
- Chapter 3 provides information on loading and executing programs with Prime's SEG utility.

Part III -- Pascal Language Reference

- Chapter 4 provides brief descriptions of Pascal language elements and of terms used throughout Part III.
- Chapter 5 lists the fundamental elements of the Pascal program structure.
- Chapter 6 describes the data types available in Pascal, including two Prime extension data types called LONGINTEGER and LONGREAL.
- Chapter 7 describes the use of Pascal expressions.
- Chapter 8 describes the use of executable Pascal statements.
- Chapter 9 describes the use of procedures and functions, including external procedures and functions, which are declared with Prime's EXTERN attribute.
- Chapter 10 offers a detailed discussion of how to input and output data in Prime Pascal.
- Chapter 11 lists standard Pascal functions.

Appendixes

- Appendix A summarizes Prime extensions and restrictions to standard Pascal. It also references the chapter in which each extension or restriction is discussed.
- Appendix B illustrates how Prime Pascal data types are represented in storage.
- Appendix C lists the ASCII character set, which Prime Pascal uses.
- Appendix D lists guidelines for interfacing Pascal to some of Prime's other high-level languages.

Error Messages

Pascal compiler error messages, which were designed to be self-explanatory, appear on your terminal at compile time, and in the listing file if one is created. Therefore, the messages are not listed in this book.

RELATED DOCUMENTS

In addition to the Pascal Reference Guide, you will most likely need other documents to help you take full advantage of Prime's powerful utilities, which are separately priced products. These documents are listed below.

Prime User's Guide

Complete instructions for creating, loading, and executing programs in Prime Pascal or in most Prime languages, plus extensive additional information on Prime system utilities for programmers, are found in the Prime User's Guide. The Prime User's Guide and the Pascal Reference Guide are both essential to the Pascal programmer.

The Prime User's Guide also contains a complete guide to all Prime documentation.

Draft Proposal "X3J9/81-093" Programming Language Pascal

The definitive reference for standard Pascal is The Draft Proposal "X3J9/81-093" Programming Language Pascal. Every installation that uses Pascal extensively should have a copy of this proposed standard, which may be obtained from American National Standards Institute, 1430 Broadway, New York, NY 10018.

New User's Guide to EDITOR and RUNOFF

Prime's EDITOR is an interactive line-oriented text-editing utility. It is used to enter and modify text in the computer. New programs that do not rely on cards or tapes can be input to the system at a terminal using EDITOR.

The New User's Guide to EDITOR and RUNOFF contains a complete description of the EDITOR, and describes RUNOFF, Prime's text-formatting utility. It also provides a basic introduction to the Prime system for those with little or no computer experience.

The caret (or arrow) that appears just above the error message points to the actual error on the line of code. The following is an example of an error message:

```
OK, PASCAL TEST.PASCAL
[PASCAL Rev. 19.1]
   14       END {main program}
             ^

ERROR 31 SEVERITY 3 BEGINNING ON LINE 14
Missing dot at program end.
```

When compilation is complete and all the error messages have been listed on the terminal, the compiler tells you how many errors were encountered and the maximum severity. For example: 19.1

```
0013 ERRORS (PASCAL-REV. 19.1)
MAXIMUM SEVERITY IS 3
```

The significance of the severity code is:

Severity	Description
1	Warning
2	Error that the compiler has attempted to correct
3	Uncorrected error (prevents optimization, code generation, and therefore successful compilation)
4	Error that immediately halts compilation

A severity 1 or 2 error will not prevent execution of your program, but the output may be unpredictable.

Error Messages Involving %INCLUDE Files

A %INCLUDE file is a Prime extension. It is an external file that is compiled with the main program after the %INCLUDE statement. The %INCLUDE statement is followed by the name of the file to be included. The format is:

```
%INCLUDE 'filename';
```

%INCLUDE files can hold any legal Pascal code -- declarations as well as executable statements. The files could, for example, contain long lists of variable declarations. (For more information on %INCLUDE files, see Chapter 5.)

If you compile a program that inserts a %INCLUDE file, and there are compile-time errors in that file, a special type of error message format is printed at the terminal:

```
<line-number> line-of-code
              ^

ERROR xxx SEVERITY y BEGINNING ON LINE line-number IN FILE 'filename'
explanation
```

line-number	The number of the line in the %INCLUDE file where the error occurred. (Lines of code in %INCLUDE files are numbered separately, and the numbers are enclosed in angle brackets in the listing file.)
line-of-code	The actual erroneous line of code in the %INCLUDE file.
xxx	The error code number.
18.3 y	Severity code number.
'filename'	The name of the %INCLUDE file.
explanation	Description of the error and possible remedies.

The caret points to the erroneous line of code.

Here is an example of a %INCLUDE file error message:

```
<23>  VAR a : integer;
          ^

ERROR 2 SEVERITY 3 BEGINNING ON LINE 23 IN FILE 'test-1'
This item in a variable definition list is
already defined in this block.
```

The compiler adds the number of errors from the %INCLUDE file to the number of errors in the main program, and gives the total number of errors at the end of compilation.

FILENAME CONVENTIONS

When you compile a program with the PASCAL command, and there are no severity 3 or 4 errors, the compiler creates an object (binary) file. 18.0 It also creates a source listing file if the -LISTING option is specified on the command line. In order for you and the compiler to identify and compile the source file and create the object and listing files, the "suffix" conventions, which are described below, should be used to name these files on Rev. 18 (or higher) systems.

Table 2-1

Options Commonly Used and Not Commonly Used
(Defaults are underlined.)

Options Commonly Used	Options Not Commonly Used
-BINARY [argument]	-BIG and -NOBIG
-DEBUG and -NODEBUG	-64V and -32I
	-EXPLIST and -NOEXPLIST
-ERRTTY and -NOERRTTY	-EXTERNAL and -NOEXTERNAL
-LISTING [argument]	-FRN and -NOFRN
-MAP and -NO_MAP	-INPUT pathname
	-OFFSET and -NOOFFSET
-OPTIMIZE, -OPT1, -OPT3, and -NOOPTIMIZE	-PRODUCTION and -NOPRODUCTION
-RANGE and -NORANGE	-SILENT and -NOSILENT
-UPCASE	-SOURCE pathname
-XREF and -NOXREF	-STANDARD and -NOSTANDARD
	-STATISTICS and -NOSTATISTICS

▶ -BIG and -NOBIG

-BIG and -NOBIG determine the type of code generated for references to ARRAY or RECORD formal variable parameters in a subprogram.

With -BIG, an ARRAY or RECORD formal variable parameter can become associated with any ARRAY or RECORD, even if the ARRAY or RECORD crosses a segment boundary.

With -NOBIG, an ARRAY or RECORD formal variable parameter can be associated only with an ARRAY or RECORD that does not cross a segment boundary.

See ARRAY or RECORD Type Variable Parameters in Chapter 9 for details.

▶ -BINARY [argument]

The -BINARY option generates an object (binary) file. If this option is not given, -BINARY YES will be assumed. The argument may be:

pathname	Object code will be written to the file pathname.
YES	Object code will be written to the file named program.BIN, or B_program, in the user's UFD, where program is the name of the source file. (This is the default.)
NO	No object file will be created. Specified when only a syntax check or listing is desired.

▶ -DEBUG and -NODEBUG

The -DEBUG option generates code for Prime's source level debugger. With -DEBUG, the object file is modified so that it will run under the debugger. Execution time increases, and the code generated will not be optimized.

18.2

-NODEBUG, the default, causes no debugger code to be generated.

See the Source Level Debugger Guide for information about debugging programs.

▶ -ERRTTY and -NOERRTTY

18.0

The -ERRTTY option prints error messages at the user's terminal. -NOERRTTY suppresses this function.

▶ -OPTIMIZE, -OPT1, -OPT3, and -NOOPTIMIZE 19.1

These options control the optimization phase of the compiler.

-OPTIMIZE, the default, will cause the object code to be optimized. Optimized code runs more efficiently than nonoptimized code, but takes somewhat longer to compile.

The -OPT1 option optimizes less code and generates less efficient code than -OPTIMIZE, but compilation time is faster than -OPTIMIZE. 19.1

The -OPT3 option optimizes more code and generates more efficient code than -OPTIMIZE, but compilation time is slower than -OPTIMIZE.

When -NOOPTIMIZE is invoked, optimization does not occur. Execution time is slowest, and compile time is fastest.

▶ -PRODUCTION and -NOPRODUCTION

-PRODUCTION produces alternative option-controlling code for the debugger.

