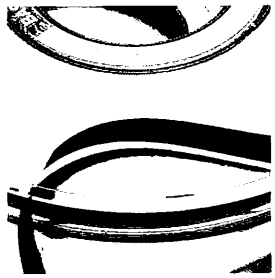
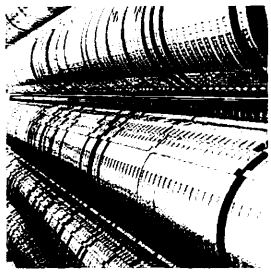
Prime Computer, Inc.

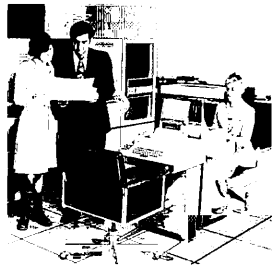
DOC 4303-191L
Pascal Reference Guide
Revision 19.2





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

Updated for Software Release 19.2

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 19.2 (Rev. 19.2).

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

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
Udpate 2	July 1982	PTU26700-086	19.0
Second Edition	December 1982	DOC4303-191	19.1
Update 1	June 1983	UPD4303-192	19.2

The Second 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.

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

ABOUT THIS BOOK xi

PART I - OVERVIEW

1 INTRODUCTION TO PRIME PASCAL

The Pascal Language 1-2
Prime Pascal 1-2
Contents of This Book 1-2
Related Documents 1-4
Interface to Other Languages 1-6

PART II - COMPILING, LOADING, AND EXECUTING PROGRAMS

2 USING THE PASCAL COMPILER

Introduction 2-1
Invoking the Compiler 2-2
Compiler Error Messages 2-2
Filename Conventions 2-4
Compiler Options 2-6
Compiler Option Abbreviations 2-13
Compiler Switches 2-16

3 LOADING AND EXECUTING PROGRAMS

Loading Programs 3-1
Executing Programs 3-7

PART III - PRIME PASCAL LANGUAGE REFERENCE

4 PASCAL LANGUAGE ELEMENTS

Definitions 4-2
Pascal Character Set 4-4
Keywords 4-7
Identifiers 4-7

Numeric Constants 4-8
Character-strings 4-11
Declarations and Statements 4-11
Line Format 4-11
Comments, Blanks, and Ends of Lines 4-11

5 PASCAL PROGRAM STRUCTURE

Program Heading 5-1
The Block 5-3
Declaration Part 5-4
LABEL 5-4
CONSTANT 5-6
TYPE 5-6
VARIABLE 5-7
PROCEDURE and FUNCTION 5-9
Executable Part 5-9
A Program Example 5-11

6 DATA TYPES

Scalar Data Types 6-1
Standard Scalar Data Types 6-2
INTEGER 6-3
LONGINTEGER 6-4
REAL 6-6
LONGREAL 6-7
BOOLEAN 6-8
CHAR 6-8
User-defined Scalar Data Types 6-10
Enumerated 6-10
Subrange 6-12
Structured Data Types 6-14
19.2| The STRING Type 6-14
The ARRAY Type 6-14
The RECORD Type 6-20
The SET Type 6-25
The FILE Type 6-27
TEXT 6-29
The Pointer Type 6-31

7 EXPRESSIONS

Operands 7-1
Operators 7-2
Arithmetic Operators 7-2
Relational Operators 7-3
SET Operators 7-5
BOOLEAN Operators 7-6
Integer Operators 7-7
19.2| STRING Concatenation Operator 7-7
Operator Precedence 7-7

8 STATEMENTS

Summary of Statements	8-1
Assignment Statement	8-2
Procedure Statement	8-3
Compound Statement	8-4
Empty Statement	8-5
Control Statements	8-5
Repetitive Statements	8-6
REPEAT	8-6
WHILE	8-7
FOR	8-8
Conditional Statements	8-10
IF	8-10
CASE	8-11
Unconditional Statement	8-14
GOTO	8-14
WITH Statement	8-16

9 PROCEDURES AND FUNCTIONS

Parameters	9-2
Procedures	9-9
Functions	9-12
Forward Procedures and Functions	9-14
External Procedures and Functions	9-15
Recursive Procedures and Functions	9-19

10 INPUT AND OUTPUT

Inputting and Outputting Data at the Terminal	10-2
Inputting and Outputting Data with PRIMOS Files	10-6
Creating and Using Input Data Files	10-6
The RESET Procedure	10-7
Creating and Using Output Data Files	10-11
The REWRITE Procedure	10-11
I/O Procedures and Functions	10-14
Input File-handling Procedures	10-15
GET	10-15
READ	10-15
READLN	10-17
Output File-handling Procedures	10-18
PUT	10-18
WRITE	10-18
WRITELN	10-22
BOOLEAN Functions	10-22
EOF	10-22
EOLN	10-23

Auxiliary Procedures 10-23
PAGE 10-23
CLOSE 10-24

11 STANDARD FUNCTIONS

Arithmetic Functions 11-1
ABS 11-1
SQR 11-1
SIN 11-1
COS 11-1
EXP 11-2
LN 11-2
SQRT 11-2
ARCTAN 11-2
Transfer Functions 11-2
TRUNC 11-2
ROUND 11-2
Ordinal Functions 11-3
ORD 11-3
CHR 11-3
SUCC 11-3
PRED 11-4
BOOLEAN Functions 11-5
ODD 11-5
EOF 11-5
EOLN 11-5
19.2 | STRING Functions 11-5

APPENDIXES

A SUMMARY OF PRIME EXTENSIONS AND RESTRICTIONS

Prime Extensions A-1
Prime Restrictions A-5

B DATA FORMATS

Overview B-1
INTEGER Type Data B-2
LONGINTEGER Type Data B-2
Subrange Type Data B-3
REAL Type Data B-3
LONGREAL Type Data B-3
CHAR Type Data B-4
BOOLEAN Type Data B-4
Enumerated Type Data B-4
ARRAY Type Data B-5
RECORD Type Data B-5
SET Type Data B-5

FILE Type Data B-6
Pointer Type Data B-8
STRING Type Data B-9 |19.2

C ASCII CHARACTER SET

Prime Usage C-1
Special Characters C-2
Keyboard Input C-2

D INTERFACING PASCAL TO OTHER LANGUAGES

Overview D-1
Interfacing INTEGER, BOOLEAN, and Enumerated D-3
Interfacing LONGINTEGER D-4
Interfacing REAL D-4
Interfacing LONGREAL D-5
Interfacing CHAR and ARRAY OF CHAR D-5
Interfacing Pointer D-6
Interfacing SET D-7
Interfacing RECORD D-7
Interfacing STRING D-9 |19.2

INDEX X-1

About This Book

This book is a reference guide to the Pascal language as implemented on Prime computers. It documents Prime's Pascal compiler, along with Prime's extensions and restrictions to standard Pascal. The generic term Prime Pascal refers to the way standard Pascal is implemented on Prime computers, including all Prime's extensions and restrictions.

You are expected to be familiar with the Pascal language, and with programming in general, but not necessarily with Prime computers. For example, if you are a programmer at an installation that uses Prime Pascal, or if you are a Pascal instructor or student at a university that uses Prime Pascal, this book would be particularly useful.

HOW TO USE THIS BOOK

This book is divided into three parts:

- Part I — Overview (Chapter 1)
- Part II — Compiling, Loading, and Executing Programs (Chapters 2 and 3)
- Part III -- Prime Pascal Language Reference (Chapters 4-11)

Four appendixes and an index follow Chapter 11.

If you are already familiar with Prime Pascal, but want to brush up on using the Prime system to compile, load, and execute programs, read Part II first.

If you are not familiar with Prime Pascal, turn to Part I for a detailed chapter-by-chapter description of what this book contains. Part I also lists several other Prime documents that you will need in conjunction with the Pascal Reference Guide.

After reading Part I, turn to the chapters in Part III that you think will help you become familiar with Prime Pascal. Chapters 4, 5, and 10 -- Pascal language elements, program structure, and input/output -- are good places to start. You should also read Appendix A, which summarizes the differences between Prime Pascal and standard Pascal (Prime extensions and restrictions).

Change bars in the margins reflect changes made to the text since the first edition, which was published at software release 17.6 (Rev. 17.6). Change bars with numbers indicate technical changes and the software release of those changes. Change bars without numbers indicate rewrites for clarifications or additional information.

DOCUMENTATION CONVENTIONS

The following conventions are used in command formats, statement formats, in examples, and in the documentation in general.

Convention	Explanation	Examples
underlining in examples of computer-user dialog	In examples of computer-user dialog, user input is underlined and system output is not.	OK, SEG -LOAD [SEG rev 19.1] $ LOAD TEST
UPPERCASE	Words in uppercase signify Pascal standard identifiers, keywords, commands, compiler options, and data types. However, any of these can be typed in uppercase or lowercase.	The REAL type The ORD function LOAD The -DEBUG option

UPPERCASE in program examples	Examples of Pascal code appear in uppercase for consistency. However, programs can be typed in uppercase or lowercase.	WHILE NOT EOLN DO
		  BEGIN
		  READ(A);

lowercase	In command formats, words in lowercase indicate items for which you must substitute a suitable value.	LOAD filename
Brackets []	In command, option, or statement formats, brackets enclose a list of one or more of these items. Choose none, one, or more of these items. (Do not confuse these brackets with array index or set brackets.)	-LISTING [argument]
Ellipsis ...	In command and statement formats, and in program examples, an ellipsis indicates the preceding item may be repeated, or that there are more statements to be processed.	statement-1... statement-n
Parentheses ()	When parentheses appear in a statement format, they must be included literally when a statement is used.	RESET (file,'filename');
Hyphen -	Whenever a hyphen appears in a command line option, it is a required part of that option.	PASCAL TEST -XREF
Vertical slash |	In command or statement formats, vertical slashes indicate a choice of one item or another.	READLN (file|variable);

PART I

Overview

1

Introduction to Prime Pascal

This document is a programmer's reference guide to the Pascal language as implemented on Prime computers.

You are expected to be familiar with the Pascal language, and with programming in general, but not necessarily with Prime computers. If you are unfamiliar with the language, there are many commercially available instruction books, such as:

Cherry, G., Pascal Programming Structures, Reston Publishing Co., Inc., New Jersey, 1980.

Cooper, Doug and Clancy, Michael, Oh! Pascal!, W. W. Norton & Company, New York and London, 1982.

Jensen, Kathleen and Wirth, Niklaus, PASCAL User Manual And Report, Second Edition. Springer-Verlag, New York, 1978.

Schneider, G., Weingart, S. and Perlman, D., An Introduction To Programming And Problem Solving With PASCAL, John Wiley & Sons, Inc., New York, 1978.

THE PASCAL LANGUAGE

Pascal is a multipurpose structured programming language that can be used for system, commercial, and scientific data processing. Pascal is also used as the principal instructional language in many educational institutions. This language, developed in 1968 by Professor Niklaus Wirth at the Eidgenossische Technische Hochschule (ETH) in Zurich, Switzerland, is a descendent of the language ALGOL-60. Pascal is named for the French mathematician Blaise Pascal.

PRIME PASCAL

Prime Pascal refers to the way standard Pascal is implemented on Prime computers, including all of Prime's enhancements and limitations to standard Pascal. In this book, when the Pascal language is mentioned, it refers to Prime Pascal as a whole.

Prime Extensions and Restrictions

Prime Pascal varies from standard Pascal in several ways. Prime has created many enhancements, which are commonly called extensions, as well as limitations, which are called restrictions. Throughout this book, Prime's extensions and restrictions are clearly identified when they are discussed.

Appendix A lists Prime extensions and restrictions, along with the chapter in which each is discussed.

CONTENTS OF THIS BOOK

The following is a brief chapter-by-chapter description of the contents of this book.

Part I -- Overview

- Chapter 1 contains a brief introduction to the Pascal language as implemented on Prime computers.

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 three Prime extension data types called LONGINTEGER, LONGREAL, and STRING. | 19.2
- 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.

EMACS Primer and EMACS Reference Guide

Prime's screen editor, EMACS, can also be used to input and modify new programs. The Primer is designed for users who do not know EMACS. The reference guide is a quick reference for users already familiar with EMACS.

LOAD and SEG Reference Guide

Ordinarily, to load and execute programs you need only the information given in the Pascal Reference Guide or the Prime User's Guide. If you wish to control the load process in more detail, or use the full range of Prime loader capabilities, see the LOAD and SEG Reference Guide.

Subroutines Reference Guide

Prime offers a large selection of applications-level subroutines and PRIMOS operating system subroutines, which can be declared as external in procedure/function declarations of a Pascal program, then referenced from any point within the program. These routines are described in the Subroutines Reference Guide. (See also Chapter 9 of this guide.)

Source Level Debugger Guide

When you specify the -DEBUG option at compile time, you can generate code that can be used to debug your program with Prime's debugger utility, DBG. For complete information, consult the Source Level Debugger Guide.

INTERFACE TO OTHER LANGUAGES

Since all Prime high-level languages are alike at the object-code level, and since all use the same calling conventions, object modules produced by the Pascal compiler can reference and be referenced by modules produced by the FORTRAN 77, FORTRAN IV, COBOL, PL/I Subset G, etc. compilers, provided that certain restrictions are observed:

- All I/O routines must be written in the same language. However, Pascal I/O should be used if and only if the main program is written in Pascal.

- There must be no conflict of data types for variables being passed as arguments. For example, an INTEGER in Pascal should be declared as FIXED BINARY(15) in PL/I-G. See Appendix B for a description of Pascal data storage formats and Appendix D for data type compatibility.

- Modules compiled in 64V or 32I mode cannot reference or be referenced by modules compiled in R mode. Modules in 64V or 32I mode may reference each other if they are otherwise compatible.

Pascal program units can also reference PMA (Prime Macro Assembler) routines, and vice versa. For information, see the Assembly Language Programmer's Guide.

Appendix D offers guidelines for interfacing Pascal with other languages.

PART II

Compiling, Loading, and Executing Programs

2
Using the Pascal Compiler

INTRODUCTION

Prime's Pascal compiler, like Prime's other high-level compilers, can output an object (binary) file, a source listing file, error and statistical information, and other useful messages and information. Upon compilation, error messages are printed at the terminal as the compiler encounters them. Your program is checked syntactically, according to the rules of Prime Pascal. Prime extensions and restrictions, which are listed in Appendix A, will be identified when discussed in this chapter and throughout the book.

This chapter discusses:

- How to invoke the compiler
- How to specify options to the compiler
- The significances of various messages that are printed during compilation
- Filename conventions | 18.0
- The meanings of compiler options
- How to specify switches to the compiler
- The meanings of compiler switches

INVOKING THE COMPILER

The Pascal compiler is invoked from PRIMOS command level with the PASCAL command:

```
PASCAL pathname [-option 1] [-option 2] ... [-option n]
```

pathname is the pathname of the Pascal source program to be compiled.

options are the names of various compiler functions, which you can invoke on the command line to provide valuable information and input while you compile, load, and execute your program. Every option name must begin with a hyphen. For example:

```
PASCAL TEST -RANGE -LISTING
```

Given this command, the compiler will check for out-of-bounds values of array subscripts and generate a source listing file for the program TEST.

COMPILER ERROR MESSAGES

If the compiler finds no syntax errors in your program's code, it will tell you there are no errors after it has successfully compiled the program.

For example:

```
OK, PASCAL TEST
[PASCAL Rev. 19.1]
0000 ERRORS (PASCAL-REV 19.1)
```

However, for every error encountered in the program, an error message will automatically be printed at the terminal and in the source listing if a listing is being generated. The general format of an error message is:

```
line-number     line-of-code
                ^

ERROR xxx SEVERITY y BEGINNING ON LINE line-number
explanation
```

line-number	The number of the line where the error occurred
line-of-code	The erroneous line of code
xxx	The error code number
y	Severity code number
explanation	Description of the error and possible remedies

The circumflex (or arrow) that appears just above the error message points to the actual erroneous 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:

```
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 a file that is inserted into 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.
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. 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.

The Suffix Filename Conventions

With the suffix method, files are easily identified and created according to the type of suffix appended to the program name. The source file has a .PASCAL suffix, the object file has a .BIN suffix, and the listing file has a .LIST suffix. If your program is called TEST, you should name the source file:

```
TEST.PASCAL
```

Upon compilation, the compiler creates an object file called TEST.BIN and a source listing file called TEST.LIST. The compiler will put these files in the directory to which you are attached.

When the file is loaded into memory for execution, Prime's loader, SEG, creates an executable file called TEST.SEG. (For information on loading and executing Pascal programs, see Chapter 3.) 18.0

When you compile your program called TEST.PASCAL, you do not have to include the .PASCAL suffix. The command:

```
OK, PASCAL TEST
```

and the command:

```
OK, PASCAL TEST.PASCAL
```

will produce the same result.

For more information on the suffix filename conventions, see Chapter 3.

The Prefix Filename Conventions

If you do not have a Rev. 18 (or higher) system, or your installation does not use the suffix method, you would use the old-style prefix filename conventions. The prefix method identifies the object, listing, and executable files with prefixes. If the source file is named TEST, the compiler will create an object file called B_TEST and a listing file called L_TEST.

When the program is loaded for execution, Prime's loader, SEG, creates an executable file, which you would have named #TEST, for example. The prefix method is less efficient than the suffix method because you, not SEG, name the executable file. (For more information on loading and executing programs, see Chapter 3.)

If you are using the prefix method and your source file is called TEST, compile it by simply issuing the command:

```
OK, PASCAL TEST
```

For more information on the prefix filename conventions, see Chapter 3.

COMPILER OPTIONS

Prime's Pascal compiler offers several compiler functions, or options, that can provide useful information while you compile, load, and execute your program. For instance, you can debug your program with Prime's source level debugger, DBG, by specifying the -DEBUG option. The -XREF option provides a list of all your program's variables and the number of every line on which each variable is referenced.

Options are invoked on the PASCAL command line and may be given in any order. For example:

```
PASCAL TEST -DEBUG -XREF

PASCAL TEST -XREF -DEBUG
```

Most compiler options come in pairs. For each option, there is an option having the opposite effect. Most option pairs direct the compiler to do or not do some action. A few present a choice between two actions. One member of each pair is always the default. For example, consider this pair of options:

```
-DEBUG
-NODEBUG
```

-NODEBUG is the default. Unless -DEBUG is specified, code for the source level debugger will not be generated.

Note

Pascal compiler option defaults are set in a special "driver program" on the Master Disk. At some point, the users at your installation might want to change a default. Your System Administrator can change defaults by following the procedure in the System Administrator's Guide. It is recommended that only your System Administrator have access to the driver files on the Master Disk.

Table 2-1 lists options that are commonly used and not commonly used. For each pair of options in Table 2-1, the Prime-supplied default is underlined. Some options have an argument in addition to the option specification. The argument follows the option and is not preceded by a hyphen. For example:

```
-BINARY NO
```

A list of option pairs, along with detailed descriptions, follows Table 2-1.

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, -NOOPT1, -OPT3, -NOOPT3, and -NOOPTIMIZE	-PRODUCTION and -NOPRODUCTION
-RANGE and -NORANGE	-SILENT and -NOSILENT
-UPCASE	-SOURCE pathname
-XREF and -NOXREF	-STANDARD and -NOSTANDARD
	-STATISTICS and -NOSTATISTICS

19.2

▶ -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.

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

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

▶ -ERRTTY and -NOERRTTY

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

▶ -EXPLIST and -NOEXPLIST

-EXPLIST inserts a pseudo-assembly code listing into the source listing. Each statement in the source will be followed by the pseudo-PMA (Prime Macro Assembler) statements into which it was compiled. For information on PMA, see the Assembly Language Programmer's Guide.

-NOEXPLIST causes no assembler statements to be printed to the listing file.

▶ -EXTERNAL and -NOEXTERNAL

-EXTERNAL creates an object file that can be linked to from other procedures and functions. This option is similar to the $E+ compiler switch, except that -EXTERNAL cannot be suppressed or resumed during compilation. ($E+ switch is discussed at the end of this chapter.)

-NOEXTERNAL causes no external procedure definitions to be generated.

▶ -FRN and -NOFRN

These options control generation of floating-point round instructions.

-FRN causes an FRN instruction to be generated before every FST (floating store) instruction in the code produced by the Pascal compiler. For explanations of these instructions, see the Assembly Language Programmer's Guide. The FRN option improves the accuracy of single-precision, floating-point calculations at some runtime performance expense.

-NOFRN will cause no FRN instructions to be generated before FSTs.

▶ -INPUT pathname

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

```
PASCAL -INPUT 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.

▶ -LISTING [argument]

The -LISTING option controls creation of the source listing file. The argument may be:

pathname	Listing will be written to the file pathname.
YES	Listing will be written to the file named program.LIST, or L_program, in the user's UFD, where program is the name of the source file.
TTY	The listing will be printed at the user terminal.
SPOOL	The listing will be spooled directly to the line printer. Default SPOOL arguments are in effect.
NO	No listing file will be generated.

When no -LISTING option is given, -LISTING NO will be the default. When -LISTING is given with no argument, -LISTING YES will be the default.

▶ -MAP and -NO_MAP

-MAP is the default and makes no user-visible changes to the listing file. The "map" of variables is included in the listing file. If -MAP
18.3 is specified, a listing file will be generated.

The -NO_MAP option generates a listing file that includes only the program and error messages without a "map" of variables and their locations in memory.

▶ -OFFSET and -NOOFFSET

-OFFSET appends an offset map to the source listing. For each statement in the source program, the offset map gives the hexadecimal offset in the object file of the first machine instruction generated for that statement.

-NOOFFSET causes no offset map to be created.

▶ -OPTIMIZE, -OPT1, -NOOPT1, -OPT3, -NOOPT3, and -NOOPTIMIZE | 19.2

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. -NOOPT1 is the default. | 19.2

The -OPT3 option optimizes more code and generates more efficient code than -OPTIMIZE, but compilation time is slower than -OPTIMIZE. -NOOPT3 is the default. | 19.2

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.