-PRODUCTION is similar to DEBUG, except that the code generated will not permit insertion of statement breakpoints. Execution time is not affected.

-NOPRODUCTION will cause no production-type code to be generated.

▶ -RANGE and -NORANGE

-RANGE checks for out-of-bounds values of array subscripts and character substring indexes. Error-checking code is inserted into the object file. If an array subscript or character substring index takes on a value outside the range specified when the referenced data item was declared, a runtime error will be generated. Range checking decreases the efficiency of the generated code.

With -NORANGE, out-of-bounds values will not be detected. The program will be more vulnerable to errors, but will execute more quickly.

▶ -SILENT and -NOSILENT

-SILENT suppresses severity 1 error messages. Severity 1 error messages will not be printed at the terminal and will be omitted from any listing file.

-NOSILENT causes severity 1 error messages to be retained.

▶ -SOURCE pathname

The -SOURCE option, which is identical to the -INPUT option, is obsolete and not useful. -SOURCE designates the source file pathname to be compiled:

```
PASCAL -SOURCE pathname
```

It is not useful because it produces the same results as:

```
PASCAL pathname
```

pathname must not be designated more than once on the command line.

▶ -STANDARD and -NOSTANDARD

The -STANDARD option generates a severity 1 error message when your code's syntax is non-ANSI standard Pascal. -NOSTANDARD does not cause a severity 1 error to be generated.

▶ -STATISTICS and -NOSTATISTICS

The -STATISTICS option lists compilation statistics at the terminal after each phase of compilation. For each phase the list contains:

DISK	Number of reads and writes during the phase, excluding those needed to obtain the source file
SECONDS	Elapsed real time
SPACE	Internal buffer space used for symbol table, in 16K byte units
PAGING	Disk I/O time used
CPU	CPU time used in seconds, followed by the clock time when the phase was completed

-NOSTATISTICS causes no statistics to be printed.

▶ -UPCASE

The -UPCASE option causes the compiler to map lowercase variables to uppercase. With -UPCASE, the compiler does not distinguish between lowercase variables and uppercase variables, except within character strings. 18.2

▶ -XREF and -NOXREF

The -XREF option appends a cross-reference to the source listing. A cross-reference lists, for every variable, the number of every line on which the variable was referenced.

-NOXREF causes no cross-reference listing to be generated.

▶ -64V and -32I

These determine the addressing mode to be used in the object code. -64V is a segmented virtual addressing mode for 16-bit machines. -32I is a segmented virtual mode, which takes maximum advantage of the 32-bit architecture of Prime's more advanced models (P450 and up).

COMPILER OPTION ABBREVIATIONS

Most compiler options have abbreviations that are accepted by the compiler. For example, instead of typing -LISTING on the command line, you could simply type -L. A list of Prime's recommended abbreviations, along with a summary of options in straight (nonpaired) alphabetical order, is given in Table 2-2.

Table 2-2

Summary of Compiler Options and Abbreviations
(Defaults are underlined.)

Option	Abbreviation	Significance
-BIG	-BIG	Generate boundary-spanning code
-BINARY	-B	Create object file
-DEBUG	-DE	Generate debugger code
-ERRTTY	-ERRT	Print error messages at terminal
-EXPLIST	-EXP	Generate an expanded source listing
-EXTERNAL	-EXT	Generate external procedure definitions
-FRN	-FRN	Generate floating-point round instructions
-INPUT	-I	Designate source file
-LISTING	-L	Create source listing
-MAP	-MA	Print listing file with map
-NOBIG	-NOB	Don't generate boundary-spanning code
-NODEBUG	-NOD	Don't generate code for debugger
-NOERRTTY	-NOERRT	Don't print error messages at terminal
-NOFRN	-NOFRN	Don't generate FRN instruction
-NO_MAP	-NOM	Don't include a map in listing file
-NOOFFSET	-NOOF	Don't append an offset map to source listing
-NOOPTIMIZE	-NOOP	Don't optimize object code

Table 4-1
Pascal Punctuation Symbols

Symbol	Description
+	Addition Identity Set union
-	Subtraction Sign-inversion Set difference
*	Multiplication Set intersection
/	Division (real)
=	Equal to Set equality Type identifier and type separator Constant identifier and constant separator
<	Less than
>	Greater than
[]	Subscript list or set constructor delimiters
.	Decimal point Record selector Program terminator
,	Parameter or identifier separator
:	Variable name and type separator Label and statement separator
;	Statement separator Record field separator Declaration separator
^	File or pointer variable indicator
()	Parameter list, identifier list, or expression delimiters
<>	Not equal to Set inequality

Table 4-1 (continued)
Pascal Punctuation Symbols

Symbol	Description
<=	Less than or equal to Set inclusion ("is contained in")
>=	Greater than or equal to Set inclusion ("contains")
:=	Assignment Operator
..	Subrange Specifier
{}	Comment delimiters
/* */	Comment delimiters (Prime extension)
(* *)	Comment delimiters
'	Character-string delimiter (apostrophe)
&	Bit Integer AND operator (Prime extension)
!	Bit Integer OR operator (Prime extension)

18.3

Table 4-3
Standard Identifiers

Constants

FALSE	TRUE	MAXINT	

Types

INTEGER	LONGINTEGER*	REAL	LONGREAL*
BOOLEAN	CHAR	TEXT	

19.1

Files

INPUT	OUTPUT

Directives

FORWARD	EXTERN*

Functions

ABS	EXP	SIN
ARCTAN	LN	SQR
CHR	ODD	SQRT
COS	ORD	SUCC
EOF	PRED	TRUNC
EOLN	ROUND	

Procedures

CLOSE*	PAGE	RESET
DISPOSE	PUT	REWRITE
GET	READ	WRITE
NEW	READLN	WRITELN

* Prime extension identifiers

19.1 There are two ways of expressing real and longreal numbers:

1. In decimal notation, the number is expressed by an optional sign, a whole number part, a decimal point, and a fractional part. There must be at least one digit on each side of the decimal point.

2. In scientific notation, the number is represented by a value, followed by the letter E or D, which is followed by an exponent. The letter E is used if the number is REAL. The letter D is used if the number is LONGREAL. The value consists of an optional sign, one or more digits, and an optional decimal point and fractional part. The exponent must be an integer with an optional sign. The letter E or D is read as "times 10 to the power of". This is a convenient way to represent very large or very small numbers.

No comma may appear in a number. Examples:

Valid Integer/Longinteger	Invalid Integer
23	-32,768 (No comma allowed)
-100	
+40000 (longinteger)	

Valid Real/Longreal Number	Invalid Real Number
-0.1	.1 (Must be a digit to the left of the decimal point)
1E6 (1000000)	
5E-8 (0.00000005)	1. (Must be a digit to the right of the decimal point)
-87.35E+15 (-87350000000000000)	
-7.0E-6 (-0.000007)	-8.0E-6.3 (Only whole number exponents allowed)
2.1D01 (longreal)	
1.234567 (longreal)	1,234D+20 (No comma allowed)

6
Data Types

Every constant, variable, function, or expression must have a data type. The data type determines the set of values a variable may assume or a function or an expression may generate. The data type also determines which operations may be performed on the values and how these values are represented in storage.

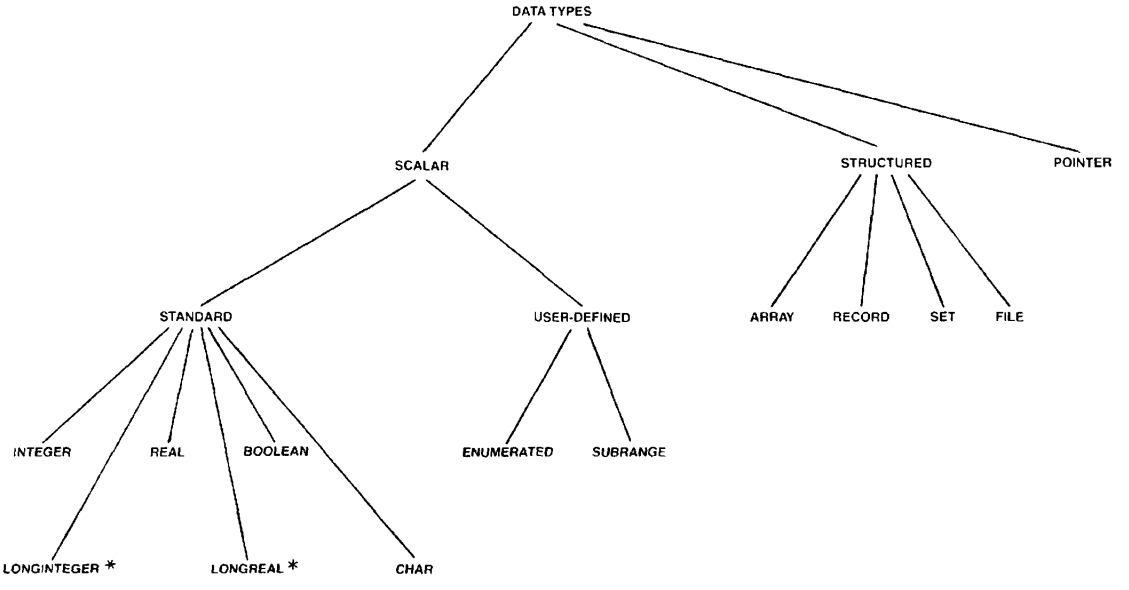
This chapter summarizes the data types available in Prime Pascal — standard Pascal data types as well as Prime extensions. There are two Prime extension data types, LONGINTEGER and LONGREAL. Each of these data types is described later in this chapter. 19.1

Figure 6-1 illustrates all of the data types in Prime Pascal. The internal representations of data types are illustrated in Appendix B. Appendix D offers guidelines for interfacing Pascal data types with those of other languages. For more information about Pascal data types, consult a commercially available text.

SCALAR DATA TYPES

Scalar data types are the fundamental data types in Pascal. All other data types must be built from scalar data types.

Each scalar data type has a group of distinct values, called constants, which have a defined linear ordering. Thus, each scalar type is ordered. Any two of these constants can be compared by asking if one is less than, equal to, or greater than the other. The total number of constants in a type is called the cardinality of that type.



19.1 The Hierarchy of Data Types in Prime Pascal
19.1 *Prime extensions are flagged with an asterisk.

Figure 6-1

Scalar data types are divided into two classes: standard scalar data types and user-defined scalar data types. The standard scalar types are the predefined, built-in data types provided by Pascal. The user-defined scalar types are data types that you create and define in a program.

Standard Scalar Data Types

19.1 There are four standard scalar types -- INTEGER, REAL, BOOLEAN, and CHAR — plus two Prime extension scalar types called LONGINTEGER and LONGREAL.

The following program compares all of the printable characters (decimal 160-255) in Prime's character set, using relational operations:

```
PROGRAM Karacter;
VAR
  I : INTEGER;
BEGIN
  FOR I := 160 TO 255 DO
    BEGIN
      WRITE(CHR(I));
      IF ((CHR(I) >= 'A') AND (CHR(I) <= 'Z')) THEN
        WRITELN('  This is a capital letter')
      ELSE
      IF ((CHR(I) >= 'a') AND (CHR(I) <= 'z')) THEN
        WRITELN('  This is a small letter')
      ELSE
      IF ((CHR(I) >= '0') AND (CHR(I) <= '9')) THEN
        WRITELN('  This is a printable number')
      ELSE
        WRITELN('  This is punctuation or other character')
    END
END.
```

> Caution
>
> Prime's character set is represented by the decimal numbers 128 to 255. You should not use the CHR function on integers less than 128 or greater than 255. Any such attempt will produce unpredictable results.

To indicate a constant of the CHAR type, place an apostrophe (a single quote) on each side of the character. To indicate an apostrophe, write it twice. Examples:

```
'A'

'7'

';'

''''     {Single quote}

' '      {Blank is considered a printable character.}
```

> Note
>
> A constant of the CHAR type is always a single character. Constructs such as '123' or 'STRING' are not constants of this type but are constants of a more complex type called ARRAY OF CHAR or "string", which is described later in this chapter.

As was explained earlier, each character corresponds to its own internal integer, which is called the ordinal number of the character. Using the standard function ORD — the opposite of CHR — you can get a character's ordinal number. For example:

```
ORD('A') yields 193      {Octal value 301}

ORD('a') yields 225      {Octal value 341}

ORD('1') yields 177      {Octal value 261}
```

There are two more standard functions particularly useful for processing character data -- PRED (predecessor function) and SUCC (successor function). Given a value, PRED produces the next lesser value and SUCC gives the next greater value. For example:

```
PRED('E') yields 'D'    {The predecessor of 'E' is 'D'}

SUCC('E') yields 'F'    {The successor of 'E' is 'F'}

PRED(8)   yields  7     {The predecessor of 8 is 7}

SUCC(8)   yields  9     {The successor of 8 is 9}

PRED(ORD('G')) yields 198  {The predecessor of G's ordinal
                            value is 198}

SUCC(ORD('F')) yields 199  {The successor of F's ordinal
                            value is 199}
```

Functions are described in detail in Chapter 11.

The relational operators =, <>, <, >, <=, and >= can be used with all character constants. For more information, see Chapter 7.

User-defined Scalar Data Types

There are two user-defined scalar types -- enumerated and subrange.

The Enumerated Types: An enumerated type defines an ordered set of values by listing these values.

To create an enumerated type, use the following type definition:

```
TYPE type-identifier = (identifier-1, identifier-2 [,identifier-3]...);
```

Array of Characters: A line of text can be represented as an array of characters. This particular array is called ARRAY OF CHAR or "string".

A typical VAR declaration of an ARRAY OF CHAR would be:

```
VAR
  A : ARRAY[1..60] OF CHAR;
```

The identifier "A" is an array with 60 character elements. A[1] is the first character, and A[60] is the last. Any character string value assigned to A must have 60 characters.

Here is an example of how an ARRAY OF CHAR (string) type is declared within a TYPE declaration:

```
TYPE
  STRING1 = ARRAY[1..10] OF CHAR;
```

Two more examples follow:

```
TYPE
  STRING1 = ARRAY[1..10] OF CHAR;
VAR
  STRING2 : STRING1;
BEGIN
  STRING2 := 'ABCDEFGHIJ';
  STRING2 := 'AB'            {This is an invalid assignment.}
                             {The string must contain 10}
                             {characters.}
END.
```

Here is another example:

```
TYPE
   LENGTH = 1..30;
   STRING30 = ARRAY [LENGTH] OF CHAR;
VAR
   ALPHA : STRING30;
   I : LENGTH;
BEGIN
   FOR I := 1 TO 30 DO
      READ (ALPHA[I])
END.
```

Note

Although Prime Pascal does not support the keyword PACKED in type definitions, an ARRAY OF CHAR is always stored as a packed ARRAY OF CHAR on Prime computers.

Array of Characters (the Prime Extension): At Rev. 19.1, the ARRAY OF CHAR was enhanced into a Prime extension that makes reading of these arrays much easier.

On Rev. 19.1 (or higher) systems, you can read an array of characters as one unit, instead of reading one character at a time. For example, Prime's ARRAY OF CHAR function allows you to declare and read character arrays like this:

```
PROGRAM Primearray;
VAR
  A : ARRAY[1..10] OF CHAR;
  B : ARRAY[1..60] OF CHAR;
BEGIN
  READLN(A);
  READLN(B)
END.
```

Previously, reading was done one character at a time within a loop:

```
BEGIN
  FOR I := 1 TO 10 DO
    READ(A[I]);
  FOR I := 1 TO 60 DO
    READ(B[I])
END.
```

19.1

You can still use loops to read an ARRAY OF CHAR; however, it is easier and more efficient to use the Prime extension.

Note

If you do not have a Rev. 19.1 (or higher) system, then you must use the loops.

You can read or write a Prime character array up to 256 characters long.

Consider the following:

```
VAR
  A : ARRAY[1..30] OF CHAR;
BEGIN
  READLN(A);
```

With READLN, if fewer than 30 characters are typed in, the remaining characters will be blank-filled. If more than 30 characters are typed in, only the first 30 characters will be assigned. You will not be warned that you have typed in extra characters.

7
Expressions

An expression is a single operand or a combination of operands and operators that are evaluated to produce a value.

OPERANDS

An operand may be any of the following expressions:

- A variable
- An unsigned or signed number
- A character string
- A constant identifier
- A function designator (explained in Chapter 9)
- NIL
- A set

Here are some examples of valid operands:

```
15

(x+y+z)

SIN(x+y)

[RED, C, GREEN]

[1, 5, 10..19, 23]

NOT P

I * J + 1

-N
```

OPERATORS

Operators modify an operand or combine two operands. Operators can be classified as arithmetic, relational, set, Boolean, or integer. (Integer operators are Prime extensions.)

Arithmetic Operators

An arithmetic operator specifies computation to be performed on its operands to produce a single numeric value. Table 7-1 lists the binary and unary arithmetic operators and the data types of operands and results.