▶ -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
-NOOPT1	-NOOPT1	Don't optimize less code
-NOOPT3	-NOOPT3	Don't optimize more code

19.2

Table 2-2 (continued)
Summary of Compiler Options and Abbreviations

Option	Abbreviation	Significance	
-NOPRODUCTION	-NOP	Don't generate production code	
-NORANGE	-NOR	Don't check subscript ranges	
-NOSILENT	-NOSI	Don't suppress severity 1 error messages	
-NOSTANDARD	-NOSTAN	Don't flag nonstandard Pascal syntax	
-NOSTATISTICS	-NOSTAT	Don't print compiler statistics	
-NOXREF	-NOX	Don't generate cross-reference	
-OFFSET	-OF	Append offset map to source listing	
-OPT1	-OPT1	Optimize less object code	19.1
-OPTIMIZE	-OP	Optimize object code	
-OPT3	-OPT3	Optimize more object code	19.1
-PRODUCTION	-P	Generate production code	
-RANGE	-R	Generate code to check subscript ranges	
-SILENT	-SI	Suppress severity 1 error messages	
-SOURCE	-S	Designate source file	
-STANDARD	-STAN	Flag nonstandard Pascal syntax	
-STATISTICS	-STAT	Print compiler statistics	
-UPCASE	-UP	Map lowercase variables to uppercase	18.2
-XREF	-X	Generate cross-reference	
-32I	-3	Produce 32I mode code	
-64V	-6	Produce 64V mode code	

COMPILER SWITCHES

Some compiler functions can be controlled through the use of compiler switches specified within the source program.

18.3 A compiler switch is written as a comment -- text enclosed in the comment delimiters (* *) or {} or /* */ -- with a dollar sign as the first character. Immediately following the $, a letter designates the specific switch. A + or - sign thereafter indicates the turning on or off of the switch. This format up to and including the + or - sign must be followed strictly, or the switch will be ignored by the compiler. Any note, if desired, may be written after the + or - sign and before the final comment delimiter. Examples:

```
{$L+}

18.3 {$P+}

/*$A - Compiler will ignore this switch because space precedes -.*/

{$A- Compiler recognizes this switch.}

(*$E+, $L- Compiler will only recognize the first switch.*)

(*$E+*) (*$L- Compiler will recognize both E+ and L- switches.*)
```

Multiple switches, written as separate comments, can be used to control the compilation of a specific part of a program.

The available compiler switches and their meanings are as follows:

Switch	Meaning	Default
A	Controls the generation of code used to perform array bounds checking at runtime. A- suppresses the generation; A+ resumes it.	A-
L	Controls the printing of source lines to the listing file at compile time if a listing file was requested. L- suppresses the printing of source lines (source text); L+ resumes it, assuming that a listing file was requested.	L+

Switch	Meaning	Default	
E	Controls the definition of globally defined procedures and variables (also called common blocks). Pascal procedures/ functions can be separately compiled by including {$E+} at the beginning of the module. (A detailed discussion is presented in Chapter 9.)	E-	
P	Controls page breaks (or page "ejects") in the listing file. {$P+} causes the printing of lines in the listing file to stop on the current page (at the line just above {$P+}) and to resume printing at the top of the next page. The default P- causes continuous printing of lines from page to page.	P-	18.3

3
Loading and Executing Programs

The PRIMOS SEG utility, and SEG's subprocessor LOAD, load and execute all Pascal programs. This chapter provides you with enough knowledge to begin loading and executing programs. Loading is described in more detail in the Prime User's Guide. For extended loading features, as well as a complete description of all SEG commands, including those for advanced system-level programming, see the LOAD and SEG Reference Guide and its updates.

LOADING PROGRAMS

When a Pascal program is loaded for execution, the following files and libraries should be loaded in this order with the SEG utility:

1. The program's object (binary) file, which is created upon compilation. (See Chapter 2 for information on compiling Pascal code.)

2. The object files of separately compiled subprograms, if any. The subprograms should be loaded in the order in which they are called by the main program.

3. The Pascal library.

4. Other Prime libraries, if needed.

5. The standard system libraries.

When all of these items are loaded and all external calls are resolved, SEG returns a LOAD COMPLETE message. Your program is then ready for execution.

Here is a simple example of loading a Pascal program called TEST.PASCAL. User input is underlined:

```
OK, SEG -LOAD          Enter SEG's LOAD subprocessor.
[SEG rev 19.1]
$ LOAD TEST.BIN        Load the main program's object file.
$ LIBRARY PASLIB       Load the Pascal library.
$ LIBRARY              Load the standard system libraries.
LOAD COMPLETE          Loader indicates load is complete.
$ QUIT                 Save the executable file and return to PRIMOS.
OK,
```

In the above example, SEG's LOAD and LIBRARY commands load the main program and the libraries. The QUIT command saves your executable file, or SEG file, and returns you to PRIMOS command level.

The Procedure to Be Followed

The SEG loader is invoked with the SEG command. Depending on how new your compiler is and what your system's restrictions are, you would either use the new, more efficient (Rev. 18 and higher) suffix conventions, or the old-style prefix conventions.

If you are using the suffix method — that is, if your source filename has a .PASCAL suffix as explained in Chapter 2 -- use the -LOAD option on the command line to enter LOAD's subprocessor. For example:

```
SEG -LOAD
```

Pre-Rev. 18 systems do not support the -LOAD option.

With the suffix method, SEG automatically generates a suffixed executable file. For example, if your object file is called TEST.BIN, SEG produces an executable file named TEST.SEG. The -LOAD option automatically enters the LOAD subprocessor and sets up an executable file. This eliminates a step in the load procedure.

Note

The prefix loading method is still available for use on all systems. If you do not have a Rev. 18 (or higher) compiler, or your installation does not use the suffix method, your source filename need not have a .PASCAL suffix. The procedure for loading programs with the prefix method is listed later in this chapter.

The Suffix Loading Procedure: If you use the suffix method, follow this procedure to load programs:

1. Invoke the SEG loader with the SEG -LOAD command. You will enter the LOAD subprocessor, and LOAD's $ prompt symbol will appear. (SEG's prompt symbol is #, but it does not appear when you use the -LOAD option on the command line.)

18.0

2. Load the program's object file with the LOAD command (abbreviated LO). For example:

   $ LO TEST

   It is not necessary, though it is acceptable, to load TEST.BIN instead of TEST.

3. Load the object files of any separately compiled subprograms with the LOAD command in the order in which they are called by the main program.

4. Load the Pascal library, PASLIB, with the subprocessor's LIBRARY command (abbreviated LI):

   $ LI PASLIB

5. Load other Prime libraries, if needed, such as the sort library, VSRTLI, or the applications library, VAPPLB. For example:

   $ LI library-name

6. Load the standard system libraries:

   $ LI

7. At this point, you should receive a LOAD COMPLETE message. If you do not receive the message, use the MAP 3 or MAP 6 command to identify the modules that were not loaded, and load them. If the unidentified module is caused by a misspelled subprogram name, repeat the load. In the unlikely event that some other SEG error messages appear, see the LOAD and SEG Reference Guide.

8. The QUIT command (abbreviated Q) saves the executable file, exits the SEG utility, and returns you to PRIMOS command level. Use QUIT if you do not want to execute the program immediately from inside the subprocessor. (Execution is discussed later in this chapter.)

Here is an example of a user's dialog with the SEG utility during a load procedure using the suffix method:

```
OK, SEG -LOAD              Enter SEG's LOAD subprocessor.
[SEG rev 19.1]
$ LO TEST                  Load the main program's object file.
$ LO subprogram-name       Load separately compiled subprograms.
$ LI PASLIB                Load the Pascal library.
$ LI library-name          Load other Prime libraries, if needed.
$ LI                       Load the standard system libraries.
LOAD COMPLETE              Loader indicates load is complete.
$ QUIT                     Save executable file and return to PRIMOS.
OK,
```

After your load is complete, you should have the following suffixed files in your directory:

- TEST.PASCAL (the source file)
- TEST.BIN (the object file)
- TEST.SEG (the executable file)
- TEST.LIST (the source listing file, if you created one with the -LISTING option at compile time)

The Prefix Loading Procedure: If you do not have a Rev. 18 (or higher) system, or if you do not use the suffix method, you would use the old-style prefix method. You cannot use the -LOAD option on a pre-Rev. 18 system, and the source filename need not have a suffix. A typical source filename would be TEST, not TEST.PASCAL.

With the prefix method, filenames are identified by the attached prefixes. For example, the object file is called B_TEST, not TEST.BIN, the listing file is called L_TEST, not TEST.LIST, and the executable file is called #TEST, not TEST.SEG. You, not SEG, enter the LOAD subprocessor and set up an executable file by issuing the LOAD command followed by the name of the executable file, #TEST. For example:

```
OK, SEG
[SEG rev 17.5]
# LO #TEST
$
```

Note that the SEG prompt # appears without the -LOAD option on the command line.

After you have entered the LOAD subprocessor and set up an executable file, and the subprocessor's $ prompt appears, you must load the object file, the object files of separately compiled subprograms, the Pascal library, and the standard system libraries. For example:

```
$ LO B_TEST
$ LO B_subprogram-name
$ LI PASLIB
$ LI
```

A step-by-step procedure follows:

1. Invoke the SEG utility with the SEG command. SEG's # prompt will appear.

2. Enter SEG's LOAD subprocessor and set up an executable file with the LOAD command (abbreviated LO) followed by the program name with a # prefix:

   ```
   # LO #TEST
   ```

3. Load the main program's object file (B_TEST) after the subprocessor's $ prompt appears. For example:

   ```
   $ LO B_TEST
   ```

4. Load the object files of any separately compiled subprograms (B_subprogram-name) in the order in which they are called by the main program.

5. Load the Pascal library, PASLIB, with the subprocessor's LIBRARY command (abbreviated LI):

   ```
   $ LI PASLIB
   ```

6. Load other Prime libraries, if needed, such as the sort library, VSRTLI, or the applications library, VAPPLB. For example:

   ```
   $ LI library-name
   ```

7. Load the standard system libraries:

   ```
   $ LI
   ```

8. At this point, you should receive a LOAD COMPLETE message. If you do not receive the message, use the MAP 3 or MAP 6 command to identify the modules that were not loaded, and load them. If the unidentified module is caused by a misspelled subprogram name, repeat the load. In the unlikely event that some other SEG error messages appear, see the LOAD and SEG Reference Guide.

9. The QUIT command (abbreviated Q) saves the executable file, exits the SEG utility, and returns you to PRIMOS command level. Use QUIT if you do not want to execute the program immediately from inside the subprocessor. (Execution is discussed later in this chapter.)

Here is an example of a user's dialog with the SEG utility during a load procedure using the prefix method:

```
OK, SEG                    Enter the SEG utility.
[SEG rev 17.5]
# LO #TEST                 Enter LOAD and set up executable file.
$ LO B_TEST                Load main program's object file.
$ LO B_subprogram-name     Load separately compiled subprograms.
$ LI PASLIB                Load Pascal library.
$ LI library-name          Load other Prime libraries, as needed.
$ LI                       Load the standard system libraries.
LOAD COMPLETE              Loader indicates load is complete.
$ QUIT                     Save executable file and return to PRIMOS.
OK,
```

After the load is complete, you should have the following files in your directory:

- TEST (the source file)
- B_TEST (the object file)
- #TEST (the executable file)
- L_TEST (the source listing file, if you created one with the -LISTING option at compile time)

18.0 Table 3-1 provides a comparison and quick reference of suffix and prefix filenames.

Table 3-1
Suffix and Prefix Filename Conventions

File	Suffixed Name	Prefixed Name
Source file	filename.PASCAL	filename
Object file	filename.BIN	B_filename
Subprogram	subprogram-name.BIN	B_subprogram-name
Executable SEG file	filename.SEG	#filename
Source listing file	filename.LIST	L_filename

18.0

EXECUTING PROGRAMS

Once your program is loaded and you have returned to the PRIMOS command level, execute your program with the SEG command:

```
SEG filename
```

filename is the name of your program's executable file. If your program was loaded with the suffix method, and your executable file is called TEST.SEG, issue the command:

```
SEG TEST
```

If your program was loaded with the prefix method, and your executable file is called #TEST, issue the command:

```
SEG #TEST
```

SEG loads the executable file into segmented memory and begins execution of the program.

Executing from within the Subprocessor

A shortcut to saving and executing a loaded program is available. Immediately after receiving the LOAD COMPLETE message, enter the subprocessor's EXECUTE command (abbreviated EX). This command will then save the loaded program and start executing it. Upon completion of execution, control returns to PRIMOS command level. For example:

```
LOAD COMPLETE
$ EXECUTE
OK,
```

Compiling, Loading, and Executing with Command Files

You can save time in compiling, loading, and executing programs by creating a command file or CPL file that will automatically compile, load, and execute a program for you. For instructions on how to create command and CPL files, see the Prime User's Guide and the CPL User's Guide.

PART III

Prime Pascal Language Reference

4

Pascal Language Elements

This chapter, and all the other chapters in Part III, serve as a reference to the Pascal language -- standard Pascal as well as Prime extensions and restrictions. This language reference is not intended to be a Pascal tutorial. It does not teach you Pascal.

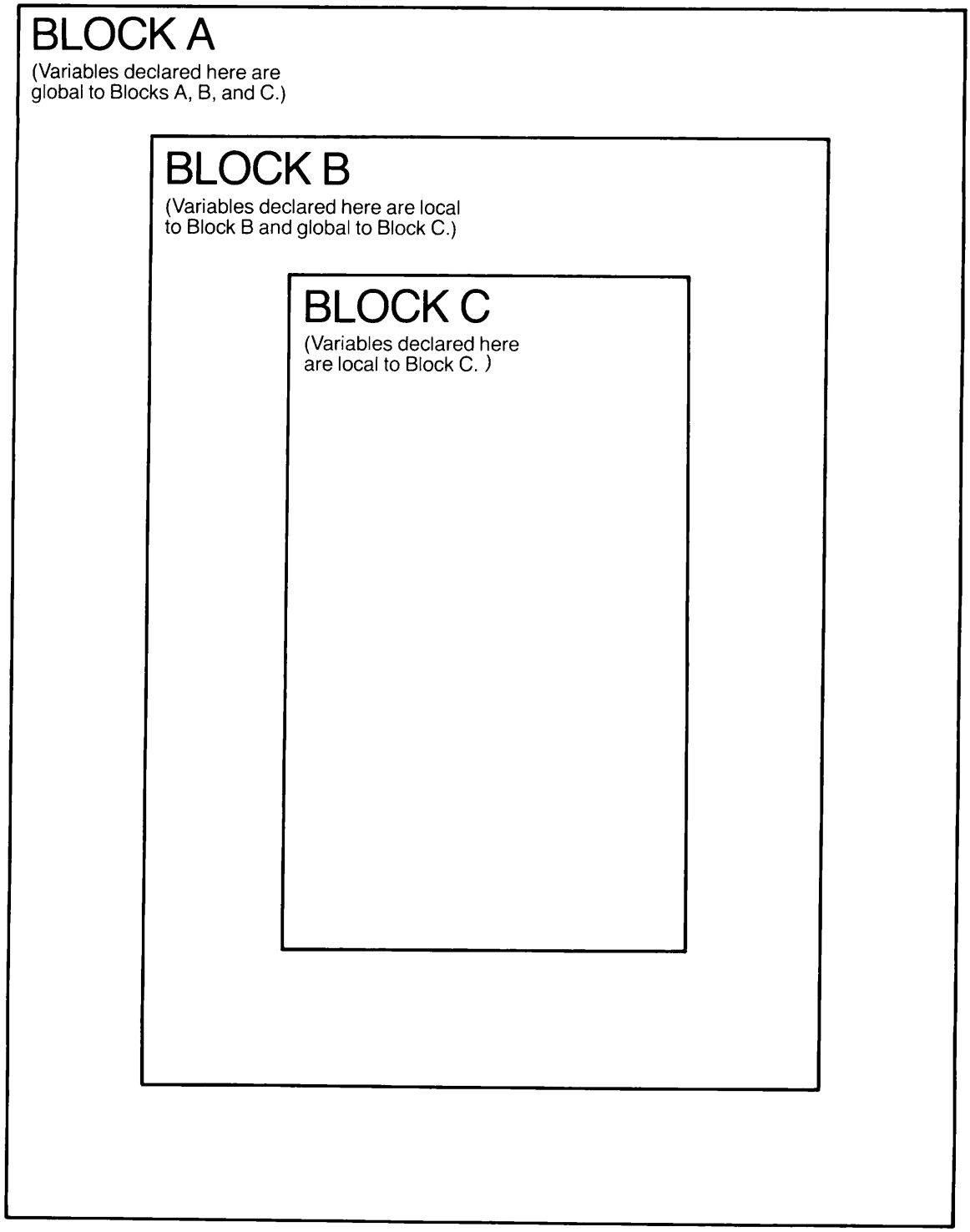
You are expected to be familiar with Pascal, but not necessarily with Prime computers. If you do not know Pascal, consult a commercially available text, such as the ones listed in Chapter 1.

Prime extensions and restrictions to standard Pascal are clearly identified throughout the book. For a summary of Prime extensions and restrictions to Pascal, along with references to where they are discussed in the book, see Appendix A.

DEFINITIONS

The terms defined below are used repeatedly throughout the book. Many other terms are defined in later chapters.

Term	Definition
Program	A main program consists of a heading and a block and ends with a period. (See Chapter 5.)
Program Unit	A program unit can be a main program, a procedure, or a function.
Subprogram	A subprogram is either a procedure or a function. It consists of a heading and a block and ends with a semicolon. (See Chapter 9.)
Heading	A heading gives a program unit a name and lists the program's parameters. (See Chapters 5 and 9.)
Object	An object is an identifier used in a program unit. (Identifiers are defined later in this chapter.)
Block	A block is the body of a program unit. It consists of a sequence of declarations (declaration part) describing data objects to be used in the program unit and a sequence of statements (executable part) describing actions to be performed on these objects. (See Chapters 5 and 9.) A program unit can have up to 64 levels of nesting of blocks within blocks. If block B is defined within block A, then B is called the inner block or inner level, and A is called the outer block or outer level. If block C is defined within block B, then B becomes an outer block to inner block C, but B is still an inner block to block A. The outermost block of a program is the program block itself. (See Figure 4-1.)
Global	The data objects, such as variables, declared in the outer block of a program unit are accessible at all inner levels of the program unit and are termed global. If block B is defined in block A and block C is defined in block B, then an object declared in A is said to be global to B and C, and an object declared in B is said to be global to C. (See Figure 4-1.) Objects declared at the program level (the outermost level) are global to all inner levels and can be referenced throughout the entire program.



Local and Global Variables in
Inner and Outer Blocks

Figure 4-1

Local	If an object is declared in a block, it is available or significant only within that block, and is said to be local to that block. However, if block B is within block A, then objects local to A are global to B and have significance in both A and B. (See Figure 4-1.)
Scope	The block in which an object is declared defines the scope of that object. In other words, the scope of an identifier or label is the block in which the declaration or definition of the identifier or label is valid.
Actual Parameter	An actual parameter is a variable or expression passed to a subprogram. Actual parameters appear in the parameter list of a procedure or function call (procedure statement or a function designator) within a block. (See Chapter 9.)
Formal Parameter	A formal parameter is a variable appearing in the parameter list of a subprogram heading. When the subprogram is invoked, the value of each actual parameter is passed to a corresponding formal parameter. A formal parameter can also be called "dummy" parameter or a "placeholder". (See Chapter 9.)

PASCAL CHARACTER SET

The Prime Pascal ANSI, ASCII 7-bit character set consists of:

- Twenty-six uppercase and 26 lowercase letters of the English alphabet (A to Z, a to z).
- Ten digits (0 - 9).
- Twenty-one punctuation symbols. These symbols are used by themselves and in certain combinations to represent operators and delimit textual elements as described in Table 4-1.

Appendix C lists the character set.

Table 4-1
Pascal Punctuation Symbols

Symbol	Description
+	Addition Identity Set union STRING concatenation (Prime extension)
-	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

| 19.2

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)

KEYWORDS

Keywords are special symbols with fixed meanings and purposes, which cannot be redefined. They can be used only as specified in the syntax for a Pascal program unit. Keywords may be written in lowercase letters, uppercase letters, or any combination of them. Lowercase letters will be interpreted the same as their uppercase counterparts. Table 4-2 lists all the available keywords.

Table 4-2
Pascal Keywords

AND	FUNCTION	PROCEDURE
ARRAY	GOTO	PROGRAM
BEGIN	IF	RECORD
CASE	IN	REPEAT
CONST	LABEL	SET
DIV	MOD	THEN
DO	NIL	TO
DOWNTO	NOT	TYPE
ELSE	OF	UNTIL
END	OR	VAR
FILE	OTHERWISE*	WHILE
FOR	PACKED	WITH
		%INCLUDE*

* Prime extension keyword

IDENTIFIERS

Identifiers are names used in Pascal source program units to denote programs, constants, types, variables, procedures, or functions. Identifiers may be written in either lowercase or uppercase letters or any combination of them. The compiler will convert all lowercase letters to their uppercase counterpart for the purpose of identifier recognition.

A Pascal identifier can be a user-defined identifier or a standard identifier.

User-defined Identifiers

User-defined identifiers are names supplied by the user. These names cannot be keywords.

A user-defined identifier must begin with a letter or a dollar sign, which may be followed by any combination of letters, digits, underscores, and dollar signs. It may contain up to 32 significant characters in its spelling. An identifier with more than 32 characters
19.1 | will result in a severity 1 error at compile time. This is a Prime restriction.

Standard Identifiers

Standard identifiers are names with predefined meanings and purposes, such as standard function names like SQR and ORD. If necessary, you may globally or locally redefine any standard identifier for another purpose. However, if an identifier is redefined, it cannot be used for its original purpose within the scope of that redefinition. For example, you may create a variable named ABS. Then, however, you would no longer be able to use the standard absolute value function ABS in the block containing the declaration of that variable. Table 4-3 lists all the available standard identifiers. Detailed descriptions are contained in appropriate chapters of this book.

NUMERIC CONSTANTS

Pascal has four forms of numeric constants — integer, longinteger, real, and longreal.

An integer or a longinteger is a whole number with an optional sign. It is either a constant of INTEGER or LONGINTEGER type respectively or a constant of a subrange of INTEGER or LONGINTEGER type respectively.

19.1 A real or a longreal number has a fractional part. It is a constant of REAL or LONGREAL type respectively.

The LONGINTEGER and LONGREAL data types are Prime extensions. LONGINTEGER allows you to use 32-bit whole numbers. LONGREAL allows you to use 64-bit real numbers. (See Chapter 6.) LONGINTEGERS have values in the range -2147483648..+2147483647.

Table 4-3
Standard Identifiers

Constants

FALSE	TRUE	MAXINT	

Types

INTEGER	LONGINTEGER*	REAL	LONGREAL*
BOOLEAN	CHAR	TEXT	STRING*

19.2

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

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)	1,234D+20 (No comma allowed)
1.234567 (longreal)	

CHARACTER-STRINGS

A character-string is a character or a sequence of characters enclosed by apostrophes. Character-strings consisting of a single character are CHAR type constants. Character-strings consisting of more than one character are either STRING type constants or ARRAY [1..n] OF CHAR type constants, where n is the number of characters in the string. The STRING type is a Prime extension. (See Chapter 6.) To include an apostrophe character in a string, double the apostrophe. Here are some character string examples: 19.2

```
''''    (single quote)

'A'

';'

'THIS IS A STRING'

'Pascal'

'Don''t give up the ship.'
```

DECLARATIONS AND STATEMENTS

Declarations describe data objects to be executed in a program unit. Statements perform explicit actions on the declared objects. Declarations must precede statements in the program text. (See Chapters 5 and 8 for detailed discussions.)

LINE FORMAT

The Pascal compiler ignores the formatting of source lines. A declaration or statement may start anywhere on a line. More than one declaration or statement may be written on a single line. However, a keyword, an identifier, or a number cannot be divided between lines.

This guide uses formatting for legibility in program examples.

COMMENTS, BLANKS, AND ENDS OF LINES

Comments, blanks (except in character-strings), and ends of lines are considered to be separators. Separators must not appear in identifiers, keywords, or numbers. At least one separator must be placed between identifiers, keywords, or numbers that are not separated by one or more of the punctuation symbols given in Table 4-1. One or more separators may occur anywhere in the program text except where it is not recommended throughout the book.

A comment has the form:

{sequence of characters}

in which the characters may be any character except the right brace } or the character sequences *) or */. In Pascal, comments may be placed anywhere blanks are allowed. Comments are inserted as notes that indicate the purpose of a program or a section of code. Also, comments are used to enable or disable compiler switches. (See Chapter 2.)

On many terminals, the brace characters are not available, so Prime Pascal also allows a comment to be delimited by the character pairs (* *) and /* */. Delimiters {}, (* *), and /* */ can be interchanged. Starting and ending comment delimiters need not have the same form. The delimiters /* */ are a Prime extension.

5

Pascal Program Structure

A standard Pascal program consists of a heading and a block and ends with a period. The block may contain up to six different kinds of declarations and a sequence of executable statements enclosed within the keywords BEGIN and END. Figure 5-1 illustrates the general structure of a program. The six different kinds of declarations, which are shown in Figure 5-1, are LABEL, CONSTANT, TYPE, VARIABLE, FUNCTION, and PROCEDURE.

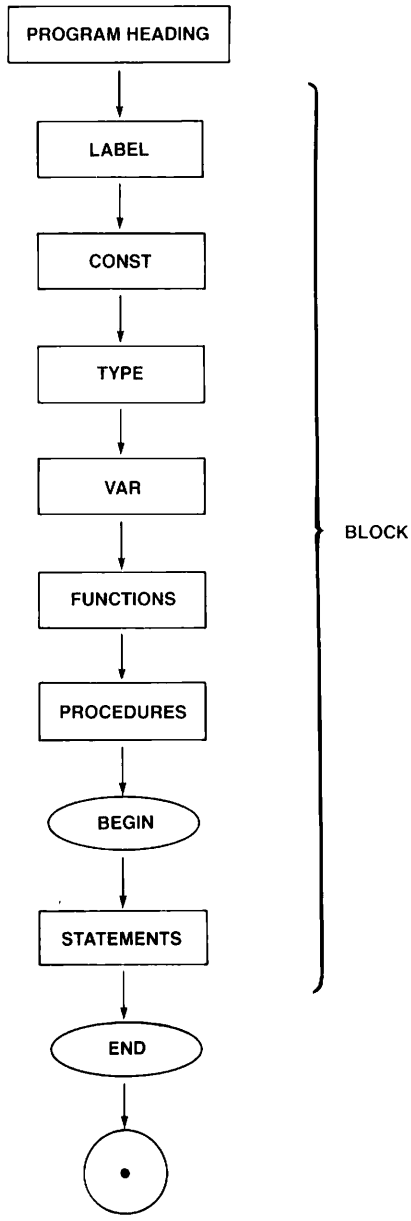
PROGRAM HEADING

The main difference between standard Pascal and Prime Pascal in program structure is the heading. In Prime Pascal, the program heading is optional. This is a Prime extension.

A program heading has the general form:

```
PROGRAM identifier [([file-identifier-list])];
```

The keyword PROGRAM must be the first word of a program heading. It is followed by an identifier, which is the name of the program, and an optional file-identifier-list, which is a list of files (separated by commas), used by the program. (Files are explained in Chapters 6 and 10.)



Program Diagram
Figure 5-1

Examples of program headings follow:

```
PROGRAM Sample;          {The file-identifier-list may be omitted.}

PROGRAM Y(OUTFILE);      {It is not necessary to list all the files}
                         {used by the program.}

PROGRAM X();

PROGRAM findroot(INPUT, OUTPUT);
```

Note

The program heading, if present, is only checked syntactically by the Prime Pascal compiler. The compiler does not check the existence of the files named in the file-identifier-list.

THE BLOCK

A block is divided into two parts — declaration and executable. The declaration part contains declarations that describe all data objects to be used in the program. The executable part, delimited by the keywords BEGIN and END, contains statements that specify the actions to be executed upon these declared objects. The general form of a block is:

```
 [LABEL declaration;]
 [CONST declaration;]
 [TYPE declaration;]
 [VAR declaration;]
 [FUNCTION declaration;]
 [PROCEDURE declaration;]
BEGIN
 [statement-1 [;statement-2]...]
END.
```

Note

The example above shows the standard Pascal order of declarations — LABEL, CONST, TYPE, VAR, FUNCTION, and PROCEDURE. You should use this standard order whenever possible. However, in Prime Pascal, the LABEL, CONST, TYPE, and VAR declarations can appear in any order. This is a Prime extension.

The following program contains LABEL, CONST, TYPE, VAR, and PROCEDURE definitions:

```
PROGRAM EX1;
LABEL
   1;
CONST
   ONE = 1;
TYPE
   SMALL = 1..3;
VAR
   TINY : SMALL;
PROCEDURE P(VAR X : SMALL);
   BEGIN {procedure P}
      X := 1
   END; {procedure P}
BEGIN {main program}
   P(TINY);
   IF (TINY <> ONE) THEN
      BEGIN
        WRITELN ('ERROR');
        GOTO 1
      END;
   IF (TINY = ONE) THEN
      WRITELN ('TINY = ', ONE)
1:
END.  {program EX1}
```

This is a null program:

```
PROGRAM Empty;
  BEGIN
  END.
```

DECLARATION PART

The declaration part's six optional subparts -- LABEL, CONSTANT, TYPE, VARIABLE, FUNCTION, and PROCEDURE parts -- must precede the executable part.

LABEL Declaration Part

The LABEL declaration part specifies all labels that mark statements in the corresponding executable part. The LABEL declaration part has the form:

```
LABEL label [,label]...;
```

The keyword LABEL heads this part.

Each declared label, which is an unsigned integer consisting of up to four digits, must be unique and mark only one statement in the executable part. However, if block B is nested in block A, a label declared in A is allowed to be redefined in B. Example:

```
PROGRAM Test(OUTPUT);
LABEL 6;
     .
     .
     .
PROCEDURE REDEFINE;
  LABEL 6;
     .
     .
     .
  BEGIN
    GOTO 6;
     .
     .
     .
  6: END; {of procedure REDEFINE}
BEGIN {main program}
     .
     .
     .
  REDEFINE;
  GOTO 6;
6: END. {of program TEST}
```

Here is an illegal use of LABEL:

```
PROGRAM Test;
LABEL 5;
BEGIN
   GOTO 5;
     .
     .
     .
   5: WRITELN ('HELLO');
   GOTO 5;
     .
     .
     .
 5:END. {illegal}
```

In the example above, the label 5 marks two statements. It can only mark one statement in a given block. This will generate a severity 3 error at compile time, and prevent successful compilation.

CONSTANT Declaration Part

All constants to be represented by names in a program must be declared in the CONSTANT declaration part. Numeric constants are discussed in Chapters 4 and 6. The form of this part is:

```
CONST     identifier-1 = constant-1;
         [identifier-2 = constant-2;]...
```

The keyword CONST heads this part.

Each identifier is a name that is associated with a specific constant. It will be used in place of the constant throughout the entire block containing the declaration unless the identifier is redefined.

A constant is a fixed value that may be an integer, longinteger, real, or longreal number with an optional sign, a character-string, or a constant-identifier (possibly signed). A constant-identifier is an identifier that has already been assigned a constant value.

Here are some examples of CONSTANT declarations:

```
CONST
   BLANK    = ' ';
   QUIT     = 'QUIT';
   TAX_RATE = 0.05;
   MAX      = 50;
   MIN      = -MAX;  {MAX is a constant identifier.}
```

Here is an example of how constants can be used:

```
CONST
   STOP = 'END OF OPERATIONS';
   MAXIMUM = 100;
VAR
   I : INTEGER;
   A : ARRAY[1..MAXIMUM] OF INTEGER;
BEGIN
   FOR I := 1 TO MAXIMUM DO
     BEGIN
         READ (A[I]);
         WRITELN(STOP)
     END;
END.
```

TYPE Declaration Part

All constants and variables in a program have types. The type of a constant is determined by the syntax of that constant. The type of a variable, on the other hand, must be explicitly specified in the VARIABLE declaration part (explained later in this chapter).

Prime Pascal provides seven standard (predefined) data types -- INTEGER, LONGINTEGER, REAL, LONGREAL, CHAR, BOOLEAN, and TEXT. In addition, Pascal permits users to define new data types in the TYPE declaration part of a program. (Data types are discussed in detail in Chapter 6.) 19.1

The TYPE declaration part, which always begins with the keyword TYPE, has the form:

```
TYPE    type-identifier-1 = data-type-1;
       [type-identifier-2 = data-type-2;]...
```

A type-identifier is the name of a specific data-type. It will be associated with one or more variables in the VARIABLE declaration part.

A data-type is either a new user-defined data type or a type-identifier that has already been associated with a new user-defined data type.

Here are some examples of TYPE declarations:

```
TYPE
  LETTERS    = 'A'..'Z';
  STRINGS    = ARRAY[1..50] OF CHAR;
  DAYSOFWORK = (MON,TUE,WED,THUR,FRI);
  STR        = FILE OF CHAR;
  CH         = LETTERS;     {CH and LETTERS denote the same type.}
```

VARIABLE Declaration Part

A variable is a named data object that can assume different values during the execution of a program. Variables to be used in the program must be declared in the VARIABLE declaration part. The form of this part is:

```
VAR    identifier-1 [, identifier-2]... : data-type-1;
      [identifier-3 [, identifier-4]... : data-type-2;]...
```

The keyword VAR heads this part.

Each identifier is the name of a variable contained in the program. The variable must be explicitly associated with a data-type which determines the range of values the variable can assume, the set of operations that can be performed on it, and the class of standard procedures and functions that can be used on it.

The data-type may either be one of the standard data types (INTEGER, LONGINTEGER, REAL, LONGREAL, CHAR, BOOLEAN, or TEXT) or a type-identifier as defined in the preceding TYPE declaration part. 19.1

For example, consider these TYPE declarations;

```
TYPE
  OPERATION_SIGNS = (PLUS, MINUS, TIMES);
  EXAM_SCORES     = 0..100;
  STRING15        = ARRAY [1..15] OF CHAR;
  DATE_RECORD     = RECORD
                      MONTH: 1..12;
                      YEAR:  INTEGER
                    END;
  LETTER_SETS     = SET OF 'A'..'Z';
  INTEGER_FILE    = FILE OF INTEGER;
```

Based on the TYPE declarations above, you can declare variables like this:

```
VAR
  OPERATORS : OPERATION_SIGNS;
  SCORES    : EXAM_SCORES;
  STRING1, STRING2 : STRING15;
  DATE      : DATE_RECORD;
  LETTERS   : LETTER_SETS;
  INTEGERS  : INTEGER_FILE;

  ROOT1, ROOT2      : REAL;
  COUNTER           : INTEGER;
  FLAG              : BOOLEAN;
  FILLER            : CHAR;
  TEXTIN, TEXTOUT   : TEXT;
```

The TYPE declaration and VARIABLE declaration may be combined. For example:

```
VAR
  OPERATORS : (PLUS, MINUS, TIMES);
  SCORES    : 0..100;
  STRING1, STRING2 : ARRAY [1..15] OF CHAR;
  DATE      : RECORD
                MONTH : 1..12;
                YEAR  : INTEGER
              END;
  LETTERS   : SET OF 'A'..'Z';
  INTEGERS  : FILE OF INTEGER;
```

However, it is necessary to keep the TYPE declaration and VAR declaration separate if the variables are to be used as actual parameters. (See Chapter 9.)

The association of an identifier and its data type is valid throughout the entire block containing the declaration unless the identifier is redefined. Suppose that block B is contained in block A. An identifier declared in A can be reassigned to a variable of any type local to B, and this redefined association is valid throughout the scope of B. Two examples follow.

Example 1:

```
PROGRAM  SAM1;
VAR
  V : INTEGER;
    .
    .
    .
PROCEDURE P1;
  VAR
    V : REAL;
     .
     .
     .
```

Example 2:

```
PROGRAM SAM2;
TYPE
  T = (TIME, TAPE, TIRE);
VAR
  V : T;
PROCEDURE P2;
  TYPE
    T = ARRAY[1..5] OF INTEGER;
  VAR
    V : T;
```

PROCEDURE and FUNCTION Declaration Parts

Procedures and functions are the two types of subprograms in Pascal. Every procedure or function must be declared in the PROCEDURE declaration part or the FUNCTION declaration part of its calling program respectively before it can be used. Procedures and functions are discussed in detail in Chapter 9.

EXECUTABLE PART

The executable part of a program, delimited by the keywords BEGIN and END, contains a sequence of statements that perform explicit actions on the data described in the declaration part of the block. All statements are discussed in detail in Chapter 8.

The following example shows the executable part of a program:

```
{Program Heading}
PROGRAM CONVERSION;

{Declaration Part}
VAR
  CHARACTER : CHAR;
  NUMBER    : INTEGER;

{Executable Part}
BEGIN
  READ(CHARACTER);
  NUMBER := ORD(CHARACTER);
  WRITELN(' CHARACTER ', '''', CHARACTER, '''',
    ' IS EQUAL TO NUMBER ', NUMBER)
END.
```

The %INCLUDE Directive

The %INCLUDE compiler directive is a Prime extension to standard Pascal. It is a Prime Pascal keyword.

%INCLUDE provides a means of directing the compiler to include the contents of a file in the program unit at compile time. %INCLUDE files can hold any legal Prime Pascal code — declarations as well as executable statements. The files could contain very long lists of variable declarations, for example.

The general form of %INCLUDE is:

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

where filename is the name of the file to be incorporated into the program unit at the position of %INCLUDE. The filename can be a pathname if the included file does not reside in the current directory.

A %INCLUDE directive can appear anywhere that a declaration or definition of the declaration part or a statement of the executable part can appear. An included file may contain additional %INCLUDEs. A %INCLUDE file commonly contains:

- Declarations that are common to more than one program unit
- Numeric key definitions, especially for the file management system and application library

%INCLUDE directives can be nested up to seven levels.

The following is an example of the %INCLUDE directive:

```
PROGRAM  SAMPLE;
  %INCLUDE 'VAR_FILE'; {Suppose that VAR_FILE contains
          .                    {a set of commonly used}
          .                    {variable declarations.}
          .
PROCEDURE  P1;
  %INCLUDE 'VAR_FILE';
          .
          .
          .
```

A PROGRAM EXAMPLE

The following is a somewhat more complex program example, which contains LABEL, CONST, TYPE, VAR, and PROCEDURE declarations:

```
PROGRAM Bowling (INPUT, OUTPUT);

{This program computes the scores of four bowlers, the number of
spares and strikes -- and the frames in which the marks were
scored -- and it calculates the winner and the winning score.}

LABEL 1;

CONST
  TOPFRAME = 10;

TYPE
  HUMAN_TYPE = ARRAY[1..20] OF CHAR;
  PIN_TYPE   = ARRAY[1..22] OF INTEGER;

VAR
  NAME, BALL, FRAME, N, TOTAL, BESTSCORE : INTEGER;
  PLAYER, WINNER : HUMAN_TYPE;
  PINS : PIN_TYPE;
  INFILE : TEXT;  {the input data file}

{The WINNING_PLAYER procedure determines the winning player.}

PROCEDURE WINNING_PLAYER;
  BEGIN {procedure winning_player}
    IF TOTAL > BESTSCORE THEN
      BEGIN
        BESTSCORE := TOTAL;
        WINNER := PLAYER
      END;
    READLN(INFILE);
    BALL := 1;
    TOTAL := 0
  END; {procedure winning_player}
```

```
{Open the input file, initialize integer counters to 0.  Read the
player's name and write the player's name.}

BEGIN  {main program}
  RESET (INFILE, 'BOWLINPUT');  {open the input data file}
  BALL := 1; FRAME := 1; TOTAL := 0; BESTSCORE := 0;
  WHILE NOT EOF(INFILE) DO
   BEGIN
    FOR NAME := 1 TO 20 DO
      BEGIN
        READ (INFILE, PLAYER[NAME]);
        WRITE (PLAYER[NAME])
      END;
    WRITELN;

{Read the total number of balls bowled. Using this number, read
the integer array of all the pin scores for that player,
reinitialize ball to 1 again after the read, and for each frame
calculate whether the scores are strikes, spares, or non-marks.
Write out the frame number, whether that frame was a strike, spare,
or nonmark, and write out the pin scores.}

READ (INFILE, N);
FOR BALL := 1 TO N DO
  READ (INFILE, PINS[BALL]);
BALL := 1;
FOR FRAME := 1 TO TOPFRAME DO
  BEGIN
    IF PINS[BALL] = 10 THEN
      BEGIN
        TOTAL := TOTAL+PINS[BALL]+PINS[BALL + 1] + PINS[BALL + 2];
        WRITELN ('FRAME ',FRAME:2,' is a strike ',
                 'and PIN SCORE is ',PINS[BALL]:5);
        IF FRAME = TOPFRAME THEN
          WRITELN ('Extra pins on strike are:',PINS[BALL+1]:3,
                   PINS[BALL+2]:3);
        BALL := SUCC(BALL)
      END
```

```
    ELSE
    IF PINS[BALL] + PINS[BALL + 1] = 10 THEN
      BEGIN
        TOTAL := TOTAL + PINS[BALL] + PINS[BALL+1] + PINS[BALL+2];
        WRITELN ('FRAME ',FRAME:2,' is a spare ',
                 'and PIN SCORES are ',PINS[BALL]:4,PINS[BALL+1]:3);
        IF FRAME = TOPFRAME THEN
          WRITELN ('Extra pin on spare is:', PINS[BALL+2]:3);
        BALL := SUCC(SUCC(BALL))
      END
    ELSE
    IF PINS[BALL] + PINS[BALL + 1] < 10 THEN
      BEGIN
        TOTAL := TOTAL + PINS[BALL] + PINS[BALL + 1];
        WRITELN ('FRAME ',FRAME:2,' is not a mark.  ',
                  'PIN SCORES are ',PINS[BALL]:3,PINS[BALL + 1]:3);
        BALL := SUCC(SUCC(BALL))
      END
  END;  {FOR loop}

{When all the pin scores for all frames have been calculated, write
out the final score for each player.}

    WRITELN ('FINAL SCORE  is ',TOTAL:4);
    WRITELN;
    WRITELN;

{The procedure  WINNING_PLAYER  is  called.  It will keep track of
highest score and keep track of the name of  the  player  with  the
highest score,  assigning  the  score  and  the  name  to TOTAL and
WINNER}

    WINNING_PLAYER  {the procedure is called here}
  END;  {WHILE loop}

{Write out the winner's name and the winning score, and  close  the
input ending the program.}

  WRITE ('The winner is ', WINNER);
  WRITELN;
  WRITE ('with a score of',BESTSCORE:4);
  CLOSE (INFILE);  {close the input data file}
  GOTO 1;  {example of GOTO statement}
  1:  {example of LABEL definition}
END.
```

The input data file looks like this:

```
peter larrington    18   9 1 0 10 10 10 6 2 7 3 8 2 10 9 0 9 1 7
lisa rubin          21   0 4 3 2 1 4 6 1 4 3 0 0 1 2 3 3 4 0 7 3 2
anne ladd           18   10 8 2 10 5 5 4 5 10 8 1 9 0 9 1 10 6 3
paul cioto          14   10 10 10 10 9 1 10 10 10 8 2 10 10 9
```

The output of this program will look like this:

```
peter larrington
FRAME  1 is a spare and PIN SCORES are    9  1
FRAME  2 is a spare and PIN SCORES are    0 10
FRAME  3 is a strike and PIN SCORE is    10
FRAME  4 is a strike and PIN SCORE is    10
FRAME  5 is not a mark.  PIN SCORES are   6  2
FRAME  6 is a spare and PIN SCORES are    7  3
FRAME  7 is a spare and PIN SCORES are    8  2
FRAME  8 is a strike and PIN SCORE is    10
FRAME  9 is not a mark.  PIN SCORES are   9  0
FRAME 10 is a spare and PIN SCORES are    9  1
Extra pin on spare is:  7
FINAL SCORE  is  165


lisa rubin
FRAME  1 is not a mark.  PIN SCORES are   0  4
FRAME  2 is not a mark.  PIN SCORES are   3  2
FRAME  3 is not a mark.  PIN SCORES are   1  4
FRAME  4 is not a mark.  PIN SCORES are   6  1
FRAME  5 is not a mark.  PIN SCORES are   4  3
FRAME  6 is not a mark.  PIN SCORES are   0  0
FRAME  7 is not a mark.  PIN SCORES are   1  2
FRAME  8 is not a mark.  PIN SCORES are   3  3
FRAME  9 is not a mark.  PIN SCORES are   4  0
FRAME 10 is a spare and PIN SCORES are    7  3
Extra pin on spare is:  2
FINAL SCORE  is   53
```

```
anne ladd
FRAME  1 is a strike and PIN SCORE is    10
FRAME  2 is a spare and PIN SCORES are    8  2
FRAME  3 is a strike and PIN SCORE is    10
FRAME  4 is a spare and PIN SCORES are    5  5
FRAME  5 is not a mark.  PIN SCORES are   4  5
FRAME  6 is a strike and PIN SCORE is    10
FRAME  7 is not a mark.  PIN SCORES are   8  1
FRAME  8 is not a mark.  PIN SCORES are   9  0
FRAME  9 is a spare and PIN SCORES are    9  1
FRAME 10 is a strike and PIN SCORE is    10
Extra pins on strike are:  6  3
FINAL SCORE  is  159


paul cioto
FRAME  1 is a strike and PIN SCORE is    10
FRAME  2 is a strike and PIN SCORE is    10
FRAME  3 is a strike and PIN SCORE is    10
FRAME  4 is a strike and PIN SCORE is    10
FRAME  5 is a spare and PIN SCORES are    9  1
FRAME  6 is a strike and PIN SCORE is    10
FRAME  7 is a strike and PIN SCORE is    10
FRAME  8 is a strike and PIN SCORE is    10
FRAME  9 is a spare and PIN SCORES are    8  2
FRAME 10 is a strike and PIN SCORE is    10
Extra pins on strike are: 10  9
FINAL SCORE  is  256


The winner is paul cioto
with a score of 256
```

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.