Table 7-1
Arithmetic Operators

Binary Operators	Type of Operands	Type of Result
+ (add) - (subtract) * (multiply)	INTEGER/LONGINTEGER REAL/LONGREAL	INTEGER/LONGINTEGER if both operands are INTEGER/LONGINTEGER; otherwise REAL/LONGREAL
/ (divide)	INTEGER/LONGINTEGER REAL/LONGREAL	REAL/LONGREAL
DIV (divide with truncation)	INTEGER or LONGINTEGER	INTEGER/LONGINTEGER
MOD (modulus or remainder)	INTEGER or LONGINTEGER	INTEGER/LONGINTEGER
Unary Operators		
+ (identity) - (sign-inversion)	INTEGER/LONGINTEGER REAL/LONGREAL	Same as operand

19.1

Relational Operators

The relational operators are used to compare values of data types — scalar, ARRAY OF CHAR (string), pointer, or SET. In any given comparison, both operands must be of the same type, except that INTEGER can be compared with LONGINTEGER, and REAL with LONGREAL. The result of the comparison is a BOOLEAN value, TRUE or FALSE. Table 7-2 lists the legal relational operators and data types of operands. 19.1

Table 7-2
Relational Operators

Operator	Operation	Type of Operands
= <>	equality inequality	SET, scalar, pointer, or ARRAY OF CHAR
< >	less than greater than	scalar or ARRAY OF CHAR
<=	less or equal	scalar or ARRAY OF CHAR
<=	set inclusion ("is contained in")	SET
>=	greater or equal	scalar or ARRAY OF CHAR
>=	set inclusion ("contains")	SET
IN	set membership	first (left) operand is any scalar type (except REAL and LONGREAL), second (right) operand is a set of that type

19.1

Here are some examples of relational operators.

First, let

```
x := ['A', 'D', 'C', 'B']

y := ['A', 'E']
```

then

```
x = ['A', 'B', 'C', 'D']    {true }

y <= x                      {false}

y <> x                      {true }

'B' IN x                    {true }
```

Integer Operators

The integer operators & and ! are Prime extensions. They perform Boolean AND and OR operations on integers respectively. These operators also work on longintegers. For example, if you wanted to perform AND and OR operations on the two numbers 10 and 12, you could say:

```
VAR
  A,B,C,D : integer;
BEGIN
  A := 10;
  B := 12;
  C := A & B;  {AND operation}
  D := A ! B;  {OR operation}
  WRITELN(C);
  WRITELN(D);
END.
```

At the machine level, the two binary numbers that stand for decimal 10 and 12 are 1010 and 1100 respectively. (The 12 leading zeros are omitted.) During the AND and OR operations, the digit 1 means TRUE and 0 means FALSE. The first digit of 1010 is compared with the first digit of 1100, and so on, to produce new binary (and hence decimal) numbers C and D. The machine, therefore, calculates:

```
1010 AND 1100 = 1000  {decimal 8}
1010 OR  1100 = 1110  {decimal 14}
C = 8
D = 14
```

Integer operators can be useful when you need a lot of Boolean TRUE and FALSE values or "switches" that can be set to 1 (TRUE) or 0 (FALSE) in the internal binary representation of any decimal number.

OPERATOR PRECEDENCE

The precedence among operators determines the order in which expressions are evaluated. The precedence of operators is as follows:

1. Operations in parentheses
2. NOT, unary - and +
3. *, /, DIV, MOD, AND, &
4. +, -, OR, !
5. =, <>, <, >, <=, >=, IN

Highest precedence (done first)

Lowest precedence (done last)

Order of Evaluation

When there are several operations at the same level of precedence, the operations are performed from left to right.

Parentheses may be used to override the normal evaluation order. An expression enclosed in parentheses is treated as a single operand, and is evaluated first. When expressions are contained within a nest of parentheses, evaluation proceeds from the innermost set to the outermost set (inside out).

For example:

```
7 + A * 2 - 5 DIV 3 + A          {Numbers below the operators
                                  indicate the order in which
  2   1   4    3    5            the operations are performed.}


( ( 7 + A ) * 2 - 5 ) DIV 3 + A

     1     2   3      4    5
```

The following are some guidelines for using assignment statements:

- The variable or function identifier and the expression must be of compatible types.
- Neither the variable/function identifier nor the expression should be a FILE type or a structured type with a FILE element.
- The variable or function identifier can be of type REAL and the expression can be of type INTEGER; however the converse is not possible. (You can assign an integer to a real, but not a real to an integer unless the TRUNC function is used.)
- The variable or function identifier can be of type LONGINTEGER and the expression can be of type INTEGER, but the converse may cause your program to fail. (You may assign an integer to a longinteger, but a longinteger will be truncated when assigned to an integer.) This rule also applies to REAL and LONGREAL for the same reason. 19.1
- Any element, group of elements, or expression that is of a particular SET type must be assigned to a variable or function identifier of the same SET type.
- The variable or function identifier and expression can be type ARRAY OF CHAR (string) as long as both arrays have the same number of elements.
- The variable or function identifier and expression can be subranges of each other.

PROCEDURE STATEMENT

A procedure statement activates the execution of a procedure. A procedure is a subprogram, which is declared in the main program.

The format of the procedure statement is:

```
procedure-identifier [(parameter-list)];
```

The procedure-identifier is the name of the procedure. When the procedure statement is encountered in the main program, the procedure is executed. The parameter-list is optional. If you want to pass values to and from the main program and the procedure, you would use parameters. The parameter-list is enclosed in parentheses, and the parameters are separated by commas.

Here are some examples of procedure statements:

```
PRINTHEADING;

TRANSPOSE(A,N,M);

BISECT(FCT, -1.0, +1.0, X);
```

For more information on procedures and functions, including external procedures and functions, see Chapter 9.

COMPOUND STATEMENT

A compound statement is a sequence of statements separated by semicolons. The general form of a compound statement is:

```
BEGIN
   statement-1 ; statement-2;...[statement-n]
END;
```

The keywords BEGIN and END must designate the start and the end of the sequence of a compound statement. They are not statements themselves. BEGIN and END should not be used on a single statement. statement-1, statement-2, etc. can be any Pascal statements. A compound statement can appear anywhere a single statement is allowed.

Example 1:

```
BEGIN
  Z := X;
  X := Y;
  Y := Z
END;
```

Example 2:

```
IF FLAG = 1 THEN
   BEGIN
     COUNTER := 0;
     READ (CHARACTER);
     WHILE (CHARACTER <> BLANK) DO
       BEGIN
         COUNTER := COUNTER + 1;
         READ (CHARACTER)
       END;
     WRITELN (' THE NUMBER OF CHARACTERS = ', COUNTER)
   END
ELSE
  FLAG := 0;
```

Example 5:

```
VAR
  I, J : INTEGER;
PROCEDURE ADD2(PROCEDURE A1);
  BEGIN {procedure ADD2}
    A1;
    A1
  END;
PROCEDURE ADD1;
  BEGIN {procedure ADD1}
    I := I + 1
  END;
PROCEDURE CALLPROC(PROCEDURE X(PROCEDURE Y); PROCEDURE Z);
  BEGIN {procedure CALLPROC}
    Z;
    X(Z)
  END;
BEGIN {main program}
  I := 0;
  CALLPROC(ADD2, ADD1)
END. {I = 3}
```

19.1

PROCEDURES

A procedure is a user-written independent program unit that performs a set of operations. A procedure must be declared in a procedure declaration, a forward procedure declaration, or an external procedure declaration before the procedure can be called by a procedure statement.

Procedure declarations are discussed below. Forward and external procedure declarations are discussed later in this chapter.

The external procedure declaration is a Prime extension to standard Pascal.

Procedure Declarations

A procedure declaration defines and names a procedure. The form of a procedure declaration is:

```
PROCEDURE identifier [(formal-parameter-list)]; block;
```

The keyword PROCEDURE begins a procedure declaration. The identifier is the name of the procedure. The list of formal parameters, if any, enclosed in parentheses, specifies the name of each formal parameter followed by its type-identifier. If you choose to use them, parameters can be passed by value or by reference to the subprogram. Parameters are discussed earlier in this chapter.

Except in forward or external declarations, the procedure heading described above is immediately followed by the procedure block.

A procedure block has the same general form as a program block. It may contain declarations for labels, constants, types, variables, procedures, and functions and a sequence of executable statements surrounded by a BEGIN and END pair. However, the procedure block ends with a semicolon instead of a period.

Unlike a function, the name of a procedure must not be assigned a value. Therefore, do not specify a data type for a procedure itself.