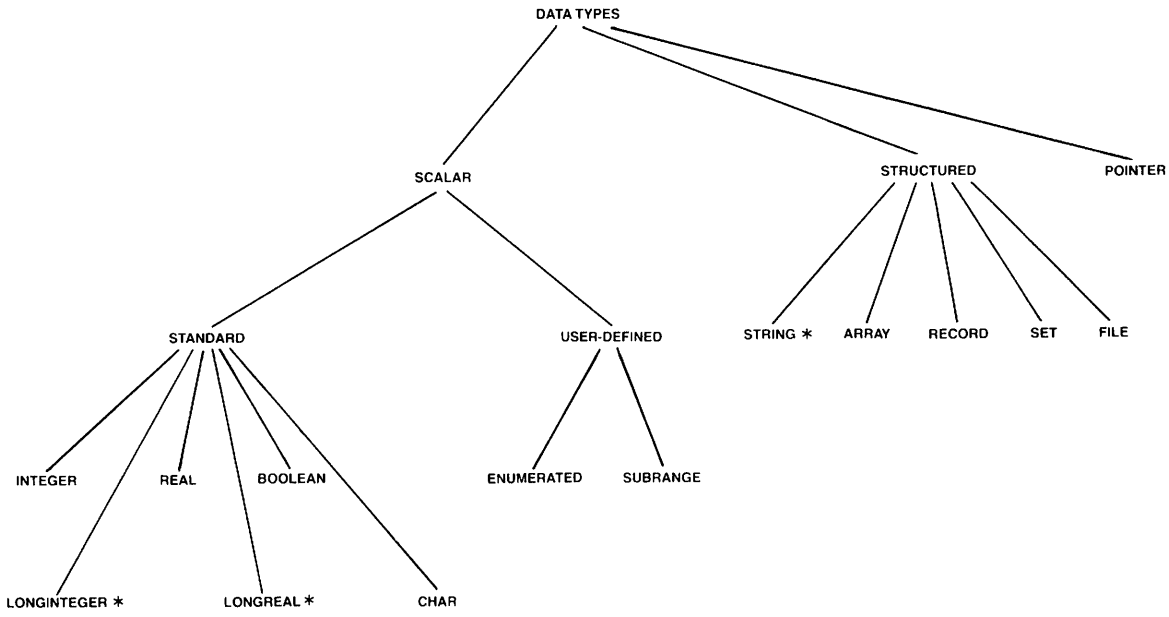
This chapter summarizes the data types available in Prime Pascal — standard Pascal data types as well as Prime extensions. There are three Prime extension data types: LONGINTEGER, LONGREAL, and STRING. | 19.2
Each of these data types is described later in this chapter.

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.



19.21

The Hierarchy of Data Types in Prime Pascal
*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

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

The INTEGER Type: The INTEGER type comprises a subset of whole numbers (integers), which are 16-bit, twos-complement, fixed-point binary numbers. The values of the INTEGER type are in the range of -32768 to +32767 or -2^{15} to $+2^{15}-1$. An integer variable is simply declared:

```
VAR
  I : INTEGER;
```

You can use 32-bit whole numbers by simply declaring variables as LONGINTEGER, which is a Prime extension. (See the discussion on LONGINTEGER, which follows.) However, if you do not have a Rev. 19.1 (or higher) compiler, and want to use 32-bit whole numbers, you must declare these numbers as the constants of a subrange of the INTEGER type itself in a TYPE or VAR declaration: 19.1

```
TYPE
  I = -45000..+70000;
VAR
  X : I;
```

or

```
VAR
  X : -155000..+999000;
```

A 32-bit whole number can be declared within the range:

```
-2147483648..+2147483647
```

In this subrange declaration, either the lower-bound or upper-bound should be outside the range of INTEGER. For example:

```
-500..+57000
```

Note

Comparison of unsigned integers is not supported.

There is a predefined Pascal constant called MAXINT, whose value is the largest available integer constant of the INTEGER type. MAXINT is 32767.

Some examples of valid and invalid INTEGER type constants are:

Valid

```
 32767
  +200
     0
MAXINT
   -11
```

Invalid

```
 32,767     {No comma allowed}
 40000      {This number is a longinteger.}
-32769      {This number is a longinteger.}
 32.00      {A valid real number but not an integer}
```

There are five arithmetic operators: +, -, *, DIV (divide), and MOD (modulus or remainder), and six relational operators: =, <>, <, >, <=, and >= available for the INTEGER type. Table 4-1 in Chapter 4 gives a brief description of each operator. Chapter 7 gives a detailed discussion of all Prime Pascal operators.

19.1 There are four standard functions used frequently to produce INTEGER (or LONGINTEGER) results. In the following examples, I is any integer or longinteger and R is any real or longreal number:

```
ABS(I)      {Absolute value of I}

SQR(I)      {Square of I}

TRUNC(R)    {R truncated to an integer or longinteger}

ROUND(R)    {R rounded to an integer or longinteger}
```

See Chapter 11 for more information on standard functions.

The LONGINTEGER Type: The LONGINTEGER type is a Prime extension. LONGINTEGER allows you to use 32-bit whole numbers without declaring a subrange. For example:

```
VAR
  I : LONGINTEGER;
```

19.1 This declaration means that the variable I can have a value anywhere within the subrange -2147483648..+2147483647.

On Prime machines an integer is a 16-bit number, within the subrange -32768..+32767. (See the previous discussion on the INTEGER type.)

Note

The LONGINTEGER type is available on Rev. 19.1 (or higher) compilers. If you do not have a Rev. 19.1 compiler and want to use 32-bit whole numbers, you must declare these numbers as the constants of a subrange of the INTEGER type itself. For example:

```
TYPE
  I = -87000..+55000;
VAR
  X : I;
```

or

```
VAR
  X : -4856000..+9990000;
```

The arithmetic and relational operators and standard Pascal functions can be used with LONGINTEGER as well as with INTEGER. LONGINTEGER values can also be passed as parameters to procedures and functions.

It is recommended that you do not mix INTEGER types with LONGINTEGER types. You can assign an integer to a longinteger, but when you try to assign a longinteger to an integer, a severity 1 error message will be given at compile time. 19.1

LONGINTEGER constants are allowed. The compiler decides whether the constant is an integer or a longinteger.

Consider the following program example:

```
PROGRAM Longint;
CONST
  X = 55000;
VAR
  A : INTEGER;
  B : LONGINTEGER;
  C : INTEGER;
BEGIN
  B := 40000;
  A := B;
  C := X;
  C := A + B
END.
```

In the example above, LONGINTEGER values are assigned to INTEGER variables A and C. Each of those statements, therefore, would receive the following severity 1 error message:

```
ERROR 121 SEVERITY 1 BEGINNING ON LINE zzz
A type conversion must be made in this statement and may cause the
program to fail if the conversion is not possible.
```

The REAL Type: The REAL type is a subset of real numbers (decimal values). The approximate range of real numbers is $-1*10^{38}$ to $+1*10^{38}$. Real numbers are declared:

```
VAR
  R : REAL;
```

19.1 Prime's real numbers are 32-bit numbers. As of Rev. 19.1, you can use 64-bit numbers by simply declaring them LONGREAL, which is a Prime extension. (See the following discussion on LONGREAL.)

There are two methods of representing real constants -- the decimal notation and the scientific notation. (See also Chapter 4.) The 19.1 letter E in scientific notation is the exponent symbol for real numbers. (The letter D is used for longreals.) The following are examples of valid and invalid REAL type constants:

Valid

```
    +12.0 }
   3.14159 }   decimal notation
 -0.123456 }
      23E3 }
    -7.0E-5 }   scientific (floating-point) notation
  +2.01E+20 }
```

Invalid

```
     2.      {No digit to the right of the decimal point}
  .10101     {No digit to the left of the decimal point}
   3E8.5     {Only whole number exponent permitted}
```

There are four arithmetic operators: +, -, *, and /, and six relational operators: =, <>, <, >, <=, and >= applicable to the REAL type. For more information on these operators, see Chapter 7.

19.1 The following standard functions produce REAL or LONGREAL type results. X is either an integer, longinteger, real, or longreal number.

```
SIN(X)        {Sine of X}

COS(X)        {Cosine of X}

LN(X)         {Natural logarithm of X}

EXP(X)        {Exponential to the X power}

SQRT(X)       {Square root of X}

ARCTAN(X)     {Inverse tangent of X}
```

See Chapter 11 for more information on standard functions, including ABS and SQR.

The LONGREAL Type: The LONGREAL type is a Prime extension. Longreals are 64-bit numbers, as opposed to reals, which are 32-bit numbers. Variables are simply declared:

```
VAR
  X : LONGREAL;
```

The arithmetic and relational operators and standard Pascal functions can be used with LONGREAL as well as REAL. LONGREAL values can be passed as parameters to procedures and functions.

It is recommended that you do not mix REAL and LONGREAL types for the same reasons given in the LONGINTEGER discussion earlier in this chapter.

Longreal numbers can also be represented in decimal notation or scientific notation. The letter D signifies the exponent in scientific notation for longreals. (The letter E is used for reals.) For example:

```
12.3456789D-01    {scientific notation}

1.234567          {decimal notation}
```

19.1

Constants of the LONGREAL type are allowed. The compiler decides whether the declared constant is a real or a longreal. If the constant has more than six digits, the compiler assumes it to be LONGREAL. If the constant has six or fewer digits, the compiler assumes it to be REAL. For example:

```
CONST
  A = 5.47;          {stored as REAL}
  B = 35.123456;     {stored as LONGREAL}
```

When an exponent is present in a CONST declaration, the compiler assumes the number to be REAL if the exponent symbol is the letter E, or LONGREAL if the exponent symbol is D. For example:

```
CONST
  X = 2.1E01;    {stored as REAL}
  Y = 2.1D01;    {stored as LONGREAL}
```

If the constant given is too large to fit into a real number but has an E exponent, an error will be generated. For example:

```
CONST
  Z = 1.2E45;
```

The BOOLEAN Type: The BOOLEAN type has two standard constant values: TRUE and FALSE. When these values are compared, TRUE > FALSE.

The six relational operators =, <>, <, >, <=, and >= operate on any standard scalar types, user-defined scalar types, or the ARRAY OF CHAR string type (discussed later in this chapter) to produce a BOOLEAN result.

In addition, three BOOLEAN operators (OR, AND, and NOT) can be applied only to BOOLEAN values to produce BOOLEAN results. All operators are described in Chapter 7.

Three BOOLEAN functions (ODD, EOF, and EOLN) return BOOLEAN values TRUE or FALSE. See Chapters 10 and 11 for more information on these functions.

The CHAR Type: The CHAR type is a group of characters, or a "character set", that includes both printable (graphic) and nonprintable (control) characters. The standard character set used by Prime is the ANSI, ASCII 7-bit character set.

Internally, each character in Prime's character set has a numeric equivalent, which establishes a chronological order of characters or "collating sequence" for the character set. These values range from octal 200 to octal 377 (decimal 128 to 255). The nonprintable (control) characters are numbered 200 to 237 (octal) or 128 to 159 (decimal). The printable (graphic) characters are numbered 240 to 377 (octal) or 160 to 255 (decimal). Appendix C lists values in the character set.

Pascal's standard CHR function can convert a decimal number to its corresponding character. For example:

```
PROGRAM Kar;
VAR
  A, B : CHAR;
BEGIN
  A := CHR(198);
  B := CHR(199);
  WRITELN(A);
  WRITELN(B)
END.
```

The decimal numbers 198 and 199 stand for the characters F and G respectively. The letters F and G, therefore, would be printed at your terminal.

You can compare character values. Since 198 is less than 199:

```
'F' is less than 'G'
```

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 two more complex types called ARRAY OF CHAR and STRING, which are described later in this chapter. STRING is a Prime extension. 19.2

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]...);
```

The identifiers contained in parentheses are the constants of the new enumerated type, and type-identifier is the name of the new type. For example:

```
TYPE
  COLOR = (RED, YELLOW, GREEN, BLUE, PINK);
  SEX   = (MALE, FEMALE);
  FLAG  = (TRUE, FALSE);
  DAYS  = (MON, TUE, WED, THUR, FRI, SAT, SUN);
  MONTH = (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV,
           DEC);
```

The ordinal number is 0 for the first (leftmost) constant and is incremented by 1 for each successive constant. The largest allowable ordinal number of an enumerated type is 32767 on Prime computers. The ordering relationship between any two constants is the same as between their ordinal numbers. Therefore:

```
YELLOW is greater than RED
```

Variables are declared to be of these newly created data types by the variable declarations:

```
VAR
  COLOURS: COLOR;
  S: SEX;
  F: FLAG;
  WEEKDAY:  DAYS;
  MONTHS: MONTH;
```

The type definition and variable declaration may be combined. Example:

```
VAR
  S : (MALE, FEMALE);
```

Hypothetically, if s1, s2, and s3 are valid statements, then the following examples, based on the above declarations, are valid statements:

```
WHILE WEEKDAY <= FRI DO s1;

FOR MONTHS := MAR TO SEP DO s2;

WEEKDAY := SUCC(WED);

F := TRUE;

IF COLOURS <> GREEN THEN s3;
```

The constants of one enumerated type may not appear in any other enumerated type. The following example is illegal:

```
TYPE
  FAMILY  = (MOTHER, FATHER, SISTER, BROTHER);
  PARENTS = (FATHER, MOTHER); {this is illegal}
```

However, a type declared as a subrange of an enumerated type is legal:

```
TYPE
  FAMILY  = (MOTHER, FATHER, SISTER, BROTHER);
  PARENTS = MOTHER..FATHER;  {this is legal}
```

The relational operators =, <>, <, >, <=, and >= are applicable on all enumerated types provided both operands are of the same enumerated type.

Three standard functions (SUCC, PRED, and ORD) apply to enumerated types. These functions also apply to INTEGER, LONGINTEGER, BOOLEAN, CHAR, and subrange types. For example, given the following type definition:

```
TYPE
  SHAPE = (SQUARE, CIRCLE, RECTANGLE, TRIANGLE);
```

then

```
SUCC(CIRCLE) yields RECTANGLE      {Successor of CIRCLE}

PRED(CIRCLE) yields SQUARE         {Predecessor of CIRCLE}

ORD(CIRCLE) yields 1               {Ordinal number of CIRCLE}
```

> **Caution**
>
> When the PRED value of the leftmost enumerated type element or the SUCC value of the rightmost enumerated type element — an out-of-bounds value — is assigned, no compile-time or runtime errors are generated.

The Subrange Types: A subrange type is a data type that comprises a specified range of any other already defined scalar data type, except 19.1 types REAL and LONGREAL.

To define a subrange type, use the following type definition.

```
TYPE type-identifier = lower-bound..upper-bound;
```

Both lower-bound and upper-bound are constants of the same standard scalar type (except REAL and LONGREAL) or previously defined enumerated type, termed the base type, and the lower-bound value must not be greater than the upper-bound value. The type-identifier is the name of the new data type that comprises only those base type constants between the lower-bound and upper-bound. The following are examples of subrange types:

```
TYPE

 EXAMSCORE = 0..100;        {Subrange of INTEGER}
 DIGITS = '0'..'9';         {Subrange of CHAR}
 LETTERS = 'A'..'Z';        {Subrange of CHAR}

 DAYS = (MON, TUE, WED, THUR, FRI, SAT, SUN);  {Enumerated type}
 WEEKDAYS = MON..FRI;       {Subrange of DAYS}

 MONTHS = (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC);
 VACATION = JUN..SEP;       {Subrange of MONTHS}
 FIRST_TERM = JAN..MAY;     {Subrange of MONTHS}
```

Once the new data types are defined, they will be associated with appropriate variables by variable declarations:

```
VAR
  SCORES : EXAMSCORE;    {EXAMSCORE is a subrange of INTEGER
                          defined in the example above.}
```

The type definition and variable declaration may be combined:

```
VAR
  SCORES : 0..100;
```

According to standard Pascal, you cannot assign an element that is outside the subrange. Based on the above example, the assignment SCORES := 95 is permissible, but SCORES := 101 is not.

> Caution
>
> Prime's compiler will not give you an error message when you mistakenly assign an element that is outside the subrange.

Any operations that are normally performed on scalar types can be applied to subrange types. For example, if you have a subrange of integers, you can use any operation that you would normally use on integers.

Constants of subranges of types INTEGER and LONGINTEGER can either be 16-bit or 32-bit twos-complement, fixed-point binary numbers respectively. Examples:

```
CONST
  NUMBERS = -33000..+33000;      {32-bit binary numbers}
  LIMITS = 40000..60000;         {32-bit binary numbers}
  YEARS = 1700..1900;            {16-bit binary numbers}
```

STRUCTURED DATA TYPES

A structured data type is characterized by the type of its components and by its structuring method. A component may have a scalar or structured data type. Although a structured data type can be quite sophisticated, it is ultimately built up from scalar data types.

19.2 There are five basic structured data types -- STRING, ARRAY, RECORD, SET, and FILE. The STRING type is a Prime extension. Each of these data types can be declared in a TYPE or VAR declaration.

> **Caution**
>
> The keyword PACKED is not supported on Prime Pascal. This is a Prime restriction. Use of PACKED will generate a severity 1 error at compile time.

THE STRING TYPE

The STRING data type is a Prime extension. Similar to the PL/I-G CHARACTER VARYING type, the STRING type makes it easy to manipulate character strings in Prime Pascal. Unlike an array of characters, 19.2 which must contain a precise number of character elements, STRING allows you to assign, compare, concatenate, read, write, and pass character strings that have a varying number of elements.

For information on passing Pascal strings to PL/I-G CHARACTER VARYING strings and vice versa, see Appendix D, INTERFACING PASCAL TO OTHER LANGUAGES.

Declaring Strings

A variable of type STRING is declared in this form:

```
VAR
  string-identifier : STRING[n];
```

The string-identifier is the variable of STRING type, and n is the maximum number of character elements allowed in the string. This number is called the maximum length of the string. If n is not given in a STRING declaration, the maximum length is 80 by default.

Consider the following example:

```
VAR
  A : STRING;  {80 characters}
  B : STRING[5];  {5 characters}
  C : STRING[10];  {10 characters}
BEGIN
  B := 'HI';
  C := 'HELLO';
  WRITELN(C)
END.
```

The maximum length of string A is 80 characters. Strings B and C have maximum lengths of 5 and 10 respectively. During execution, at the WRITELN statement, B contains two characters and C contains five characters. Therefore, variables declared as type STRING can hold character-string values of any length less than or equal to the maximum length of the string. The length of a character string assigned to a STRING variable is called the operational length of the string. Thus, in the example above at the WRITELN statement, string B has a maximum length of 5 and an operational length of 2. String C has a maximum length of 10 and an operational length of 5. The operational lengths may change when new values are assigned to the character strings. 19.2

You can use CONST and TYPE declarations with STRING. For example:

```
CONST
  STRING_LENGTH = 20;
TYPE
  STRING_2 = STRING[2];
  STRING_5 = STRING[5];
  STRING_20 = STRING[STRING_LENGTH];
VAR
  ST2 : STRING_2;
  ST5 : STRING_5;
  ST20 : STRING_20;
```

Note

A string can be declared to have a maximum length of 32767 characters and a minimum length of 1 character.

The Null String

A null string, which is specified by '', is allowed. Null strings can be used to initialize strings. You may assign a null string, but an attempt to write a null string will generate a runtime error. The null string is also a Prime extension. Here is an example of a null string assignment:

```
VAR
  S : STRING[10];
BEGIN
  S := '';
```

Assigning Strings

Strings can be assigned to one another. When the value of one string is assigned to another string, the operational length is also assigned.

Note

A character literal string consists of one or more characters enclosed in single quotes. It should not be confused with a string, which is a variable that represents a STRING type value. Character literal strings, such as 'HELLO' or 19.2 'greetings', may be assigned to strings.

Here is an example that assigns character literals to strings and assigns one string to another string:

```
VAR
  ST2 : STRING[2];
  ST5 : STRING[5];
BEGIN
  ST2 := 'HI';
  ST5 := 'HELLO';
  ST5 := ST2      {operational length of ST5 is 2}
                  {and its value is 'HI'}
END.
```

If the operational length being assigned is larger than the maximum length of the string receiving the assignment, the excess characters are truncated. For example:

```
VAR
  ST2 : STRING[2];
  ST5 : STRING[5];
BEGIN
  ST5 := 'HELLO';
  ST2 := ST5      {value of ST2 is now 'HE'}
END.              {and its operational length is 2}
```

Here is another example of string assignments:

```
CONST
  STR_LENGTH = 10;
VAR
  A : STRING;
  B : STRING[4];
  C : STRING[8];
  D : STRING[STR_LENGTH];
BEGIN
  B := 'four';       {operational length is 4}
  B := 'fo';         {operational length is 2}
  D := '1234567890';      {operational length is 10}
  D := '12345';           {operational length is 5}
  D := B; {value of D is 'fo'}
  D := '123456';
  B := D; {value of B is '1234'}
  A := ''; {this is a legal assignment}
  WRITELN(A) {but this will cause a runtime error}
END.
```

Here are two rules governing string assignments:

- If the operational length of the string being assigned (the sending string) is less than or equal to the maximum length of the receiving string, then the entire string value is assigned, and the receiving string assumes the operational length of the sending string. 19.2

- If the operational length of the sending string is greater than the maximum length of the receiving string, then only the number of characters in the sending string equal to the maximum length of the receiving string are assigned. The remaining characters are not assigned.

Assigning Arrays and Strings to Each Other

Strings and arrays of characters can be assigned to one another through the use of two functions, STR and UNSTR. The STR function converts an array of characters or a single character to a string, and the UNSTR function converts a string to an array of characters or to a single character. The STR and UNSTR functions are Prime extensions.

The result of the STR function is a string with a length of the same number of characters as the array of characters argument. The result of an STR function may be used anywhere a string may be used.

The result of the UNSTR function is an array of characters or a single character. The number of characters in the newly formed array is determined by context. That is, the context of whatever array length is expected determines the length. The result of an UNSTR function may be used anywhere an array of characters is expected. Here are some specific rules governing the use of the UNSTR function:

- If the result of the UNSTR function is being assigned to an array of characters, then that result will have the same number of characters as the receiving array of characters.
- If the result of the UNSTR function is being passed to a procedure or function, then that result will have the same number of characters as the formal parameter.
- If the result of the UNSTR function is being compared to an array of characters, then that result will have the same number of characters as the array of characters to which it is being compared.
- If the UNSTR function is used in any other context, the length of the resultant array will be the same as the operational length of the string argument.

19.2

Here is an example that converts strings and arrays of characters to one another using STR and UNSTR:

```
VAR
  ST4 : STRING[4];
  ST8 : STRING[8];
  AR4 : ARRAY[1..4] OF CHAR;
  AR8 : ARRAY[1..8] OF CHAR;
BEGIN
  AR4 := 'JUNK';
  ST4 := STR(AR4);  {value of ST4 is 'JUNK'}
  AR4 := 'BLUE';
  ST8 := STR(AR4);  {value of ST8 is 'BLUE'}
  AR8 := 'LAVENDER';
  ST4 := STR(AR8);  {value of ST4 is 'LAVE'}
  ST4 := 'JUNK';
  AR4 := UNSTR(ST4);  {value of AR4 is 'JUNK'}
  AR8 := UNSTR(ST4);  {value of AR8 is 'JUNK    '}
  ST8 := 'LAVENDER';
  AR4 := UNSTR(ST8)  {value of AR4 is 'LAVE'}
END.
```

Arrays of characters are discussed later in this chapter.

Comparing Strings

String comparisons are allowed according to the following rules:

- If the strings have the same operational length, a normal comparison operation will be done.
- If the operational lengths of the strings are different, blanks will be assumed to follow the shorter string.

Here is an example that compares strings:

```
VAR
  ST4 : STRING[4];
  ST8 : STRING[8];
BEGIN
  ST4 := 'BLUE';
  ST8 := 'LAVENDER';
  IF ST8 > ST4 THEN
    WRITELN('Pass')  {this will pass}
  ELSE
    WRITELN('Fail');
  ST8 := ST4;        {ST8 is now 'BLUE'}
  IF ST8 = ST4 THEN
    WRITELN('Pass')  {this will pass}
  ELSE
    WRITELN('Fail')
END.
```

19.2

Concatenating Strings

Prime Pascal's concatenation operator (+) concatenates two strings into one string. The concatenation operator is a Prime extension. There is no concatenation operator in standard Pascal.

The resultant length of the newly formed string equals the sum of the operational lengths of the two concatenated strings. Either or both of the strings may be a character literal string.

Here is an example that uses concatenation:

```
VAR
  ST2 : STRING[2];
  ST4 : STRING[4];
  ST6 : STRING[6];
  ST11 : STRING[11];
  AR2 : ARRAY[1..2] OF CHAR;
  AR4 : ARRAY[1..4] OF CHAR;
  AR6 : ARRAY[1..6] OF CHAR;
BEGIN
  ST2 := 'HI';
  ST4 := 'BALL';
  AR2 := 'GO';
  AR4 := 'BLUE';
  ST6 := ST2 + ST4;  {ST6 equals 'HIBALL'}
  ST6 := ST4 + ST2;  {ST6 equals 'BALLHI'}
  ST4 := ST2 + ST4;  {ST4 equals 'HIBA'}
  ST11 := ST2 + ST4 + 'HELLO';  {ST11 equals 'HIHIBAHELLO'}
  ST4 := ST2; {ST4 equals 'HI'}
  ST4 := ST2 + ST4;  {ST4 equals 'HIHI'}
  ST6 := STR(AR2) + STR(AR4);  {ST6 equals 'GOBLUE'}
  AR6 := UNSTR(STR(AR4) + STR(AR2));  {AR6 equals 'BLUEGO'}
  ST2 := 'PA';
  ST4 := 'SCAL';
  ST6 := ST2 + ST4;
  IF ST6 = 'PASCAL' THEN
    WRITELN('Pass')  {this passes}
  ELSE
    WRITELN('Fail');
  IF ST6 = ST2 + ST4 THEN
    WRITELN ('Pass again')  {this passes}
  ELSE
    WRITELN ('Fail');
  IF ST6 = 'PA' + 'SCAL' THEN
    WRITELN ('This works too')  {this passes}
  ELSE
    WRITELN ('Fail');
  AR6 := UNSTR(ST6);
  IF AR6 = 'PASCAL' THEN
    WRITELN ('Passes to array')  {this passes}
  ELSE
    WRITELN ('Array fails');
  IF AR6 = UNSTR(ST2 + ST4) THEN
    WRITELN ('Passes to array again')  {this passes}
  ELSE
    WRITELN ('Array fails')
END.
```

19.2

The concatenation operator is also discussed in Chapter 7.

Reading and Writing Strings

When reading a string, you can enter any number of characters up to the maximum length. Consider the following program, which contains a READ statement:

```
VAR
  ST10 : STRING[10];
BEGIN
  READ(ST10)
END.
```

If the input were:

```
ABC(carriage return)
```

the program would assign 'ABC' to ST10 when the carriage return is entered.

If the input were:

```
ABCDEFGHIJK
```

19.2

the program would complete execution the moment the 'K' character was typed, because the 'J' character is the tenth character.

When you use a READLN statement, the number of characters before the carriage return becomes the operational length of the string up to the maximum length of that string.

Consider the following example:

```
VAR
  ST5 : STRING[5];
BEGIN
  READLN(ST5)
END.
```

If the input were:

```
ABC(carriage return)
```

the value of ST5 would be 'ABC' and ST5 would have an operational length of 3 characters.

If the input were:

```
ABCDE(carriage return)
```

or

```
ABCDEFGHIJKLM(carriage return)
```

the value of ST5 would be 'ABCDE' and the operational length of ST5 would be 5 characters. In either case, the program would not terminate until the carriage return was typed.

When reading two strings with one READ or READLN statement, you must enter all of the characters of the first string, up to its maximum length, before you can begin entering characters for the second string. Consider the following example:

```
VAR
  ST1, ST2 : STRING[10];
BEGIN
  READ(ST1, ST2)
END.
```

If the input were:

19.2

```
ABCDEFGHIJKLM
```

the characters 'ABCDEFGHIJ' would be assigned to ST1, and 'KLM' would be assigned to ST2. In order to assign characters to ST2, 10 characters must be assigned to ST1.

If you enter less than 10 characters, or if you enter only 10 characters, then the null string is assigned to ST2. (Null strings cannot be written out.)

When a string is written, the default field width is the operational length of the string. If a field width is specified, and the width of the field to be printed is greater than the operational length of the string, then the string is right justified in the field and blank padded on the left. If the specified field width is too small, then only the specified number of characters will be printed.

Here is an example of writing strings with different field widths:

```
VAR
  ST10 : STRING[10];
BEGIN
  ST10 := 'ABCDEFGH';  {eight characters}
  WRITELN(ST10);
  WRITELN(ST10:12);
  WRITELN(ST10:2)
END.
```

The output will look like this:

```
ABCDEFGH
    ABCDEFGH
AB
```

Here is another example that reads and writes strings to and from the terminal and PRIMOS data files:

```
VAR
  ST5 : STRING[5];
  ST10 : STRING[10];
  STRINGINPUT : FILE OF CHAR;
  STRINGOUTPUT : FILE OF CHAR;
BEGIN
  WRITE('Enter an ST5 value: ');
  READLN(ST5);
  WRITELN(ST5);
  WRITE('Enter an ST10 value: ');
  READLN(ST10);
  WRITELN(ST10);
  WRITELN(ST5 + ST10);
  RESET(STRINGINPUT, 'STINPUT');
  READLN(STRINGINPUT, ST5);
  REWRITE(STRINGOUTPUT, 'STOUTPUT');
  WRITELN(STRINGOUTPUT, ST5);
  READLN(STRINGINPUT, ST10);
  WRITELN(STRINGOUTPUT, ST10);
  WRITELN(STRINGOUTPUT, ST5 + ST10);
  CLOSE(STRINGINPUT);
  CLOSE(STRINGOUTPUT)
END.
```

19.2

Passing Strings to Procedures and Functions

Strings can be passed as parameters to procedures and functions. They may be passed by value or by reference and may return as arguments from functions.

The STRING assignment rules, given earlier in this chapter, apply to passing strings to procedures and functions.

Here is an example that passes strings to procedures and functions:

```
TYPE
  STRING_6 = STRING[6];
  STRING_3 = STRING[3];
  STRING_10 = STRING[10];
VAR
  GLOBAL_10 : STRING_10;
  GLOBAL_6 : STRING_6;
PROCEDURE PROC1(S : STRING_6);  {GLOBAL_10 is passed to S}
  BEGIN                         {and is truncated to 'TESTIN'}
    WRITELN(S)                  {'TESTIN' will be written}
  END;
PROCEDURE PROC2(VAR S : STRING_6);  {GLOBAL_10 is assigned to}
  BEGIN                             {the parameter GLOBAL_6}
    S := GLOBAL_10
  END;
FUNCTION FUNC(S : STRING_6) : STRING_3;  {GLOBAL_10 becomes}
  BEGIN                                  {substring 'TIN'}
    FUNC := SUBSTR(S, 4, 3)              {inside function}
  END;
BEGIN {main}
  GLOBAL_10 := 'TESTING';
  PROC1(GLOBAL_10);
  PROC2(GLOBAL_6);
  WRITELN(GLOBAL_6);  {'TESTIN' will be written}
  GLOBAL_10 := FUNC(GLOBAL_10);
  WRITELN(GLOBAL_10)  {'TIN' will be written}
END.
```

For complete information on procedures and functions, see Chapter 9.

String Functions

There are seven other built-in functions that manipulate strings in addition to the STR and UNSTR functions. All of these functions are Prime extensions. They are:

- LENGTH
- INDEX
- SUBSTR
- DELETE
- INSERT
- TRIM
- LTRIM

The LENGTH Function: This function takes a string as an argument and returns an integer that is the operational length of the string. A string literal may not be used with this function.

The INDEX Function: This function takes two strings as arguments. It searches the first string to determine if it contains the second string. The first argument, therefore, is the string to be searched. The second argument is the string to be searched for. The function returns an integer that gives the position in the first string that indicates the beginning of the second string. If the second string is not found in the first string, a zero is returned. The first argument must be a string and not a string literal. The second argument may be a string, a string literal, or a character.

The SUBSTR Function: This function takes three arguments -- a string and two integers. It yields a substring of the first argument, which is a string. The second argument is the starting position of the substring in that string. The third argument is the desired length of the substring. The function returns a string. The first argument must be a string and not a string literal.

The DELETE Function: This function takes three parameters -- a string and two integers. It deletes a specified substring within the given string, and returns a string. The function takes the first argument, the string, starting at the position specified by the first integer, and deletes the number of characters specified by the second integer. The first argument must be a string, not a string literal. 19.2

The INSERT Function: This function takes three arguments -- two strings and an integer. It inserts the second string into the first string, and returns a string. The integer specifies the position in the first string where the second string is to be inserted. The first argument must be a string and not a string literal. The second argument may be a string, a string literal, or a character.

The TRIM Function: This function takes a string as an argument and returns a string. It removes all trailing blanks. The argument must be a string, not a string literal.

The LTRIM Function: This function takes a string as an argument and returns a string. It removes all leading blanks. The argument must be a string, not a string literal.

Here is an example that uses all these functions:

```
VAR
  ST8 : STRING[8];
  ST10 : STRING[10];
  I, J, K : INTEGER;
BEGIN
  ST10 := 'ABCDEF';
  I := LENGTH(ST10);  {I equals 6}
  ST8 := 'CDE';
  I := INDEX(ST10, ST8);  {I equals 3}
  J := INDEX(ST8, ST10);  {J equals 0}
  ST8 := SUBSTR(ST10, 3, 2);  {ST8 equals 'CD'}
  ST8 := DELETE(ST10, 3, 2);  {ST8 equals 'ABEF'}
  ST8 := INSERT(ST8, 'HI', 2);  {ST8 equals 'AHIBEF'}
  ST10 := ' A  B  C  ';  {10 characters}
  ST10 := TRIM(ST10);  {ST10 equals ' A  B  C' - 8 characters}
  ST10 := LTRIM(ST10)  {ST10 = 'A  B  C' - 7 characters}
END.
```

19.2

THE ARRAY TYPE

An array is a data structure that is a collection of elements of identical type. This group of elements is identified by one variable name. An element of an array is accessed by its location within the array. For example, an array can be declared:

```
VAR
  A : ARRAY[1..10] OF INTEGER;
```

The array called "A" has 10 consecutive integer elements. The first element is A[1], the second is A[2], and so on. The number in square brackets that identifies the array element is called the index. You can read and write an array element this way:

```
READ(A[1]);
WRITE(A[1]);
```

To read and write all of the elements, you can say:

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

Here is an example of how ARRAY types are declared within a TYPE declaration:

```
TYPE
  NUMBERS = ARRAY[1..50] OF INTEGER;
```

The new data type (ARRAY type), which has 50 integer elements, is called NUMBERS. In the VAR declarations, therefore, you can declare variables to be of type NUMBERS. For example:

```
VAR
  X, Y : NUMBERS;
```

The identifiers X and Y are arrays, each having 50 integer elements.

The type of array index, termed the index type, must be a scalar data type other than REAL or LONGREAL. The data type of the array itself can be any data type, including arrays and other structured types.

Three examples of arrays follow:

Example 1:

```
TYPE
  SAMPLE1 = ARRAY[1..100] OF REAL;
VAR
  R : SAMPLE1;
```

This declaration indicates that R will be a 100-element array of REAL. The first element will be accessed by R[1], the second by R[2], and the hundredth by R[100].

Example 2:

```
TYPE
  DAYS = (MON, TUE, WED, THUR, FRI, SAT, SUN);
  SAMPLE2 = ARRAY[DAYS] OF INTEGER;
VAR
  D : SAMPLE2;
```

These declarations indicate that D will be a seven-element array of INTEGER. The first element will be referenced by D[MON], the second by D[TUE], and the seventh by D[SUN].

Example 3:

```
TYPE
  EXAMSCORE = 0..100;
VAR
  STUDENTSCORE : ARRAY[1..50] OF EXAMSCORE;
BEGIN
  STUDENTSCORE[1] := 98
END.
```

To define arrays as external in a program, that is, so that they can be used by external subprograms, you can use the following technique:

```
VAR

{Local variables:}

  D: ARRAY[1..10] OF INTEGER;
  Y: (BLUE, PINK, YELLOW, RED);

{$E+ Defines external variables}

  A: ARRAY[-32767..+32767] OF REAL;
  C: ARRAY[-32767..+32767] OF CHAR;

{$E- Ends external definitions}
```

INTEGER is allowed as an array index. The array must be declared as an external array with the {$E+} compiler switch. For example:

```
VAR
  {$E+}
  A : ARRAY[INTEGER] OF INTEGER;
  {$E-}
```

18.3

The declaration above produces the same results as:

```
A : ARRAY[-32768..+32767] OF INTEGER;
```

LONGINTEGER is <u>not</u> allowed as an array index.

The maximum size of an array is 64K words long. If an array is declared as external with the {$E+} compiler switch, then more space will be allocated to the array.

<u>Note</u>

The {$E+} compiler switch is similar in function to the PL/I EXTERNAL attribute or the FORTRAN COMMON block. The {$E+} compiler switch is discussed in Chapters 2 and 9.

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

Unlike a STRING type character string, which can have values of a varying number of character elements, an array of characters must contain a precise number of character elements.

A variable receiving an array-of-character assignment or a formal parameter receiving an array-of-character value must be declared to have the precise number of elements as the array being assigned or passed. 19.2

Therefore, if you want to manipulate character strings of varying length, use STRING. If you want to manipulate character strings that always contain a precise number of elements, use ARRAY OF CHAR. (The STRING type is a Prime extension, and it is fully discussed earlier in this chapter.)

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.

The result of the UNSTR function is an array of characters or a single character. The number of characters in the newly formed array is determined by context. That is, the context of whatever array length is expected determines the length. The result of an UNSTR function may be used anywhere an array of characters is expected. Here are some specific rules governing the use of the UNSTR function:

- If the result of the UNSTR function is being assigned to an array of characters, then that result will have the same number of characters as the receiving array of characters.
- If the result of the UNSTR function is being passed to a procedure or function, then that result will have the same number of characters as the formal parameter.
- If the result of the UNSTR function is being compared to an array of characters, then that result will have the same number of characters as the array of characters to which it is being compared.
- If the UNSTR function is used in any other context, the length of the resultant array will be the same as the operational length of the string argument.

Here is an example that converts strings and arrays of characters to one another using STR and UNSTR:

```
VAR
  ST4 : STRING[4];
  ST8 : STRING[8];
  AR4 : ARRAY[1..4] OF CHAR;
  AR8 : ARRAY[1..8] OF CHAR;
BEGIN
  AR4 := 'JUNK';
  ST4 := STR(AR4);  {value of ST4 is 'JUNK'}
  AR4 := 'BLUE';
  ST8 := STR(AR4);  {value of ST8 is 'BLUE'}
  AR8 := 'LAVENDER';
  ST4 := STR(AR8);  {value of ST4 is 'LAVE'}
  ST4 := 'JUNK';
  AR4 := UNSTR(ST4);  {value of AR4 is 'JUNK'}
  AR8 := UNSTR(ST4);  {value of AR8 is 'JUNK    '}
  ST8 := 'LAVENDER';
  AR4 := UNSTR(ST8)  {value of AR4 is 'LAVE'}
END.
```

Arrays of characters are discussed later in this chapter.

If you are using a READ, all 30 characters must be typed in. The remainder of the array will not be padded with blanks, according to the standard Pascal definition of READ.

Multidimensional arrays can be read more easily with Prime's ARRAY OF CHAR:

```
VAR
  A : ARRAY[1..10, 1..200] OF CHAR;
BEGIN
  READLN(A[1]);
```

19.1

This will read the first row of 200 characters into A[1]. A READ(A) statement would generate an error because you would be trying to read an array of strings, and not a single string.

For more information on multidimensional arrays, see the discussion that follows.

Multidimensional Arrays: As defined in the previous section, the type of an array can be any data type. If the type of an array is an ARRAY type or a sequence of two or more ARRAY types, the array is a multidimensional array. Example:

```
CONST
  SIZE = 100;
VAR
  SAMPLE : ARRAY[BOOLEAN] OF ARRAY[1..10] OF ARRAY[SIZE] OF REAL;
```

The above declaration can be simplified to more convenient forms:

```
VAR
  SAMPLE : ARRAY[BOOLEAN, 1..10, SIZE] OF REAL;
```

or

```
  SAMPLE : ARRAY[BOOLEAN] OF ARRAY[1..10, SIZE] OF REAL;
```

or

```
  SAMPLE : ARRAY[BOOLEAN, 1..10] OF ARRAY[SIZE] OF REAL;
```

An array can have up to eight dimensions.

In general, to create a multidimensional array, use the following type definition:

```
TYPE type-identifier = ARRAY [t1, t2,...] OF base-type;
```

where t1, t2, etc. are index types. If three index types are specified, the ARRAY type is called three-dimensional, and an array element is designated by three indexes. For example:

```
CONST
   NUM_OF_CLASSES = 3; {3 classes}
   NUM_OF_STUDENTS = 20; {20 students in each class}
   NUM_OF_EXAMS = 4; {Each student took 4 exams}
TYPE
   SCORE = ARRAY [1..NUM_OF_CLASSES, 1..NUM_OF_STUDENTS,
                  1..NUM_OF_EXAMS] OF INTEGER;
VAR
   STUDENTSCORE : SCORE;
```

STUDENTSCORE[3, 20, 4] would designate the fourth exam of the twentieth student in class number 3.

STUDENTSCORE[2, 10, 3] would designate the third exam of the tenth student in class number 2.

The RECORD Type

A record is a structure consisting of a fixed number of elements, called fields, which may be of different data types. Each record's field has a name, called the field identifier.

To define a RECORD structure, use the following TYPE declaration:

```
TYPE record-identifer = RECORD
                          field-identifier-1: type;
                                    .
                                    .
                                    .
                          field-identifier-n: type
                        END;
```

where record-identifier is the name given to the entire record. Each field-identifier and its associated type, which can be any type, even another RECORD type, are listed between the keywords RECORD and END.

Example 1:

```
TYPE
  PERSON = RECORD
             NAME : ARRAY [1..25] OF CHAR;
             AGE : 0..99;
             SEX : (MALE, FEMALE);
             SOC_NUM : LONGINTEGER
           END;
```

Example 2:

```
TYPE
  CUSTOMER_RECORD =
    RECORD
      NAME : ARRAY [1..30] OF CHAR;
      ID_NUM : INTEGER;
      INVOICE_DATE : RECORD
                       MONTH : (JAN, FEB, MAR, APR,
                                MAY, JUN, JUL, AUG,
                                SEP, OCT, NOV, DEC);
                       DAY_OF_MON : 1..31
                     END; {OF the INVOICE_DATE record}
       DISCOUNT, AMT_PAID : REAL
     END; {OF the CUSTOMER_RECORD}
```

Example 3:

```
TYPE DATE = RECORD
              DAYOFWEEK : (SUN, MON, TUE, WED,
                           THUR, FRI, SAT);
              MONTH     : (JAN, FEB, MAR, APR,
                           MAY, JUN, JUL, AUG,
                           SEP, OCT, NOV, DEC);
              DAYOFMON  : 1..31;
              YEAR      : INTEGER
            END;