Note

Identifiers and labels declared in the main program are global. That is, they can be referenced throughout the entire program, including these procedures (or functions), so long as the procedures are contained within the main program (are not external). However, those identifiers and labels applying only to a particular procedure (or function) but not to the program as a whole should be declared within that procedure (or function). These identifiers and labels are local.

Invoking Procedures

A procedure statement invokes, or calls, a procedure. A procedure statement has the form:

```
procedure-identifier [(actual-parameter-1 [,actual-parameter-2]...)]
```

The procedure-identifier is the name of the called procedure. When the called procedure has one or more formal parameters defined in its heading, the procedure statement must contain the corresponding actual parameters along with the procedure-identifier.

Example 1:

```
PROGRAM TEST;
   .
   .
   .
PROCEDURE INDATA;...BEGIN...END;
PROCEDURE SORT;...BEGIN...END;
PROCEDURE OUTDATA;...BEGIN...END;
{Main program begins here.}
BEGIN
   INDATA;
   SORT;
   OUTDATA
END.
```

Example 2:

```
PROGRAM CURVE(INPUT,OUTPUT);
VAR
  X, Y : REAL;
  I : INTEGER;
  .
  .
  .
PROCEDURE PLOT(A, B: REAL; J: INTEGER); {A, B, & J are formal value
  .                                      parameters.}
  .
  .
BEGIN...END;
PROCEDURE ENDPLOT;
  .
  .
  .
BEGIN...END;
{Main program begins here.}
BEGIN
  X := 0.0;
  Y := 1.0 + SIN(X);
  READLN(I);
  I := I + 2;
  PLOT(X, Y, I);  {X, Y, and I are actual parameters.}
  .
  .
  .
  ENDPLOT;
  .
  .
  .
END.
```

Standard Procedures

A standard procedure, denoted by a predefined identifier, is a built-in procedure supplied by the Pascal language.

Prime Pascal supports the following standard procedures:

- File Handling Procedures: RESET, GET, REWRITE, PUT, READ, READLN, WRITE, and WRITELN. (See Chapter 10.)
- I/O Auxiliary Procedures: PAGE and CLOSE. (See Chapter 10.)
- Dynamic Allocation Procedures: NEW and DISPOSE. (See Chapter 6.)

Note

The CLOSE procedure is a Prime extension to standard Pascal.

Use of the standard transfer procedures PACK and UNPACK in Prime Pascal will generate an error message and cause your program to fail because PACK and UNPACK are not supported in Prime Pascal. This is a Prime restriction.

FUNCTIONS

Functions are also user-written subprograms. Here are some characteristic traits of functions:

- The keyword FUNCTION is used instead of PROCEDURE.
- Similar to a procedure, a function is a subprogram.
- Unlike procedures and standard functions, the names of user-written functions must represent values. Procedure names and standard function names cannot represent values.
- Unlike a procedure, a data type must be specified for the function itself in the function heading.

A function is an independent program unit that accepts zero or more parameters to produce a single output value. A function must be declared in a function declaration, a forward function declaration, or an external function declaration before the function can be invoked.

Function declarations are discussed below. Forward and external function declarations are discussed later in this chapter.

The external function declaration is a Prime extension to standard Pascal.

Using the -EXTERNAL Option Instead of {$E+}: An alternative to using the {$E+} switch in the subprogram is to use the -EXTERNAL option every time you compile the file of subprograms. For example:

```
PASCAL filename -EXTERNAL
```

The filename is the name of the file that contains the external subprograms. (See Chapter 2 for more information on compiling programs.)

Defining External (Global) Variables with {$E+}: If you want your external subprograms to reference the variables that are declared in the calling program, you must use the {$E+} and {$E-} switches in the VAR declaration of the calling program. For example:

```
VAR
  I, J : INTEGER;
{$E+}
  X, Y, Z : INTEGER;
{$E-}
```

Here is an example of a program that calls an external procedure. It has one variable, ADDSUM, that is used externally:

```
PROGRAM File 1;
VAR
  I, J : INTEGER;
{$E+}
  ADDSUM : INTEGER;
{$E-}
PROCEDURE ADD(A, B : INTEGER); EXTERN;
BEGIN {main program}
  I := 23;
  J := 45;
  ADD(I, J); {external procedure is called here}
  WRITELN(ADDSUM)
END.
```

Here is the external procedure ADD, which the above program calls. Notice that the external variable ADDSUM must also be declared in the subprogram at the top of the file, outside the procedure or function block:

```
{$E+}
VAR
  ADDSUM : INTEGER;
PROCEDURE ADD(A, B : INTEGER);
BEGIN
  ADDSUM := A + B
END;
```

Compiling and Loading Subprograms: Remember that each external subprogram file must be compiled and loaded separately. After you have entered SEG's LOAD subprocessor, the main program must be loaded before the separately compiled subprograms. For more information on compiling, loading, and executing programs, see Chapters 2 and 3.

External subprogram names, as well as the names of main programs, cannot be more than 32 characters long.

Caution

Do not define a main program as external. An error message will result. The following example is invalid:

```
{$E+}
PROGRAM Main;
  .
  .
  .
BEGIN
  .
  .
  .
END.
```

Subprograms Written in Other Languages

Subprograms declared in external procedure or function declarations in the main program can be written in any Prime high-level language or Prime Macro Assembly (PMA) language with certain restrictions:

- There must be no conflict of data types for variables being passed as parameters. For example, a FIXED BINARY(15) in PL/I is equivalent to an INTEGER in Pascal.
- Programs compiled in either 64V or 32I mode cannot reference or be referenced by programs compiled in R mode. Programs in 64V or 32I mode may reference each other.

For more information on interfacing Pascal with other languages, see Appendix D.

10
Input and Output

In Prime Pascal, data can either be input from your terminal or be input from a PRIMOS input data file. Similarly, the output can either be written out to your terminal or to a PRIMOS output data file.

This chapter explains how to input and output data in Prime Pascal, using both of these methods.

Throughout this chapter, various built-in I/O (input/output) functions and procedures that manipulate data are discussed. These include eight file-handling procedures (RESET, GET, READ, READLN, REWRITE, PUT, WRITE, and WRITELN), two BOOLEAN functions (EOF and EOLN) and two auxiliary procedures (PAGE and CLOSE).

Note

Prime Pascal performs I/O operations only on data stored in disk files or data supplied at the terminal.

INPUTTING AND OUTPUTTING DATA AT THE TERMINAL

When you execute a program, and your program requests data at execution time, it can wait for you to input the data at your terminal. For example:

```
PROGRAM Add;
VAR
  A, B, C : INTEGER;
BEGIN
  READLN(A);
  READLN(B);
  C := A + B;
  WRITELN(C)
END.
```

In the example above, the computer expects you to enter two integers at your terminal upon execution. The execution would look like this, where user input is underlined:

```
OK,
SEG ADD
30
50
          80 {computer writes out result here}
OK,
```

For more information on executing programs, see Chapter 3.

If you were using READs instead of READLNs in the example above, you could place the integers on the same line, separated by spaces or a comma. For example, given the following statements:

```
READ(X, Y);
Z := X + Y;
WRITELN(Z);
```

your terminal input and execution would look like this:

```
OK,
SEG ADD
30 50          80
OK,
```

A space placed after the 30 and after the 50 signals the end of each integer. It also tells the computer that each integer has two digits. Notice that with READs, the computer outputs the sum on the same line as your input.

You can make the computer prompt you for input by putting WRITE or WRITELN statements in your program. For example:

```
VAR
  A,B,C : INTEGER;
BEGIN
  WRITELN('Enter two numbers:');
  READLN(A);
  READLN(B);
  C := A+B;
  WRITELN(C)
END.
```

Your input and execution would look like this:

```
OK,
SEG ADD
Enter two numbers:
10
20
        30
OK,
```

If you were using READs on CHAR type data instead of INTEGER or REAL, you would not put spaces between the input characters. Therefore, with the following program:

```
PROGRAM Letters;
VAR
  X, Y, Z : CHAR;
BEGIN
  WRITE('Enter three letters: ');
  READ(X, Y, Z);
  WRITELN(X:10, Y, Z)
END.
```

your input and execution would look like this:

```
OK,
SEG LETTERS
Enter three letters: PQR         PQR
OK,
```

The 10 in the WRITELN statement formats the output so that nine spaces are placed before the P. Notice that the WRITE statement prompts you for input.