    FAMILY = (FATHER, MOTHER, BROTHER, SISTER);

 VAR
   DATE1, DATE2 : DATE;
   BIRTHDAY : ARRAY[FAMILY] OF DATE;
```

To access a particular record element, follow the name of the variable by a period and the name of the element:

```
record-variable.field-identifier
```

Using Example 3 above, if DATE1 is to contain the date:

Tuesday, July 15, 1983

the following assignment statements will be written:

```
DATE1.DAYOFWEEK := TUE;
DATE1.MONTH     := JUL;
DATE1.DAYOFMON  := 15;
DATE1.YEAR      := 1983;
```

If the BIRTHDAY of SISTER is:

Saturday, December 6, 1975

the following assignment statements will be written:

```
BIRTHDAY[SISTER].DAYOFWEEK := SAT;
BIRTHDAY[SISTER].MONTH     := DEC;
BIRTHDAY[SISTER].DAYOFMON  := 6;
BIRTHDAY[SISTER].YEAR      := 1975;
```

The maximum size of a record is 64K words. If a record is declared as external with the {$E+} compiler switch, then more space will be allocated to the record. (For more information on the {$E+} compiler switch, see Chapters 2 and 9.)

The references to elements in a record structure can be simplified by using the WITH statement. The general form of the WITH statement is:

WITH record-variable-1 [, record-variable-2]...DO statement

Within the statement after DO, record elements may be referred to by field identifiers only. This form of record access allows the compiler to generate more efficient code and allows you to write more readable code. WITH should be used when a large number of components of a record is to be accessed.

Using the WITH statement, you would write the previous assignment statements as follows:

```
WITH DATE1 DO
  BEGIN
    DAYOFWEEK := TUE;
    MONTH     := JUL;
    DAYOFMON  := 15;
    YEAR      := 1983
  END;
```

and

```
WITH BIRTHDAY[SISTER] DO
  BEGIN
    DAYOFWEEK := SAT;
    MONTH     := DEC;
    DAYOFMON  := 6;
    YEAR      := 1975
  END;
```

Note

The WITH statement is also discussed in Chapter 8.

Records with Variants: In Pascal, different record values of the same RECORD type need not have the same fields. In most cases, each of these records can be divided into two parts -- a fixed part, which has fields common to all these records, and a variant part, which has fields varying from record to record. The fixed part must precede the variant part.

To define records with variants, use the following RECORD type definition:

```
TYPE  record-identifier =
        RECORD
           [field-identifier-1: type;]...              {fixed part}
           [CASE [tag-field:] tag-type-identifier OF  {variant part}
              variant-1 [; variant-2]...]
        END;
```

Note

Variant field values share the same storage area. Therefore, when a value of one field is assigned, it replaces or "overlays" the previously assigned field value in storage. (See Example 4.)

Example 1:

```
TYPE PERSON = RECORD
                L_NAME, F_NAME: ARRAY[1..20] OF CHAR;
                AGE: 0..100;
                SEX: (MALE, FEMALE);
                CASE MARRIED: BOOLEAN OF
                  TRUE: (SPOUSE_NAME: ARRAY[1..40] OF CHAR;
                         SPOUSE_AGE: 0..100);
                  FALSE: ( )  {No variant fields for this case}
              END;
```

If a person is married, the field MARRIED, called the tag-field, will be TRUE, and two additional fields, called variant fields, will exist — SPOUSE_NAME and SPOUSE_AGE. These variant fields will not exist if MARRIED is FALSE.

Example 2:

```
TYPE PERSON = RECORD
                L_NAME, F_NAME: ARRAY[1..20] OF CHAR;
                AGE: 0..100;
                SEX: (MALE, FEMALE);
                MARRIED: BOOLEAN;
                CASE BOOLEAN OF
                  TRUE: (SPOUSE_NAME: ARRAY[1..40] OF CHAR;
                         SPOUSE_AGE: 0..100)
              END;
```

Although this is a valid example of a RECORD type definition, its usage is not advised. The declaration of the tag-field in the CASE clause and the definition of every possible value of the tag-field as shown in Example 1 give better program readability.

Example 3:

```
TYPE
  SHAPE = (POINT, LINE, CIRCLE);
  FIGURE = RECORD
             CASE TAG: SHAPE OF
               LINE: (M, B: REAL);
               CIRCLE: (A, C: REAL; RADIUS: REAL);
               POINT: (Xo, Yo: REAL)
           END;
VAR
  V : FIGURE;
BEGIN
  V.TAG := LINE;
  TAG := LINE    {This assignment is invalid.}
END.
```

Example 4:

```
TYPE
  DATA = (INT, BOOL, CH);
  DATATYPE = RECORD
               CASE DATA OF
                  INT: (INTERVALUE: INTEGER);
                 BOOL: (BOOLVALUE: BOOLEAN);
                   CH: (CHVALUE: CHAR)
             END;
VAR
  DATAVALUE: DATATYPE;
BEGIN
  DATAVALUE.INTERVALUE := 100;
  WRITELN(DATAVALUE.INTERVALUE);
  DATAVALUE.BOOLVALUE := TRUE;
  WRITELN(DATAVALUE.BOOLVALUE);
  DATAVALUE.CHVALUE := 'A'
  WRITELN(DATAVALUE.CHVALUE);
  WRITELN(DATAVALUE.INTERVALUE);  {This will not output 100}
                                  {because its storage space}
                                  {has been overlaid with}
                                  {DATAVALUE.CHVALUE}
END.
```

Example 5:

```
TYPE
  EMPTY = RECORD  {The record EMPTY contains no fields; therefore,}
          END;    {it has a null value.}
```

Note

The CASE statement is discussed in Chapter 8.

The SET Type

A set is a collection of elements that are of the same data type, termed the base type. A base type can be any scalar data type other than REAL or LONGREAL. To create a SET type, use the following type definition: | 19.1

```
TYPE type-identifier = SET OF base-type;
```

The type-identifier is the name of a new SET type. You cannot have more than 256 elements in a set in Prime Pascal. Example:

```
TYPE
  LETTERS = SET OF 'A'..'Z';  {26 elements}
```

Variables of type LETTERS are declared in the variable declaration part:

```
VAR
  VOWELS, LIST, EMPTY, CH : LETTERS;
```

Similarly, a SET can also be declared this way:

```
VAR
  VOWELS, LIST, EMPTY, CH : SET OF 'A'..'Z';
```

Each variable above is a set whose members are chosen from the alphabetic characters 'A' to 'Z'. Set members are set constants that are always presented in a pair of square brackets []. Values of SET constants can be assigned to the variables by following assignment statements:

```
VOWELS := ['A', 'E', 'I', 'O', 'U'];   {Set members can be in}
CH     := ['B', 'C', 'A'];             {arbitrary order.}

EMPTY  := [ ];                         {A set may have no members}
                                       {at all; it is called the}
                                       {empty set.}

LIST   := ['F'..'P'];                  {If the set members are}
                                       {consecutive values of the}
                                       {base type, only the first}
                                       {and last need be specified}
                                       {in subrange form.}
```

There are three SET operators that operate on sets to produce new sets:

+	Set union
-	Set difference
*	Set intersection

The union of two sets is a set that contains all the members of both sets. The difference is a set that contains all the members of the first set that are not also members of the second set. The intersection is a set that contains all the values that belong to both sets. For example:

```
['Q'] + ['P', 'Q'] yields ['P', 'Q']

['A', 'B', 'E', 'F'] - ['B', 'C', 'D'] yields ['A', 'E', 'F']

['E', 'I', 'O'] * ['A', 'E'] yields ['E']
```

There are five relational operators that compare sets -- the four standard relational operators plus the set relational operator IN. The result of the comparison is a BOOLEAN value:

=	Equals
<>	Does not equal
<=	Is contained by
>=	Contains
IN	Is a member of

Examples:

```
['A', 'B'] = ['B', 'C'] is FALSE

['A', 'B'] <> ['B', 'C'] is TRUE

['B'] <= ['B', 'C'] is TRUE

['A'..'Z'] >= ['M'..'S'] is TRUE

'I' IN ['A', 'E', 'I', 'O', 'U'] is TRUE

'I' IN ['P', 'S', 'X'] is FALSE
```

The compiler will issue an error message when the SET operator IN is used on a non-SET type. Operators and operations are discussed in Chapter 7. 18.3

The FILE Type

A file is a collection or receptacle of data values that are external to a program. The values within each file must be of the same data type. Your Pascal program can take data from a file (called the input file), process it, and output data to another file (called the output file). These data files, which are PRIMOS files, can reside in your directory.

Data values within files cannot be accessed at random. They must be accessed sequentially, one at a time, in the order in which they appear in the file.

You can declare FILE data types, and declare variables as files using TYPE and VAR declarations. The format of the TYPE declaration is:

```
TYPE
  type-identifier = FILE OF base-type;
```

The format of the VAR declaration is:

```
VAR
  file-identifier : FILE OF base-type;
```

The base-type specifies the data type of data values in the file. It must not be a FILE type or a structured type with a FILE component. The type-identifier is the name of the new FILE data type. The file-identifier is the name of the file. For example:

```
TYPE
  INTEGERFILE = FILE OF INTEGER;    {A sequence of internal binary}
                                    {integers}

  ARR = ARRAY[1..15] OF REAL;
  ARRAYFILE = FILE OF ARR;          {A sequence of groups of 15}
                                    {internal binary real numbers}

  CHARFILE = FILE OF CHAR;          {A sequence of characters --}
                                    {a textfile}
```

Using these new FILE types you can declare:

```
VAR
  I: INTEGERFILE;     {The FILE type of each variable has already}
  A: ARRAYFILE;       {been defined in the previous example.}
  C: CHARFILE;
```

You can declare file variables directly, without creating your own FILE data types, by saying:

```
TYPE
  AR = ARRAY[1..15] OF REAL;
VAR
  I : FILE OF INTEGER;
  A : FILE OF AR;
  C : FILE OF CHAR;
```

There are five Input/Output procedures — RESET, REWRITE, GET, PUT, and CLOSE — and one BOOLEAN function, EOF (End-Of-File), that manipulate files. Five other I/O procedures -- READ, READLN, WRITE, WRITELN, and PAGE — and another BOOLEAN function, EOLN (End-Of-Line), manipulate textfiles (FILE OF CHAR), although READ and WRITE work on nontextfiles as well. Explanations of all of these procedures and functions, as well as complete information on Input/Output in Prime Pascal, are given in Chapter 10.

The TEXT File Type: There is a standard Pascal data type called TEXT. The TEXT type is a FILE type that is identical to FILE OF CHAR. Both TEXT and FILE OF CHAR are "textfiles" or files consisting of printable characters, including integers and real numbers. The following declarations, therefore, are identical:

```
VAR
  A : FILE OF CHAR;
```

```
VAR
  A : TEXT;
```

The following are also identical:

```
TYPE
  A = FILE OF CHAR;
```

```
TYPE
  A = TEXT;
```

A TEXT file or a FILE OF CHAR should not be confused with FILE OF INTEGER and FILE OF REAL, which are files composed of internal binary numbers, rather than printable characters.

A textfile may be subdivided into variable length lines. Each line in the file is separated from the next by the ASCII control character LF (Line Feed).

Note

Textfiles have a maximum of 256 characters per line.

PRIMOS Files Versus the Terminal: Files are usually thought of as PRIMOS files, where data can be read from or written out to a file that is in your directory. The data, however, can also be read from and written to your terminal. That is, the terminal itself is considered a data receptacle or "terminal" file. Upon execution of your program, you supply the data input at your terminal, and the computer can respond by sending back the data output to your terminal.

If you want to read and write data at your terminal, simply use READ, READLN, WRITE, and WRITELN statements. You do not have to declare any FILE types. For example:

```
PROGRAM Terminal;
VAR
  A : ARRAY[1..10] OF CHAR;
  B : ARRAY[1..20] OF CHAR;
BEGIN
  READLN(A);
  READLN(B);
  WRITELN(A);
  WRITELN(B)
END.
```

Upon execution of this program, the computer will wait for you to type in a 10-character string and a 20-character string on separate lines (ended by carriage returns). When it has the two strings, the computer will send them back to your terminal.

In Prime Pascal, if you want to read and write data to or from a PRIMOS textfile, you must not only declare a file to be TEXT or FILE OF CHAR, but also supply the name of the file in the RESET and REWRITE procedures. The naming of textfiles is a Prime extension. For example:

```
VAR
  CH : CHAR;
  INFILE, OUTFILE : TEXT;
BEGIN
  RESET(INFILE, 'INDATA');
  REWRITE(OUTFILE, 'OUTDATA');
  READ(INFILE, CH);
  WRITE(OUTFILE, CH);
  CLOSE(INFILE);
  CLOSE(OUTFILE)
END.
```

In the example above, RESET and REWRITE open the textfiles called INDATA and OUTDATA, which must be enclosed in single quotes. Read and write statements associate INFILE and OUTFILE with the actual names of the files, INDATA and OUTDATA. The CLOSE statements must be used to close the textfiles at the end of your program. For a complete explanation of RESET, REWRITE, and CLOSE, as well as other I/O features of Prime Pascal, see Chapter 10.

The Standard Textfiles INPUT and OUTPUT: Pascal has two standard textfiles, INPUT and OUTPUT. When you want to use PRIMOS data files, you do not have to declare INPUT and OUTPUT as TEXT or FILE OF CHAR. However, in the RESET and REWRITE statements, you must still give the name of the file. In the following example, the standard textfiles INPUT and OUTPUT are associated with two PRIMOS data files called INDATA and OUTDATA. Notice that INPUT and OUTPUT need not be declared as FILE OF CHAR:

```
PROGRAM Primefile;
VAR
  CH : CHAR;
BEGIN
  RESET(INPUT, 'INDATA');
  REWRITE(OUTPUT, 'OUTDATA');
  WHILE NOT EOLN(INPUT) DO
    BEGIN
      READ(INPUT, CH);
      WRITE(OUTPUT, CH)
    END;
END.
```

For more information on the standard textfiles INPUT and OUTPUT, as well as Prime I/O procedures, see Chapter 10.

THE POINTER TYPE

A pointer is a type of variable that references or points to a storage location, contrary to a scalar or structured type variable, which already has been allocated its own location in memory.

A scalar or structured type variable is accessible by its identifier. All the necessary memory is allocated for the variable at compile time. The memory taken up by the variable exists during the entire execution. These variables are called static variables.

A variable accessed by a pointer, on the other hand, is created and destroyed dynamically during the execution of the program. Accordingly, this variable is called a dynamic variable.

Dynamic variables are not explicitly declared in variable declarations and are not referenced by identifiers. Instead, they are referenced by pointers. A pointer is the storage address of a newly created dynamic variable, which is created by the predefined procedure NEW.

Pointer types are declared with this format:

```
TYPE type-identifier = ^base-type;
```

The type-identifier is the name of a pointer (dynamic) data type whose pointers will point to elements of the specified base-type. However, there is a special pointer constant, termed NIL, which is always an element of a pointer type and points to no element at all. Here is an example of a pointer declaration:

```
TYPE
  POINTER = ^INTEGER;
VAR
  P : POINTER;
```

P is a pointer that references or "points to" an element of the INTEGER type. P^ is the actual integer being pointed to. The difference is important.

There are four procedures, called dynamic allocation procedures, that create and destroy dynamic variables:

NEW(P)	Creates (or allocates) a new dynamic variable. A pointer to this new variable is assigned to the pointer variable P.
NEW(P, cl,...,cn)	Creates (or allocates) a new dynamic variable of RECORD type with variants. A pointer to this new variable is assigned to the pointer variable P. The variants of the variable (tag-field) correspond to the case-constants cl,...,cn. The case-constants must be listed contiguously and in the order of their declaration. They must not be changed during execution.
DISPOSE(P)	Indicates that the storage occupied by the variable P^ is no longer accessible. That storage becomes available for future use. P is then undefined. NEW(P) and DISPOSE(P) are complementary.
DISPOSE(P, cl,...,cn)	Indicates that the storage occupied by P^, which was allocated by the second form of NEW, is no longer accessible. That storage becomes available for future use. P is then undefined. The case-constants of both procedures must be identical.

Note

19.1 As of Rev. 19.1, Prime supports approximately 16 segments (1024K words) of dynamic storage.

Example 1:

```
PROGRAM POINTER_SAMPLE(INPUT, OUTPUT);
VAR
  PTR : ^INTEGER;          {PTR is a pointer  variable  bound to}
                           {type INTEGER.}
  I : INTEGER;
BEGIN
  FOR I := 1 TO 10 DO

    BEGIN
      NEW (PTR);           {Allocates a  variable of type  INTEGER}
                           {and stores its address in PTR.}
      PTR^ := I;           {PTR^  is  the  actual  variable  being}
                           {pointed to.  A value of I is  assigned}
                           {to this new variable.}
      DISPOSE (PTR)        {Destroys the variable PTR^ and returns}
                           {its storage for future use.}
    END
END.
```

Example 2:

```
{This example creates a linked list (or a chain) to which elements
can be added or deleted at random.  A linked list is essentially a
chain of RECORD elements, each of which has a POINTER field called
NEXT, which points to the next element in the chain.}

TYPE
  LINK = ^PERSON;
  PERSON = RECORD
             NEXT : LINK;
             NAME : CHAR
           END;
VAR
  ROOT, P : LINK;
  I : INTEGER;
  CH : CHAR;
BEGIN
  ROOT := NIL;
  FOR I := 1 TO 50 DO
    BEGIN
      READ(CH);
      NEW(P);
      P^.NAME := CH;          {A name code is stored in NAME.}
      P^.NEXT := ROOT;        {The sequence of these two statements}
      ROOT := P               {is a general algorithm for inserting}
                              {an  element at the  beginning of the}
                              {list.}
    END;
END.
```

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, integer, or concatenation. (The concatenation operator and the integer operators 19.2 | 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

Relational Operators

The relational operators are used to compare values of data types -- scalar, STRING, ARRAY OF CHAR, 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. | 19.2

Table 7-2
Relational Operators

Operator	Operation	Type of Operands
=	equality	SET, scalar, pointer, STRING, or ARRAY OF CHAR
<>	inequality	
<	less than	scalar, STRING, or ARRAY OF CHAR
>	greater than	
<=	less or equal	scalar, STRING, or ARRAY OF CHAR
<=	set inclusion ("is contained in")	SET
>=	greater or equal	scalar, STRING, 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

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 }
```

SET Operators

SET operators, listed below, operate on sets to produce new sets.

Operator	Operation
+	set union
-	set difference
*	set intersection

The union of two sets is a set that contains all the members of both sets. The difference is a set that contains all the members of the first set that are not also members of the second set. The intersection is a set that contains all the values that belong to both sets.

Here are some examples of SET operators.

First, let

```
x := ['A', 'E', 'O']

y := ['I', 'U', 'O']

z := ['A', 'B']
```

then

```
w := x + y    {w is ['A', 'E', 'I', 'O', 'U']}

w := x - y    {w is ['A', 'E']}

w := x * z    {w is ['A']}
```

Note

Five relational operators, =, <>, <=, >=, and IN, also apply to the SET data type; they produce BOOLEAN results. See Relational Operators.

BOOLEAN Operators

Boolean operators operate on BOOLEAN values to produce a BOOLEAN result, TRUE or FALSE. The operators are OR, AND, and NOT. In the following examples, P and Q are of type BOOLEAN.

As these examples indicate, if P is true or Q is true, then the expression P OR Q is true. If P is true and Q is true, then the expression P AND Q is true. (These expressions would be false otherwise.)

NOT Q negates the value of Q. If Q is true, then NOT Q is false. If Q is false, then NOT Q is true.

The OR Operator

P	Q	P OR Q
F	F	F
F	T	T
T	F	T
T	T	T

The AND Operator

P	Q	P AND Q
F	F	F
F	T	F
T	F	F
T	T	T

The NOT Operator

Q	NOT Q
F	T
T	F

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.

STRING Concatenation Operator

Prime Pascal's concatenation operator (+) concatenates two strings into one. The concatenation operator is a Prime extension. There is no concatenation operator in standard Pascal.

The concatenation operator works only on operands of the STRING data type. The STRING type is also a Prime extension. (See Chapter 6.) 19.2

The resultant length of the newly formed string equals the sum of the operational lengths of the two concatenated strings. Either or both of the strings may be a character literal string, enclosed in single quotes.

Here is an example that uses concatenation operators:

```
VAR
  ST2 : STRING[2];
  ST4 : STRING[4];
  ST6 : STRING[6];
  ST12 : STRING[12];
BEGIN
  ST2 := 'PA';
  ST4 := 'SCAL';
  ST6 := ST2 + ST4;  {value of ST6 is 'PASCAL'}
  ST12 := ST2 + ST4 + 'STRING'  {value of ST12 is 'PASCALSTRING'}
END.
```

19.2

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	Highest precedence (done first)
2. NOT, unary - and +	
3. *, /, DIV, MOD, AND, &	
4. +, -, OR, !	
5. =, <>, <, >, <=, >=, IN	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
  2   1   4    3    5             indicate the order in which
                                   the operations are performed.}

( ( 7 + A ) * 2 - 5 ) DIV 3 + A
     1     2   3      4    5
```

8
Statements

This chapter discusses the various types of executable statements in Prime Pascal. These statements, which specify algorithmic actions, comprise the executable part of a program.

SUMMARY OF STATEMENTS

The various types of Pascal statements are:

- Assignment Statement
- Procedure Statement
- Compound Statement
- Empty Statement
- Control Statements:

  REPEAT
  WHILE
  FOR
  IF
  CASE
  GOTO

- WITH Statement

ASSIGNMENT STATEMENT

An assignment statement assigns a value to a variable or a function identifier. The form of the statement is:

```
variable | function-identifier := expression;
```

The assignment operator := can be read "becomes" or "gets the value of". The expression on the right-hand side of the operator is evaluated and the value obtained becomes the current value of the variable or the function-identifier on the left-hand side of the operator.

A function-identifier is a function name. Within the function block of the function, it may appear on the left-hand side of the assignment operator. (See Chapter 9.)

A variable is represented by its name. Variables on the left-hand side of the assignment operator may or may not have been assigned values previously.

For example, if the user has made the following declarations in a program:

```
VAR
  CH : CHAR;
  R  : REAL;
  NUMBER, F, I, J, K : INTEGER;
```

then the following assignment statements are valid:

```
CH := '5';

NUMBER := ORD(CH) - ORD('0');

R := 123.3;

F := TRUNC(R) MOD 5;

I := F;

J := I + NUMBER;

K := J DIV 2;

I := SQR(K) - (I*J);
```

Assignment Compatibility

The data type of the expression on the right-hand side of the assignment operator must be compatible with the data type of the variable or function identifier on the left-hand side.

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.
- 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 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;
```

It is not necessary to place a semicolon after the statement that precedes the END delimiter. END is part of the compound statement, not a statement in itself. The use of a semicolon will generate an empty statement.