Using Erase and Kill Characters

PRIMOS provides two special character functions called erase and kill. The erase character (the double quotation mark) erases the immediately preceding character. For example, if you type 1235 when you wanted to type 1234, you can correct your mistake by typing the double quote followed by the correct input:

```
1235"4
```

The kill character (the question mark) deletes your entire current line. For example, if you mistakenly type this:

```
123456789
```

and were supposed to type this:

```
ABCDEFGHI
```

you can correct your mistake by typing the question mark followed by the correct input:

```
123456789?ABCDEFGHI
```

Note

Your System Administrator may have changed the Prime-supplied erase and kill characters to some other characters. If so, find out what they are. (You can change them yourself, too.)

How to Use Erase and Kill on Terminal Input: Before Rev. 19.1, use of Prime's erase and kill characters on input from the terminal was not possible because each character was assigned to the program as soon as it was typed. Not only was it too late to use an erase or kill character, but also an erase or kill character itself was assigned.

Now you can use the erase and kill characters by using the -INTERACTIVE switch in the RESET statement in your program. For example:

19.1

```
VAR
  I, J : INTEGER;
BEGIN
  RESET(INPUT, '-INTERACTIVE');
  READLN(I);
  READLN(J)
END.
```

The -INTERACTIVE switch is a Prime extension. When this switch is used, you can erase or kill anything on the current line — that is, before you enter a carriage return. The word -INTERACTIVE must be enclosed in single quotes.

> Caution
>
> You can only use READLNs with the -INTERACTIVE switch. Do not use READs. A READ will not work with -INTERACTIVE because a READ, by definition, still assigns a character as soon as it is typed at the terminal, even before the carriage return is hit. An attempt to use READs will generate an error message at runtime.

The RESET statement opens a PRIMOS data file for reading. RESET is usually used to open input data files; however, there are special cases, such as the example above, where RESET is used to manipulate input from the terminal. (RESET is fully discussed later in this chapter.)

The word INPUT in the RESET statement is a standard Pascal textfile identifier. -INTERACTIVE can only be used with the file INPUT. (For more information on the special functions of the file types INPUT and OUTPUT in Prime Pascal, see Chapter 6 and the discussion on data input files later in this chapter.)

How to Turn the -INTERACTIVE Switch Off: Since the -INTERACTIVE feature is a switch, you can turn it on or off within a program. If you want to turn the -INTERACTIVE feature off use the -TTY feature in another RESET statement. For example: 19.1

```
VAR
  A, B, C, D : INTEGER;
BEGIN
  RESET(INPUT, '-INTERACTIVE');
  READLN(A);
  READLN(B);
  RESET(INPUT, '-TTY');
  READ(C);
  READ(D)
END.
```

Use of -TTY lets you go back to inputting data from the terminal in the "normal" way, without the use of Prime's erase and kill characters. The -TTY switch must be used only with the standard file INPUT. (For information on the other uses of -TTY, see the discussion on input data files later in this chapter.)

Prime's -INTERACTIVE extension differs from standard Pascal in the following ways:

- There is no such feature in standard Pascal.
- READs are not allowed when using -INTERACTIVE.

- In standard Pascal, assignments are supposed to be done when a character is typed at the terminal. With the -INTERACTIVE switch, assignments are done only after the carriage return is hit. 19.1

- The erase and kill characters are given special meaning. In standard Pascal, the carriage return is the only special character.

INPUTTING AND OUTPUTTING DATA WITH PRIMOS FILES

In Prime Pascal, data can be input from an input data file. Similarly, the computer can output data to an output data file. These data files are PRIMOS files, similar to the PRIMOS file that contains your program. These PRIMOS files can be placed in any directory that you wish.

Upon execution of your program, the computer opens input and output files, retrieves the data from the input file, performs operations using that data, outputs results into an output file, and closes the input and output files.

Note

If you do not use input and output files, data will be input from and output to the terminal by default.

CREATING AND USING INPUT DATA FILES

When you want to place data in a file to be read and operated on by a program, you can create a new PRIMOS file and type your data into that file, using Prime's line editor, ED, or Prime's screen editor, EMACS. (See the New User's Guide to EDITOR and RUNOFF, the EMACS Primer, or the EMACS Reference Guide.)

Once your data has been typed into the file, you would name the file, as you would name any PRIMOS file.

Opening the Input File

In your program, you must tell the computer that the data your program needs is located in a PRIMOS data file. This is called opening the input file. All input files are opened with Pascal's RESET procedure.

When INPUT is used with a data file, the name of the file must be given as the second parameter in the RESET procedure, as shown above.

If a file is not specified in a READ or READLN statement, the standard textfile INPUT is assumed. For example, the following have the same effect, whether the standard textfile INPUT is a data file or the terminal:

```
READ(INPUT, A);

READ(A);
```

For more information on INPUT, see Chapter 6.

Switching from Standard INPUT File to Terminal

If you open an input data file with the standard textfile INPUT, and want to switch to inputting data from the terminal, use the -TTY switch in another RESET procedure. For example:

```
VAR
  A, B : INTEGER;
  INFILE : FILE OF CHAR;
BEGIN
  RESET(INPUT, 'INDATA');
  READLN(INPUT, A);
  RESET(INPUT, '-TTY');
  READ(B)
END.
```

18.3

The value of A will be read from an input file named INDATA, and the value of B will be read from the terminal. The standard file INPUT is the first parameter with -TTY. The -TTY switch must be enclosed in single quotes.

The -TTY switch also works with REWRITE and the standard textfile OUTPUT.

CREATING AND USING OUTPUT DATA FILES

When you want to write data out to an output file, simply open the file and name it using the REWRITE procedure.

The REWRITE Procedure

The format of the REWRITE procedure statement is:

```
REWRITE(file, 'filename');
```

The first parameter file is a Pascal file variable of a FILE type that is associated with the output file. The second parameter, 'filename' is the actual name of the PRIMOS file. This name must be enclosed in single quotes. The inclusion of the second parameter is a Prime extension.

You do not have to create a PRIMOS output file beforehand. The REWRITE procedure will create a PRIMOS file for you upon execution. For example:

```
PROGRAM Writeout;
VAR
  A, B, C : INTEGER;
  OUTFILE : FILE OF CHAR;
BEGIN
  READLN(A);
  READLN(B);
  C := A + B;
  REWRITE(OUTFILE, 'OUTDATA');
  WRITELN(C);
  CLOSE(OUTFILE)
END.
```

OUTFILE is declared as FILE OF CHAR. A and B are read from the terminal. REWRITE creates a PRIMOS file named OUTDATA in your directory. The value of C is written out to the new file, and the file is closed with CLOSE. (The CLOSE procedure is discussed later in this chapter.)

The second parameter 'filename' can also be a pathname. For example:

```
REWRITE(OUTFILE, 'PAUL>HOMEWORK>OUTDATA');
```

An output file called OUTDATA will be created in the subdirectory HOMEWORK within the directory PAUL.

Note

Be sure to find out what your directory access rights are at your installation.

The use of EOF, as well as RESET, GET, REWRITE, and PUT is illustrated in the following example:

```
VAR
  INFILE, OUTFILE : TEXT;
BEGIN
  RESET(INFILE, 'INDATA');
  REWRITE(OUTFILE, 'OUTDATA');
  WHILE NOT EOF(INFILE) DO
    BEGIN
      OUTFILE^ := INFILE^;
      PUT(OUTFILE);
      GET(INFILE)
    END;
  CLOSE(INFILE);     {The CLOSE procedure is discussed at the end}
  CLOSE(OUTFILE)     {of this chapter.}
END.
```

The EOLN Function: The function EOLN tests for an end-of-line condition in a textfile. It has the form:

```
EOLN(file)
```

This function is true if the buffer variable file^ corresponds to the position of a line separator marking the end of the current line. The line separator is the ASCII character LF (Line feed), which is a carriage return. EOLN is applied to the standard textfile INPUT, if the parameter file is omitted, whether INPUT is a data file or the terminal.

Auxiliary Procedures

There are two auxiliary procedures that manipulate I/O in Prime Pascal — PAGE and CLOSE. The CLOSE procedure is a Prime extension.

The PAGE Procedure: The form of the PAGE procedure is:

```
PAGE(file)
```

The PAGE procedure generates a skip to the top of a new page before the next line of the output textfile file is written. If the single parameter file is omitted, then this procedure is applied to data that is written out to the standard textfile OUTPUT by default, whether OUTPUT is a data file or the terminal.

For example:

```
WRITELN('Page Test');
WRITELN('Page 1');
PAGE;
WRITELN('Page 2');
```

The CLOSE Procedure: All input and output data files must be explicitly closed using the CLOSE procedure. Otherwise they will remain open after the program terminates.

The form of the CLOSE procedure is:

```
CLOSE(file);
```

The CLOSE procedure is a Prime extension to standard Pascal.

For example:

```
VAR
  Fyle: TEXT;
BEGIN
  REWRITE(Fyle, 'FYLE');
  WRITELN(Fyle, 'ABC');
  WRITELN(Fyle, 'DEF');
  CLOSE(Fyle)
END.
```

11
Standard Functions

A standard function, denoted by a standard identifier, is a built-in function supplied by the Pascal language. There are four types of standard functions — arithmetic, transfer, ordinal, and BOOLEAN.

ARITHMETIC FUNCTIONS

ABS(X) Computes the absolute value of X. The type of X must be INTEGER, LONGINTEGER, REAL, or LONGREAL. The type of the result is the same as that of X. | 19.1

SQR(X) Computes the square of X. X and the result will be of the same data type: INTEGER, LONGINTEGER, REAL, or LONGREAL. | 19.1

Note

For the following arithmetic functions, the type of X must be INTEGER, LONGINTEGER, REAL, or LONGREAL. The type of result is always REAL or LONGREAL. | 19.1

SIN(X) Computes the sine of X.

COS(X) Computes the cosine of X.

EXP(X) Computes the value of the base of natural logarithms raised to the power X. This is exponential function (e^x).

LN(X) Computes the natural logarithm of X. X must be greater than zero.

SQRT(X) Computes the non-negative square root of X. X must be non-negative.

ARCTAN(X) Computes the value, in radians, of the arctangent of X.

TRANSFER FUNCTIONS

19.1 TRUNC(X) Truncates a real number into an integer. X must be of type REAL or LONGREAL. The result is of type INTEGER or LONGINTEGER. If X is positive then the result is the greatest integer less than or equal to X; otherwise it is the least integer greater than or equal to X. Examples:

TRUNC(3.7) yields 3

TRUNC(-3.7) yields -3

19.1 ROUND(X) Rounds a real number to the nearest integer. X must be of type REAL or LONGREAL. The result, which is of type INTEGER or LONGINTEGER, is the value X rounded. That is, if X is positive, ROUND(X) is equivalent to TRUNC(X + 0.5); otherwise ROUND(X) is equivalent to TRUNC(X - 0.5). Examples:

ROUND(3.7) yields 4

ROUND(-3.7) yields -4

ROUND(3.2) yields 3

ROUND(-3.2) yields -3

Note

Be careful when the result of your TRUNC or ROUND function is of an INTEGER type. You can assign an INTEGER value to a LONGINTEGER variable without any possible errors, but when you attempt to assign a LONGINTEGER value to an INTEGER variable an 19.1 error is generated. This also applies to REAL and LONGREAL. (See Chapter 6 for more information on LONGINTEGER and LONGREAL.)

BOOLEAN FUNCTIONS

ODD(X)	X must be of type INTEGER or LONGINTEGER. The result is TRUE if X is odd and FALSE otherwise.	19.1
EOF(F)	F is the file variable of an input file. This function returns the value TRUE if an end-of-file condition exists for F and FALSE otherwise. It applies to the standard textfile INPUT if the argument F is omitted.	
EOLN(F)	F is the file variable of an input textfile. This function returns the value TRUE if the end of the current line is reached and FALSE otherwise. It applies to the standard textfile INPUT if F is omitted.	

$ and _ in identifiers (Chapter 4)	Dollar signs and underscores are allowed in identifiers in Prime Pascal. However, the underscore cannot be the first character.
The & and ! integer operators (Chapter 7)	Prime's integer operators & and ! perform Boolean AND and OR operations respectively on decimal integer and longinteger numbers.
The OTHERWISE keyword (Chapter 8)	Prime's OTHERWISE keyword can be used at the bottom of a CASE statement to execute an alternative statement, or group of statements, if no statement in the list of CASE statements has been selected.
The EXTERN attribute (Chapter 9)	When an external, separately compiled subprogram is declared in Prime Pascal, it must be declared with the word EXTERN at the end of the declaration heading.
The {$E} compiler switch (Chapters 2 and 9)	External Pascal subprograms can be separately compiled by including the {$E+} at the beginning of the subprogram file. This switch can also be used in the calling program's variable declarations so that the variables can be referenced by the external subprograms.
The {$A} compiler switch (Chapter 2)	The {$A} switch controls the generation of code used to perform array bounds checking at runtime.
The {$L} compiler switch (Chapter 2)	The {$L} switch controls the printing of source lines to the listing file at compile time, if -LISTING was specified.
The {$P} compiler switch (Chapter 2)	The {$P} switch controls page breaks or page "ejects" in the listing file.

18.3

The second parameter 'filename' in RESET and REWRITE procedures (Chapter 10)	When input or output data files are used, your RESET and REWRITE procedures, which open the files, should have as their second parameter the name of the PRIMOS file that has to be opened for reading or writing. This filename must be enclosed in single quotes. The first parameter is a variable declared as a FILE type, which is associated with the second parameter, 'filename'.
The CLOSE procedure (Chapter 10)	The CLOSE procedure must be used to close an input or output data file after it has been opened with RESET or REWRITE.
The standard data files INPUT and OUTPUT (Chapters 6 and 10)	The standard data files INPUT and OUTPUT, when used in a RESET or REWRITE procedure without the second parameter 'filename' will automatically default to I/O to and from the terminal. If a file is not specified in a READ or READLN statement, the standard textfile INPUT is assumed, whether the standard textfile INPUT is a file or the terminal. This also applies to WRITE, WRITELN, and the standard textfile OUTPUT.

B
Data Formats

OVERVIEW

This appendix illustrates how values of Prime Pascal data types are represented in storage. For more information on all of the data types, see Chapter 6. In Prime Pascal, a word consists of 16 bits.

Prime Pascal supports the following data types:

Scalar Data Types

INTEGER
LONGINTEGER (Prime extension) 19.1
Subrange
REAL
LONGREAL (Prime extension) 19.1
CHAR
BOOLEAN
Enumerated

Structured Data Types

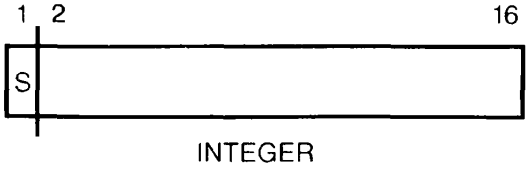
ARRAY
RECORD
SET
FILE

Pointer Data Type

Pointer

INTEGER TYPE DATA

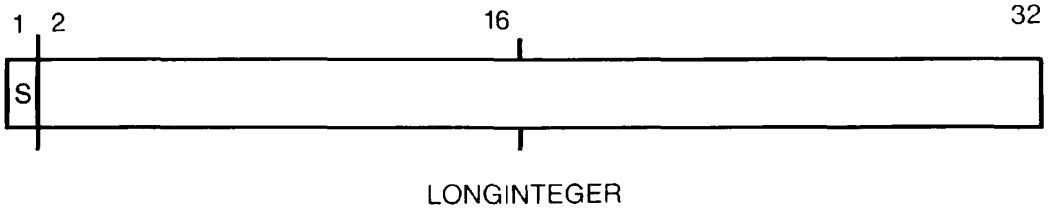
Integers are 16-bit (one word) twos-complement, fixed-point whole binary numbers. Integers can hold values within the range -32768 to +32767. Bit 1 is the sign bit, which indicates whether the integer is positive or negative. Bits 2-16 are the integer itself.