EMPTY STATEMENT

An empty statement denotes no action and, as its name implies, consists of no letters, digits, or punctuation symbols. Using hypothetical statements s1, s2, and s3, two examples follow.

Example 1:

```
CASE DAYS OF
  SUN:  ;  {An empty statement is right here.}
  MON, WED, FRI:  s1;
  TUE, THUR:  s2;
  SAT:  s3
END;
```

Example 2:

```
BEGIN
   READ(CH);
   WRITE(CH);  {The semicolon separates the WRITE procedure from}
END;           {the END so an empty  statement precedes the END.}
```

CONTROL STATEMENTS

Statements are normally executed in the order of their appearance in a program unit. However, it is often necessary to interrupt the normal processing of statements for a special purpose, such as the repeated processing of a sequence of statements or the execution of one group of statements as opposed to another. Control statements are used to alter the normal sequential execution of statements.

There are three types of control statements: repetitive statements, conditional statements, and unconditional statements.

Note

Control statements that are enclosed within other control statements are called "nested statements" and "nested loops". In the discussions that follow, some examples of nested control statements are given.

Repetitive Statements

A repetitive statement specifies that a certain group of statements is to be executed repeatedly. This repetition is called a loop. There are three types of repetitive statements — REPEAT, WHILE, and FOR.

REPEAT Statement: The form of the REPEAT statement is:

```
REPEAT statement-1 [; statement-2...] UNTIL boolean-expression;
```

The statement (or the sequence of statements) between the keywords REPEAT and UNTIL is executed repeatedly until the boolean-expression becomes true. The statement (or statement sequence) will be executed at least once, because the Boolean-expression is evaluated at the end of the cycle.

Example 1:

```
SPACE := ' ';
REPEAT
  READ(CH);
  WRITELN(CH)
UNTIL CH = SPACE;
```

Example 2:

```
REPEAT
   K := I MOD J;
   I := J;
   J := K
UNTIL J = 0;
```

It is not necessary to place a semicolon after the statement that immediately precedes UNTIL, because UNTIL is part of the statement, not a statement itself.

Note

In a REPEAT loop, the beginning and the end of the statements to be executed repeatedly are marked by the keywords REPEAT and UNTIL. Therefore, it is not necessary to use the keywords BEGIN and END to bracket the statement sequence. However, if BEGIN and END are used, it is not wrong, just redundant.

WHILE Statement: The form of the WHILE statement is:

```
WHILE boolean-expression DO statement;
```

The statement, which may be any statement (including a compound statement), is executed repeatedly while the boolean-expression is true. The boolean-expression is evaluated at the beginning of each cycle. If its value is false initially, the statement will not be executed at all.

Example 1:

```
WHILE A[I] <> X DO
  I := I + 1;
```

Example 2:

```
WHILE I>0 DO
  BEGIN
    IF ODD(I) THEN
      Z := Z * X;
    I := I DIV 2;
    X := SQR(X)
  END;
```

Example 3:

```
WHILE NOT EOF(INPUT) DO
  BEGIN
    WHILE NOT EOLN(INPUT) DO
      BEGIN
        READ(INPUT, CH);
        WRITE(OUTPUT, CH)
      END; {inner WHILE loop}
    READLN(INPUT)
  END; {outer WHILE loop}
```

Example 3 illustrates a loop within a loop or a nested loop.

A loop controlled by WHILE may be converted into a loop controlled by REPEAT. For example, the WHILE statement:

```
WHILE b DO body;
```

is equivalent to:

```
IF b THEN
  REPEAT
    body
  UNTIL NOT(b)
```

FOR Statement: A FOR statement causes a statement to be executed a specified number of times while a progression of values is assigned to a variable called the control variable of the FOR statement.

The general form of a FOR statement is:

```
FOR control-variable := initial-value TO final-value
   DO statement;
```

The alternative form (for decreasing initial value) is:

```
FOR control-variable := initial-value DOWNTO final-value
   DO statement;
```

The control-variable is increased to or decreased down to the next value in the loop. It counts, and therefore controls, the number of times the statements are executed.

The statement constitutes the body of the FOR loop. It may be any statement, including a compound statement.

The control-variable is of any scalar type (except REAL or LONGREAL). The initial-value and the final-value must be of a type compatible with the control-variable's type. Upon completion of the FOR statement, the control-variable is undefined.

When the FOR-TO form is used, the control-variable is tested to determine whether it is less than or equal to the final-value. If it is, the statement is executed, the control-variable is incremented by 1, and the cycle is repeated. If the elements of an enumerated type are being incremented, then the control-variable gets the value of the control-variable's successor.

In the FOR-DOWNTO form, the control-variable is tested to determine whether it is greater than or equal to the final value. If it is, the statement is executed, the control-variable is decremented by 1, and the cycle is repeated.

Here are some examples of FOR loops:

Example 1:

```
FOR I := 1 TO 20 DO
  BEGIN
    READ(A[I]);
    WRITE(A[I])
  END;
```

Example 2:

```
FOR I := 2 TO 63 DO
   IF A[I] > MAX THEN
     MAX := A[I];
```

Example 3:

```
FOR C := RED TO BLUE DO
   WRITELN(ORD(C));
```

Example 4:

```
FOR J := K DOWNTO 1 DO
   SUM := SUM + J;
```

Example 5:

```
FOR linenumber := 1 TO 20 DO
  BEGIN
    WRITE(linenumber);
    FOR I := 1 TO 60 DO
      BEGIN
        READ(CH);
        WRITE(CH)
      END; {inner FOR loop}
    READLN;
    WRITELN
  END; {outer FOR loop}
```

Example 5 above contains a nested FOR loop.

Conditional Statements

A conditional statement selects one of a number of alternate courses of action based upon the evaluation of a certain condition. There are two types of conditional statements — IF and CASE.

IF Statement: The form of an IF statement can be either:

```
IF boolean-expression THEN statement-1;

                or

IF boolean-expression THEN statement-1 ELSE statement-2;
```

where statement-1 and statement-2 may be any statement, including a compound statement.

When the IF-THEN form is used, statement-1 is executed only if the boolean-expression is true. Otherwise, statement-1 is bypassed and the next sequential statement is executed.

The IF-THEN-ELSE form allows the selection of one of two statements depending upon the value of the boolean-expression. If the boolean-expression is true, statement-1 is executed and statement-2 is bypassed. If the boolean-expression is false, statement-1 is bypassed and statement-2 is executed.

After the execution of the IF statement, control is passed to the next sequential statement.

Examples:

```
IF A > B THEN
  WRITELN(' A IS GREATER THAN B.');


IF X < 1.5 THEN
  BEGIN
    Z := X + Y;
    WRITELN(Z)
  END
ELSE
  Z := 1.5
```

Never put a semicolon immediately before ELSE because ELSE is not a statement. It is part of the IF statement.

An IF statement is nested within another IF statement whenever it appears as statement-1 or statement-2 or as part of statement-1 or statement-2. In these cases, any ELSE encountered must be paired with the immediately preceding IF, which has not been already paired with an

ELSE. The number of ELSEs in a nested IF structure need not be the same as the number of IFs. Here are two examples of nested IF statements:

Example 1:

```
IF X > 0 THEN
   IF Y > 0 THEN
     Y := Y + 1
   ELSE
     X := X + 1;
```

Example 2:

```
IF A < C THEN
   IF C < D THEN
     X := 1
   ELSE
   IF A < C THEN
     IF B < D THEN
       X := 2
     ELSE
       X := 3
   ELSE
   IF A < D THEN
     IF B < C THEN
       X := 4
     ELSE X := 5
   ELSE X := 6
ELSE X := 7;
```

CASE Statement: A CASE statement can be a much more efficient way to do multiple IF statements. A CASE statement is used to select one of a group of statements for execution depending on the value of an expression. The general form of a CASE statement is:

```
CASE expression OF
   case-constant-list-1 : statement-1;
                .
                .
                .
   case-constant-list-n : statement-n
   [; OTHERWISE statement]
END;
```

If the value of the expression matches any of the case-constants, then the statement or group of statements that corresponds to that case-constant is executed.

The expression can be of any scalar type, except REAL and LONGREAL. Multiple constants in a list are separated by commas. The case-constants can be written in any order.

Any statement, including a compound statement, may be controlled by a CASE statement.

Example 1:

```
VAR
   OPERATOR : (PLUS,MINUS,TIMES);
   X, Y : INTEGER;
BEGIN
   CASE OPERATOR OF
      PLUS :  X := X + Y;
      MINUS:  X := X - Y;
      TIMES:  X := X * Y
   END
END.
```

Example 2:

```
TYPE
   DAYS = (SUN,MON,TUE,WED,THUR,FRI,SAT);
VAR
   TODAY,TOMORROW,YESTERDAY : DAYS;
BEGIN
   FOR TODAY := SUN TO SAT DO
     BEGIN
        CASE TODAY OF
          SUN : BEGIN YESTERDAY := SAT ; TOMORROW := MON END;
          SAT : BEGIN YESTERDAY := FRI ; TOMORROW := SUN END;
          MON,TUE,WED,THUR,FRI:
                BEGIN
                  YESTERDAY := PRED(TODAY);
                  TOMORROW  := SUCC(TODAY)
                END
        END;  {CASE statement}
     WRITELN ('TODAY', ORD(TODAY), ' TOMORROW', ORD(TOMORROW),
              ' YESTERDAY', ORD(YESTERDAY))
     END  {FOR statement}
END.
```

Example 3:

```
TYPE
   WEEKDAYS = (SUN, MON, TUE, WED, THUR,
               FRI, SAT);
VAR
   DAYS : WEEKDAYS;
BEGIN
   CASE DAYS OF
      SUN, SAT : ;  {Since there is no action required for
                     SUN and SAT, the space before the
                     semicolon is an empty statement
                     producing no action.}
      MON, WED, FRI : statement-1;
      TUE, THUR : statement-2
   END
END.
```

When the CASE statement is executed, the expression must match one of the constant values; otherwise, the effect of the CASE statement is undefined.

However, you can use the OTHERWISE clause to execute an alternative statement, or group of statements, if no other statement in the case-constant list has been selected.

The OTHERWISE clause option is a Prime extension. OTHERWISE is a Prime keyword. This clause, if present, must immediately precede the keyword END, which terminates the CASE statement. For example:

```
VAR
   I : 1..20;
BEGIN
   CASE I OF
      1,20 : statement-1;
         4 : statement-2;
     5,7,9 : statement-3;
   3,11,17 : statement-4;
   OTHERWISE statement-5
   END
END.
```

In standard Pascal, the function of the OTHERWISE clause can be achieved by combining the standard CASE statement and an IF statement. The previous example of the OTHERWISE clause may be rewritten in standard Pascal as:

```
VAR
   I : 1..20;
   IF I IN [1,3,4,5,7,9,11,17,20] THEN
      CASE I OF
         1,20 : statement-1;
            4 : statement-2;
        5,7,9 : statement-3;
      3,11,17 : statement-4
      END
   ELSE
     statement-5;
```

Note

The CASE statement is different from the CASE clause in the variant part of a record. The CASE clause is discussed in Chapter 6.

Unconditional Statement

GOTO Statement: A GOTO statement is an unconditional statement that transfers control to the statement designated by the label, without testing or satisfying any condition. The form of the GOTO statement is:

```
GOTO label;
```

The label is an unsigned integer, which can be up to four digits long. You must declare the label prior to its appearance in the GOTO statement. The designated statement must be prefixed with the integer followed by a colon.

There are some restrictions on the use of a GOTO statement. A GOTO statement can transfer control within a block, or from an inner block to an outer block; it cannot transfer control from an outer block to an inner block. In particular, a GOTO statement may transfer control out of a subprogram (procedure or function), but not into one.

Example 1:

```
        .
        .
        .
LABEL 10;  {DATA} ...
PROCEDURE P1;
   LABEL 20, 30; ...
   BEGIN ...
   20:  IF s1 THEN GOTO 30;
        .
        .
        .
        GOTO 20;
   30: s5;
        .
        .
        .
   IF s7 THEN GOTO 10
   END; {P1}
BEGIN {DATA} ...
10: s9;  {A "GOTO 20" or "GOTO 30" is not}
         {permitted in DATA.}
        .
        .
        .
```

Example 2:

```
{This is an invalid example.}

PROGRAM Main;
PROCEDURE P;
   BEGIN
     5 : s1
   END;  {Of procedure P}
BEGIN {main}
   GOTO 5;  {Transferring control to an inner block}
        .   {is not permitted.}
        .
        .
END.  {Of program Main}
```

Note

In general, GOTO statements make a program algorithm hard to understand, and their use is discouraged. Therefore, a GOTO statement should be used only when it cannot be easily replaced by other available Pascal statements.

WITH STATEMENT

A particular field of a record is normally accessed by using both the name of the record and the name of the field, separated by a period. (See Chapter 6.)

However, if a field is accessed many times, the WITH statement can simplify this access by indicating the record variable name only once.

The form of a WITH statement is:

```
WITH record-variable-1 [,record-variable-2...]
  DO statement;
```

This form is equivalent to:

```
WITH record-variable-1 DO
  [WITH record-variable-2 DO]...statement
```

The statement may be any statement, including a compound statement. Within the statement, fields may be referred to only by field identifiers. For example:

```
WITH DATE DO
  IF MONTH = 12 THEN
    BEGIN
      MONTH := 1;
      YEAR := YEAR + 1
    END
  ELSE MONTH := MONTH + 1;
```

This is equivalent to:

```
IF DATE.MONTH = 12 THEN
  BEGIN
    DATE.MONTH := 1;
    DATE.YEAR := DATE.YEAR + 1
  END
ELSE DATE.MONTH := DATE.MONTH + 1;
```

9

Procedures and Functions

In addition to the main program, a Pascal program may contain a number of procedures and functions that can be collectively called subprograms. In Prime Pascal, a subprogram has the following features:

- A subprogram can be at most 64K words (128 bytes) in size.
- Values, called parameters, can be passed to and used by subprograms.
- Subprograms themselves can be passed as parameters to other subprograms. 19.1
- A subprogram can be separated from the main program (external subprogram) or embedded within the main program.
- Before it is fully defined, a subprogram can be referenced by other subprograms within the same Pascal program. However, the referenced subprogram must have been declared using the FORWARD attribute.
- An external, separately compiled subprogram can be written in any Prime supported language. If the subprogram is declared in a Pascal program using the EXTERN attribute, the subprogram can be referenced from any point within the Pascal program. (See Appendix D for more information on interfacing Pascal with other languages.)
- A subprogram can call itself. This process is called recursion.

This chapter discusses how to declare, invoke, and manipulate subprograms, and presents the following topics:

- Parameters
- Procedures
- Functions
- Forward procedures and functions
- External procedures and functions
- Recursive procedures and functions

PARAMETERS

Parameters allow information to be passed between the calling programs and the called programs. There are two kinds of parameters -- actual and formal.

Actual Parameters

An actual parameter, appearing in a subprogram call (procedure statement or function designator), is a variable whose location or value is passed to the formal parameter in the corresponding position in the called procedure or function heading. The actual parameters must agree in order, number, and data type, but not necessarily in name, with the formal parameters. For example, the following procedure statement has three actual parameters (X, Y, and I) that are passed to the procedure PLOT:

```
PLOT(X, Y, I);
```

Formal Parameters

Formal parameters are "placeholders" for the actual parameters. They mark the places where the values of the actual parameters are to be passed. Formal parameters are declared in the formal parameter list of a procedure or function heading. This list specifies the order, number, and data type of the corresponding actual parameters.

For example, the following procedure heading has three formal parameters (A, B, and J) that mark the "spots" or "places" where the values of the actual parameters are to be passed:

```
PROCEDURE PLOT (A, B : REAL; J : INTEGER);
```

In standard Pascal, there are four kinds of formal parameters -- value, variable, procedure, and function. (Procedures and functions can be passed as parameters. Passing procedures and functions is discussed later in this section.)

Value Parameters: If a formal parameter is not preceded by the keyword VAR, then it is a value parameter.

A value parameter is a variable that receives the value of its corresponding actual parameter from the procedure statement. When the subprogram is called, the value is passed to this variable so that the procedure can use this value to perform its operations. However, the value is never passed back to the main program. A value parameter is also known as a pass-by-value parameter.

When the subprogram is called, the current value of the actual parameter is passed to the variable. Although the subprogram can change the value in its operations, the subprogram does not change the value of the actual parameter in the calling program. Therefore, when the values of actual parameters need to be protected, value parameters are used. For example:

```
PROGRAM Parameters (OUTPUT);
VAR
  A, B : INTEGER;
PROCEDURE VALUE_PAR(I, J : INTEGER);
  BEGIN
    I := I + 1;  {I = 2}
    J := J + 2;  {J = 3}
    WRITELN(I, J)
  END;  {Procedure VALUE_PAR}
BEGIN {main program}
  A := 1; B := 1;
  VALUE_PAR(A,B);
  WRITELN(A, B)  {A=1, B=1}
END.
```

In the above example, each of the variables A and B has an integer value of 1. These values are passed to the variables I and J respectively. The values of I and J change to 2 and 3 when the procedure is executed, but the values of A and B remain at 1.

The data type of the value parameter must be compatible with the data type of the corresponding actual parameter.

Variable Parameters: If a formal parameter is preceded by the keyword VAR, then it is a variable parameter.

A variable parameter is also a variable that receives the value of its corresponding actual parameter. Unlike the value parameter, however, the variable parameter also causes changes to the actual parameter. That is, the value of the variable parameter and its address are passed back to the calling program. (Values of value parameters can pass only from the calling program to the subprogram.)

Variable parameters are also known as pass-by-reference parameters because only variables can be passed to the subprogram.

Here is an example of using variable parameters:

```
PROGRAM Parameters (OUTPUT);
VAR
  A, B : INTEGER;
PROCEDURE VAR_PAR(VAR I : INTEGER; J : INTEGER);
                              {I is a variable parameter}
                              {J is a value parameter.}
  BEGIN
    I := I + 6;  {I = 7}
    J := J + 3   {J = 4}
  END; {Procedure Var_Par}
BEGIN {main program}
  A := 1;
  B := 1;
  VAR_PAR(A, B);
  WRITELN(A, B)   {A = 7; B = 1}
END.
```

Caution

Do not pass a constant, another expression, or a function call to a variable parameter, or a compile time error will be generated. The following example is invalid:

```
PROGRAM Main; ...
PROCEDURE VAR_PAR(VAR X : INTEGER);
  BEGIN ... END;
BEGIN
   .
   .
   .
  VAR_PAR(10);   {This will generate a compile-time error}
   .
   .
   .
END.
```

Constants and expressions can be passed to value parameters, however.

ARRAY or RECORD Type Variable Parameters: The Prime Pascal compiler produces two types of object code. Ordinary code can address only within a segment. Boundary-spanning code can address across the boundary between one segment and the next.

Whenever an array or record extends across a segment boundary, all references to it must consist of boundary-spanning code. All references in the program to any array or record the compiler knows to be longer than a segment will automatically be compiled with boundary-spanning code. No special action is required of the user in this case.

However, when an ARRAY or RECORD type variable parameter appears in a subprogram, the compiler has no way of knowing the storage status of any corresponding ARRAY or RECORD type actual parameter when the subprogram is invoked. Therefore, the compiler cannot know whether to compile references to that variable parameter with ordinary or boundary-spanning code. You must inform the compiler of the correct action in this case, through use of the -BIG/-NOBIG compiler options.

When a subprogram is compiled without -BIG (-NOBIG is the default), ARRAY or RECORD type variable parameter references will generate ordinary code; the corresponding ARRAY or RECORD type actual parameter must then be contained within one segment.

When a subprogram is compiled with -BIG, all references it makes to any ARRAY or RECORD type variable parameter will generate boundary-spanning code; the corresponding ARRAY or RECORD type actual parameter may then span a segment boundary, though it need not do so.

Boundary-spanning code executes more slowly than ordinary code because it performs more complex address calculations. The -BIG option should therefore not be used unnecessarily.

Caution

Arrays or records associated with value parameters must not span segment boundaries. The following example is invalid:

```
TYPE
  LONGARRAY = ARRAY[-32767..32767] OF REAL;
      .
      .
      .
PROCEDURE X(A: LONGARRAY);
      .
      .
      .
```

Procedures and Functions as Parameters

In Pascal, you can declare and pass a procedure or function as a parameter. Any procedure or function can pass any other procedure or function as a parameter.

Declaring Procedures and Functions as Parameters: A procedure or function declaration must list another procedure or function as a formal parameter. For example:

```
PROCEDURE A (PROCEDURE X);
```

If the procedure or function that is being passed has parameters of its own, the number and types of parameters must also be listed. For example:

```
PROCEDURE A (PROCEDURE X(X1, X2 :  INTEGER));
```

If you are passing a function, the type that the function returns must also be given:

19.1

```
PROCEDURE A (FUNCTION Y(Y1, Y2 : INTEGER) : INTEGER);
```

A procedure or function parameter can even have other procedures and/or functions as parameters:

```
PROCEDURE A (FUNCTION Y(PROCEDURE Y1;
                        FUNCTION Y2 : CHAR) : INTEGER);
```

The procedure or function that is declared must match -- parameter for parameter, in number and type — the declaration for the procedure or function that is passed.

Passing Procedures and Functions as Parameters: The name of a procedure or function is passed the same way any variable is passed. For example:

```
ADD(SORT);
```

The procedure named SORT is passed to a procedure named ADD. The name SORT is passed without parameters, but the number and type of parameters declared in the SORT procedure must match those in the ADD declaration, parameter for parameter.