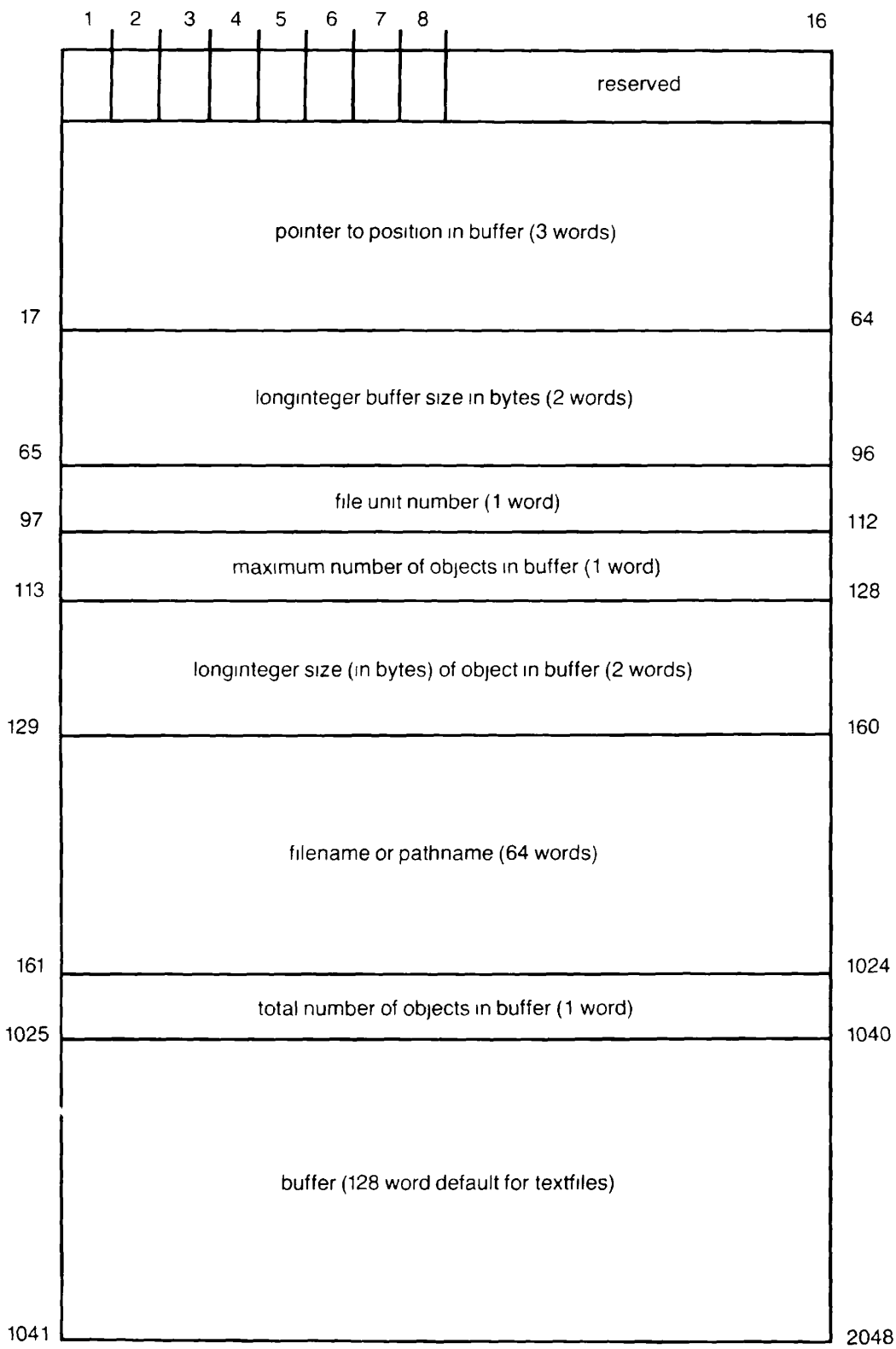


LONGINTEGER TYPE DATA

Longintegers are 32-bit (two-word) twos-complement, fixed-point whole binary numbers that hold values within the range -2147483648 to +2147483647. Bit 1 is the sign bit, which indicates whether the longinteger is positive or negative, and bits 2-32 are the longinteger itself. The LONGINTEGER type is a Prime extension.

19.1





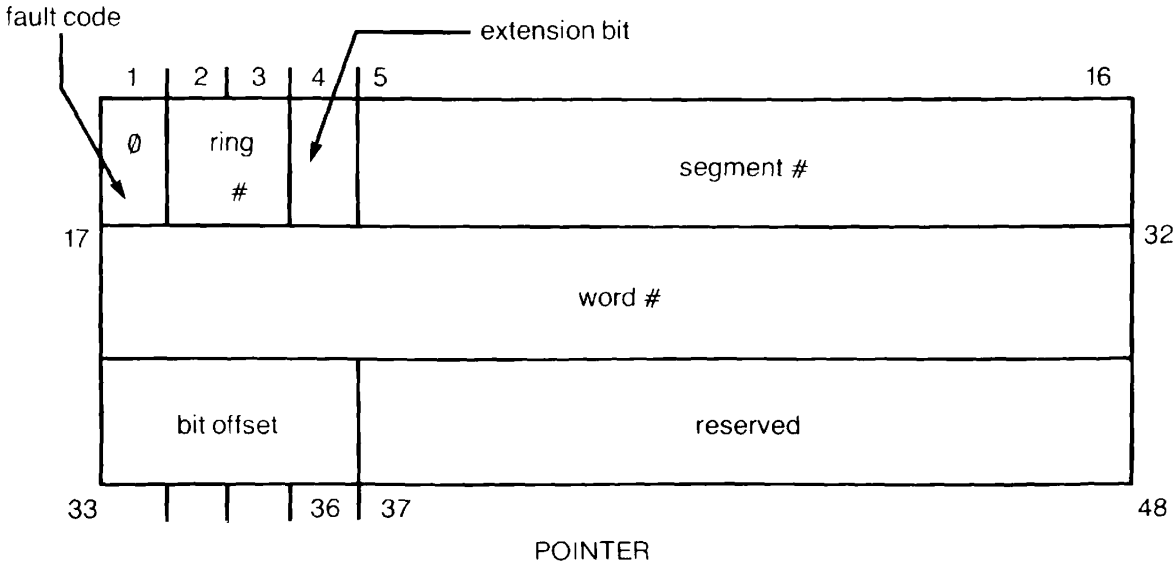
FILE CONTROL BLOCK

POINTER TYPE DATA

Each value of a pointer type variable is the actual address of the data to which each variable is pointing. Therefore the storage area for each pointer variable contains an address.

A pointer is represented in storage by 48 bits (three words). Specifically:

- Bit 1 is the fault code, which determines if the desired data is found or not found.
- Bits 2 and 3 contain the ring number of the data that is being pointed to.
- Bit 4 is the extension bit, which indicates whether the pointer contains a bit offset (three-word pointer) or doesn't contain a bit offset (two-word pointer).
- Bits 5-16 contain the segment number of the data.
- Bits 17-32 contain the word number of the data within that segment.
- Bits 33-36 are the bit offset, which allows the pointer to point to any bit in memory.
- Bits 37-48 are reserved for future storage.



POINTER

D

Interfacing Pascal to Other Languages

OVERVIEW

This appendix offers guidelines for interfacing Pascal data types with compatible data types of other Prime languages.

The key to interfacing compatible data types is storage representation. For example, a Pascal INTEGER value and a PL/I Subset G Fixed Bin(15) value are both stored as 16-bit (one-word) whole binary numbers. Therefore, an INTEGER value can be passed to a Fixed Bin(15) value and vice versa. In order to interface Pascal to another language successfully, you should be familiar with how Prime Pascal data types are represented in storage. (See Appendix B.) You should also be familiar with the other Prime language and how data types of that language are represented in storage.

Table D-1 matches the compatibility of Prime Pascal data types with the data types of Prime's PL/I Subset G, FORTRAN 77, FORTRAN IV, COBOL, and BASIC/VM. The leftmost column lists the generic storage unit, which is measured in bits, bytes, or words for each data type. Each storage unit matches the data types listed to the right on the same row. Following Table D-1, this appendix briefly discusses data type compatibility and includes several program examples.

For more information on interfacing Pascal to other languages, as well as calling Prime's standard subroutines, see the Subroutines Reference Guide.

Table D-1
Compatible Data Types

GENERIC UNIT/PMA	BASIC/VM	COBOL	FORTRAN IV	FORTRAN 77	PASCAL	PL/I SUBSET G
1 bit	—	—	—	—	—	Bit Bit(1)
16 bits (one word)	INT	COMP	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2	INTEGER BOOLEAN ENUMERATED	Fixed Bin Fixed Bin(15)
32 bits (two words)	INT*4	—	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	LONGINTEGER	Fixed Bin(31)
64 bits (four words)	—	—	—	—	—	—
32-bit Float single precision	REAL	—	REAL REAL*4	REAL REAL*4	REAL	Float Binary Float Bin(23)
64-bit Float double precision	REAL*8	—	REAL*8	REAL*8	LONGREAL	Float Bin(47)
Byte string (Max. 32767)	INT	DISPLAY(5) PIC A(n) PIC 9(n) PIC X(n)	INTEGER	CHARACTER *n	CHAR ARRAY [1..n] OF CHAR	Char(n)
Varying character string	—	—	—	—	*	Char(n) Varying
48-bits (three words)	—	—	—	—	^<type>	Pointer
256 bits	—	—	—	—	SET	Bit(256)

— Not available.

* See Subroutines Reference Guide

Index