Here are some examples:

Example 1:

```
VAR
  I : INTEGER;
PROCEDURE ADD1;
  BEGIN {procedure ADD1}
    I := I + 1;
  END;
PROCEDURE CALLPROC(PROCEDURE X);
  BEGIN {procedure CALLPROC}
    X;
  END;
BEGIN {main program}
  I := 0;
  CALLPROC(ADD1)
END. {I = 1}
```

19.1

Example 2:

```
VAR
  I : INTEGER;
FUNCTION ADD10 : INTEGER;
  BEGIN {function ADD10}
    ADD10 := 10
  END;
FUNCTION CALLF(FUNCTION X : INTEGER) : INTEGER;
  BEGIN {function CALLF}
    CALLF := X + X
  END;
BEGIN {main program}
  I := CALLF(ADD10)
END. {I = 20}
```

Example 3:

```
VAR
  I, J : INTEGER;
FUNCTION SQUARE(X : INTEGER) : INTEGER;
  BEGIN {function SQUARE}
    SQUARE := X * X
  END;
FUNCTION FCALLFUNC(FUNCTION Z(R : INTEGER) : INTEGER) : INTEGER;
  BEGIN {function FCALLFUNC}
    FCALLFUNC := Z(5)
  END;
PROCEDURE PCALLFUNC(FUNCTION Y(Q : INTEGER) : INTEGER);
  BEGIN {procedure PCALLFUNC}
    I := Y(5)
  END;
BEGIN {main program}
  J := FCALLFUNC(SQUARE);
  PCALLFUNC(SQUARE)
END. {I = 25 and J = 25}
```

Example 4:

19.1

```
VAR
  I, J, K : LONGINTEGER;
PROCEDURE ADD(X : LONGINTEGER);
  BEGIN {procedure ADD}
    I := I + X
  END;
FUNCTION FCALLPROC(PROCEDURE Z(R : LONGINTEGER)) : LONGINTEGER;
  VAR
    R : REAL;
  BEGIN {function FCALLPROC}
    R := 2.19;
    Z(ROUND(R)); {result of function call is passed}
    FCALLPROC := 10
  END;
PROCEDURE PCALLPROC(PROCEDURE Y(Q : LONGINTEGER));
  BEGIN {procedure PCALLPROC}
    Y(8);
    J := 10
  END;
BEGIN {main program}
  I := 0;
  K := FCALLPROC(ADD);
  PCALLPROC(ADD)
END. {I, J, and K each = 10}
```

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}
```

Note

A procedure or function cannot be a variable parameter. For example:

```
PROCEDURE X (VAR PROCEDURE Y);  {this is illegal}
```

Any attempt to use a procedure or function as a variable parameter will cause the VAR to be ignored and a severity 2 error to be given. 19.2

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.

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.

Conformant Array Parameters

Conformant array parameters have been added to ISO Standard Pascal in order to overcome a major difficulty created by Pascal's strict typing.

Note

> Conformant arrays are part of the ISO Pascal standard, but not part of the ANSI standard. If your programs must conform to the ANSI standard, do not use conformant array parameters.

Without conformant arrays, it is impossible to use the same procedure or function to handle arrays that have the same type and shape, but different bounds.

For instance, you might wish to write a procedure that sorts a one-dimensional array of integers -- an array that might consist of 10, 25, or 50 integers. In earlier versions of Pascal (and in ANSI Standard Pascal), there are two possible solutions. You can declare a single 50-integer array type, pad the smaller arrays to fit it, and thus have a single procedure to handle the sorting; or you can declare three different array types, but then have three sort procedures that are identical except for their array bounds.

Both of these solutions are cumbersome and involve wasted storage space. Conformant array parameters, however, make it possible both to declare the three different array types and to use a single procedure to sort the arrays.

A conformant array parameter definition occurs in the formal parameter specification of a procedure or function. Its general form is as follows:

```
arrpar : ARRAY [lowbound..highbound : ordtype] OF anytype
```

The parameter arrpar can be either a value parameter or a variable parameter. If arrpar is passed by value, the component type, anytype, cannot be a file or a type containing a file. The identifiers lowbound and highbound must be of an ordinal type, specified by ordtype.

The following program example uses conformant array parameters. PROCEDURE BUBBLE_SORT sorts arrays of different sizes by means of a simple sorting algorithm.

```
PROGRAM Sortarrays(INPUT, OUTPUT, INARR, OUTARR);
TYPE
  RANGE = -100..100;
  SMALLARRTYPE = ARRAY [1..10] OF INTEGER;
  BIGARRTYPE = ARRAY [-10..20] OF INTEGER;
VAR
  INARR, OUTARR : TEXT;
  SMALLARR : SMALLARRTYPE;
  BIGARR : BIGARRTYPE;
  I : RANGE;

PROCEDURE BUBBLE_SORT
        (VAR ARR : ARRAY [LOW..HIGH : RANGE] OF INTEGER);
  VAR
    I, J : RANGE;
    HOLD : INTEGER;
  BEGIN
    FOR I := LOW TO (HIGH - 1) DO
      FOR J := LOW TO (HIGH - 1) DO
        IF ARR[J] > ARR[J + 1] THEN
          BEGIN
            HOLD := ARR[J];
            ARR[J] := ARR[J + 1];
            ARR[J + 1] := HOLD
          END;
  END; {BUBBLE_SORT}

BEGIN
  FOR I := 1 TO 10 DO
    BEGIN
      WRITE('Enter an integer: ');
      READLN(SMALLARR[I]);
    END;
  BUBBLE_SORT(SMALLARR);     {first call to BUBBLE_SORT}
  FOR I := 1 TO 10 DO
    WRITELN(SMALLARR[I]);
  RESET(INARR);
  FOR I := -10 TO 20 DO
    READLN(INARR, BIGARR[I]);
  BUBBLE_SORT(BIGARR);       {second call to BUBBLE_SORT}
  REWRITE(OUTARR);
  FOR I := -10 TO 20 DO
    WRITELN(OUTARR, BIGARR[I])
END. {Sortarrays}
```

When procedure BUBBLE_SORT is invoked, the identifier LOW takes on the value of the lower bound of the actual parameter, and HIGH takes on the value of the upper bound of the actual parameter. When SMALLARR is sorted, LOW is 1 and HIGH is 10; when BIGARR is sorted, LOW is -10 and HIGH is 20.

The actual parameter must be compatible with the conformant array definition in the formal parameter. The actual parameter is compatible if all of the following conditions hold:

- It has the same number of dimensions as the conformant array parameter.
- Its index type is compatible with the index-type specification of the conformant array parameter.
- Its lower and upper bounds are within the range specified in the conformant array parameter. (If they are not, the error is detected only if the -RANGE option is specified.)
- Its component type is the same as or compatible with that of the conformant array parameter.

Conformant arrays can be multidimensional. For a multidimensional conformant array, put semicolons between the dimensions instead of commas. An ordinary multidimensional array is declared

```
VAR SOMEARR : ARRAY [1..10, 1..20] OF INTEGER;
```

but a multidimensional conformant array parameter is declared

```
PROCEDURE CONF(VAR SOMEARR : ARRAY [LO..HI : SMALLRANGE;
                        TOE..HEAD : LARGERANGE] OF INTEGER);
```

A conformant array can also be PACKED, but only the last dimension of a multidimensional conformant array can be PACKED. Moreover, there are severe limits on the operations you can perform on a PACKED ARRAY OF CHAR that is a conformant array parameter. In this situation, a PACKED ARRAY OF CHAR behaves like an ordinary array: you cannot use relational operators on it; you cannot read, write, or assign it as a unit; you cannot use the STR function on it; and you cannot assign the result of the UNSTR function to it. To avoid these limitations, use the STRING type instead of the PACKED ARRAY OF CHAR. (The STRING type is a Prime extension. See Chapter 6.)

If your program passes more than one array at a time as a conformant array parameter, you should be careful to observe the rules for assignment compatibility given in the ASSIGNMENT STATEMENT section of Chapter 8. The following program is invalid because the array variables are declared in different ways in the main program and in the formal parameter specification.

```
PROGRAM Badconf;
VAR
  A : ARRAY [1..5] OF INTEGER;        {These arrays are of}
  B : ARRAY [1..5] OF INTEGER;        { different types.}
{The declaration A, B : ARRAY [1..5] OF INTEGER would be valid.}
  COUNT : INTEGER;

PROCEDURE CONF (VAR C, D : ARRAY [LOW..HIGH : INTEGER] OF INTEGER);
{These two arrays share the same declaration, so they are of
 the same type.}
BEGIN
  C[LOW] := D[LOW]
END;

BEGIN
  FOR COUNT := 1 TO 5 DO    {initialize arrays}
    BEGIN
      A[COUNT] := 0;
      B[COUNT] := 2 * COUNT
    END;
  CONF(A, B)                {call the procedure}
END.
```

This program would receive the following error message:

```
   21        CONF(A, B)
                  ^
ERROR 240 SEVERITY 3 BEGINNING ON LINE 21
     This parameter must have the same type definition as the
     previous parameter.
```

You should also be careful not to assign a value to either of the array-bound identifiers. The assignment HIGH := 10, for example, would generate the error message

```
   10        HIGH := 10;
             ^
ERROR 178 SEVERITY 3 BEGINNING ON LINE 10
     FOR loop control variable or a conformant array bound
     identifier may not be assigned to, read in, or passed as
     a VAR parameter.
```

Procedures and Functions as Parameters

In Pascal, you can declare and pass a procedure or function as a parameter. Any procedure or function can pass any other procedure or function as a parameter.

Declaring Procedures and Functions as Parameters: A procedure or function declaration must list another procedure or function as a formal parameter. For example:

```
PROCEDURE A (PROCEDURE X);
```

If the procedure or function that is being passed has parameters of its own, you must also list the number and types of parameters. For example:

```
PROCEDURE A (PROCEDURE X(X1, X2 :  INTEGER));
```

If you are passing a function, you must state the type that the function returns:

```
PROCEDURE A (FUNCTION Y(Y1, Y2 : INTEGER) : INTEGER);
```

A procedure or function parameter can even have other procedures and/or functions as parameters:

```
PROCEDURE A (FUNCTION Y(PROCEDURE Y1;
                        FUNCTION Y2 : CHAR) : INTEGER);
```

The procedure or function that is declared must match -- parameter for parameter, in number and type -- the declaration for the procedure or function that is passed.

Passing Procedures and Functions as Parameters: The name of a procedure or function is passed the same way any variable is passed. For example:

```
ADD(SORT);
```

The procedure named SORT is passed to a procedure named ADD. The name SORT is passed without parameters, but the number and type of parameters declared in the SORT procedure must match those in the ADD declaration, parameter for parameter.

Five examples of procedure and function parameters follow.

Example 1:

{This program invokes a procedure that invokes another procedure.}

```
PROGRAM Call1(OUTPUT);
VAR
  I : INTEGER;

PROCEDURE ADD1;
  BEGIN
    I := I + 1;
  END;

PROCEDURE CALLPROC(PROCEDURE X);
  BEGIN
    X;        {Invokes procedure X}
  END;

BEGIN {main program}
  I := 0;
  CALLPROC(ADD1);  {PROCEDURE ADD1 is the actual parameter}
  WRITELN(I)       {I = 1}
END.
```

Example 2:

{This example invokes functions rather than procedures. FUNCTION ADDF adds together the values returned by calling FUNCTION VAL10 twice.)

```
PROGRAM Call2(OUTPUT);
VAR
  I : INTEGER;

FUNCTION VAL10 : INTEGER;
  BEGIN
    VAL10 := 10       {Function value is 10}
  END;

FUNCTION ADDF(FUNCTION X : INTEGER) : INTEGER;
  BEGIN
    ADDF := X + X  {Adds two function values together}
  END;

BEGIN {main program}
  I := ADDF(VAL10);  {Actual parameter is FUNCTION VAL10}
  WRITELN(I)         {I = 20}
END.
```

Example 3:

{This example uses both a procedure and a function as parameters.}

```
PROGRAM Call3(OUTPUT);
VAR
  I, J : INTEGER;

FUNCTION SQUARE(X : INTEGER) : INTEGER;
  BEGIN
    SQUARE := X * X
  END;

FUNCTION FCALLFUNC(FUNCTION Z(R : INTEGER) : INTEGER) : INTEGER;
  BEGIN
    FCALLFUNC := Z(5)  {Invokes specified function, with 5}
  END;                 { as actual parameter}

PROCEDURE PCALLFUNC(FUNCTION Y(Q : INTEGER) : INTEGER);
  BEGIN
    I := Y(5)          {Invokes specified function, with 5}
  END;                 { as actual parameter}

BEGIN {main program}
  J := FCALLFUNC(SQUARE); {Invokes FCALLFUNC, then PCALLFUNC,}
  PCALLFUNC(SQUARE);      { both with SQUARE as actual parameter}
  WRITELN(I, J)           {I = 25; J = 25}
END.
```

Example 4:

{This example also uses both a procedure and a function as parameters. The procedure and function in this program, however, do not operate identically, as they did in Example 3.}

```
PROGRAM Call4(OUTPUT);
VAR
  I, J : INTEGER;

PROCEDURE ADD(X : INTEGER);
  BEGIN
    I := I + X    {Add value of parameter to}
  END;            { that of global variable}

FUNCTION FCALLPROC(PROCEDURE Z(R : INTEGER)) : INTEGER;
  VAR
    R : REAL;
  BEGIN
    R := 2.19;
    Z(ROUND(R));      {Call specified procedure}
    FCALLPROC := 10   {Function value is 10}
  END;

PROCEDURE PCALLPROC(PROCEDURE Y(Q : INTEGER));
  BEGIN
    Y(8);     {Call specified procedure, with 8}
  END;        { as formal parameter}

BEGIN {main program}
  I := 0;
  J := FCALLPROC(ADD); {Invokes FCALLPROC with ADD}
  WRITELN(I, J);       {I = 2; J = 10}
  PCALLPROC(ADD);      {Invokes PCALLPROC with ADD}
  WRITELN(I, J)        {I and J both = 10}
END.
```

Example 5:

{The effect of this program is to call the function ADD1 three times.}

```
PROGRAM Call5(OUTPUT);
VAR
  I : INTEGER;

PROCEDURE ADD2(PROCEDURE A1);
  BEGIN
    A1;       {Invokes specified procedure twice}
    A1
  END;

PROCEDURE ADD1;
  BEGIN
    I := I + 1    {Increments global variable}
  END;

PROCEDURE CALLPROC(PROCEDURE X(PROCEDURE Y); PROCEDURE Z);
  BEGIN
    Z;      {Invokes PROCEDURE Z; here, ADD1}
    X(Z)    {Invokes PROCEDURE X, with PROCEDURE Z}
  END;      { as actual parameter; here, ADD2 has ADD1}
            { as actual parameter}

BEGIN {main program}
  I := 0;
  CALLPROC(ADD2, ADD1);
  WRITELN(I)              {I = 3}
END.
```

Note

A procedure or function cannot be a variable parameter. For example:

```
PROCEDURE X (VAR PROCEDURE Y);  {this is invalid}
```

Any attempt to use a procedure or function as a variable parameter causes the VAR to be ignored and a severity 2 error to be given.

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. (CLOSE is a Prime extension. See Chapter 10.)
- Dynamic Allocation Procedures: NEW and DISPOSE. (See Chapter 6.)

Note

19.2 Use of the standard transfer procedures PACK and UNPACK in Prime Pascal will generate a severity 3 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.

Function Declarations

The general form of a function declaration is:

```
FUNCTION identifier [(formal-parameter-list)]:
    result-type-identifier;
      block;
```

The identifier is the name of the function. The result-type-identifier is the data type of the function. The formal-parameter-list consists of parameters.

Example 1:

```
FUNCTION SQRT(X : REAL) : REAL;
{This function computes the square root of X (X>0) using Newton's
method.}
VAR
  OLD, NEW : REAL;
BEGIN
  NEW := X;
  REPEAT
    OLD := NEW;
    NEW := (OLD + X/OLD) * 0.5
  UNTIL ABS(NEW - OLD) < EPS * NEW; {EPS being a global constant}
  SQRT := NEW
END;  {Function Sqrt}
```

Example 2:

```
FUNCTION MAX(A : VECTOR; N : INDEXTYPE) : REAL;
{This function finds the largest value in A, which is declared
  A : ARRAY[INDEXTYPE] OF REAL and where
  INDEXTYPE = 1..LIMIT}
VAR
  LARGESTSOFAR : REAL;
  FENCE : INDEXTYPE;
BEGIN
  LARGESTSOFAR := A[1];
  {Establishes LARGESTSOFAR = MAX(A[1])}
  FOR FENCE := 2 TO N DO
      IF LARGESTSOFAR < A[I]  THEN
        LARGESTSOFAR := A[I];
        {Re-establishing LARGESTSOFAR = MAX(A[1],...,A[FENCE])}
  {So now LARGESTSOFAR = MAX(A[1],...,A[N])}
  MAX := LARGESTSOFAR
END; {Function MAX}
```

Invoking Functions

The appearance of the function name in a program invokes (or calls) and executes the function subprogram. This invocation is called the function designator. The function designator has the form:

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

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

```
VAR
  J, K : INTEGER;
FUNCTION CUBE(I : INTEGER) : INTEGER;
  BEGIN
    CUBE := I * SQR(I)
  END; {Function CUBE}
BEGIN {main program}
  READLN(J);
  K := CUBE(J);  {Function CUBE is invoked here.}
  .
  .
  .
END.
```

Standard Functions

A standard function, which has a predefined function name, is a built-in function supplied by the Pascal language. The available standard functions are listed and explained in Chapter 11.

FORWARD PROCEDURES AND FUNCTIONS

Pascal permits subprograms to call each other within the same Pascal program. Subprogram A may call subprogram B before B is fully defined if B has already been declared using the forward declaration.

Forward Declarations

A forward declaration is like other subprogram declarations, except that the subprogram block is replaced by the word FORWARD. This block, led by the keyword PROCEDURE or FUNCTION and its associated subprogram name, appears later in the program. Example:

```
FUNCTION GCD(N,M : INTEGER) : INTEGER; FORWARD;
PROCEDURE LOWTERM(VAR N,D : INTEGER);
VAR
  CD : INTEGER;
BEGIN
  CD := GCD(N,D);  {This statement needs the forward declaration.}
  N := N DIV CD;
  D := D DIV CD
END; {procedure LOWTERM}
FUNCTION GCD;  {Note the abbreviated heading}
{Full declaration of GCD begins here.}
VAR
  R : INTEGER;
BEGIN
  REPEAT
    R := M MOD N;
    IF R <> 0 THEN
      BEGIN
        M := N;
        N := R
      END
  UNTIL R = 0;
  GCD := N
END; {function GCD}
```

EXTERNAL PROCEDURES AND FUNCTIONS

Prime Pascal allows a program to call independent, external, separately compiled subprograms after they have been declared with the external declarations within the program. These subprograms can be external Pascal procedures and functions or subprograms written in other languages. This is a Prime extension.

External Declarations

To declare an external, separately compiled subprogram, simply use the word EXTERN at the end of a procedure or function heading, similar to the FORWARD declaration. For example:

```
PROCEDURE PLOT(X, Y : REAL; I : INTEGER); EXTERN;
```

The body of the external subprogram does not appear in the calling program. The external subprogram file will be located at load time.

Note

Use the word EXTERN in every external subprogram declaration, no matter what language the subprogram is written in.

The calling program calls the subprogram and passes parameters in the usual way:

```
PLOT(X, Y, 3);
```

The parameters that are passed must be compatible with the parameters of the subprogram in number and data type. (See Appendix D for information on interfacing Pascal data types with those of other languages.)

Subprograms Written in Pascal

When you write a subprogram in Pascal, you must tell the compiler that the subprogram is to be compiled externally and that the subprogram will be called by other programs.

Using the {$E+} Compiler Switch: To achieve these requirements, simply put Pascal's {$E+} compiler switch at the beginning of every external subprogram file. For example:

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

Do not use the {$E-} switch at the end of the file. Also, the subprogram ends with a semicolon, not a period. Without the {$E+} switch, the compiler would expect the main body of the program to be included; that is, it would expect BEGIN...END followed by a period.

Note

You can have many subprograms in the same file. If you have many subprograms in a file, put only one {$E+} switch at the top of the file. All of the subprograms within a file will compile when the file itself is compiled.

For more information on the {$E+} compiler switch see Chapter 2.

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;
```

Declaring External Procedures and Functions: If you want your external subprogram to call a procedure or function that is contained in the main program, you must use the {$E+} and {$E-} switches around the procedure or function declaration in the main program.

Here is an example of a main program that contains an externally declared procedure:

```
VAR
  A, B, C, D : INTEGER;
{$E+}
PROCEDURE ADD (X : INTEGER Y : INTEGER);
  VAR
    Z : INTEGER;
  BEGIN {add}
    Z := X + Y;
    WRITELN('Sum is ',Z)
  END;
{$E-}
PROCEDURE MULT (P : INTEGER; Q : INTEGER); EXTERN;
BEGIN {main}
  A := 8;
  B := 9;
  ADD(A, B);
  C := 5;
  D := 6;
  MULT(C, D)
END.
```

19.2

Here is the external subprogram, that calls the procedure:

```
{$E+}
PROCEDURE ADD(X : INTEGER; Y : INTEGER); EXTERN;
PROCEDURE MULT (I : INTEGER; J : INTEGER);
  VAR
    M : INTEGER;
    K, L : INTEGER;
  BEGIN {mult}
    K := 50;
    L := 60;
    M := I * J;
    WRITELN('Mult is ',M);
    ADD (K, L)    {external procedure called here}
  END;
```

Notice that the procedure is declared again under the {$E+} switch, and that this procedure heading ends with EXTERN.

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.

Subprograms from Libraries

Prime supplies several libraries of application-level subroutines and PRIMOS operating system subroutines. These subroutines can be declared as external procedures or functions and then called from any point within the program. When a subroutine from such a library is being used, the library must be loaded with SEG's LIBRARY command. (See Chapter 2 for instructions.)

For more information on Prime's subroutines, see the Subroutines Reference Guide.

RECURSIVE PROCEDURES AND FUNCTIONS

A subprogram can call itself. This process is called recursion. A subprogram can keep calling itself for as many times as necessary.

Recursive subprograms are said to be at different "levels." Whenever a subprogram calls itself, a new set of identical local variables is set up automatically and the values of these variables change back or "initialize". The computer remembers and stores the values at each "level", so that when the program recurses back from the innermost subprogram to the outermost, the operations at each level finish executing.

The following program is a simple example of recursion in Pascal. This program writes out a palindrome. A palindrome is a word that is spelled the same way forward and backward. Given just half of the palindrome in the input textfile, the program recurses and echoes the entire palindrome back to the terminal:

```
PROGRAM PALINDROME;
VAR
  X : CHAR;
PROCEDURE PUTPAL;
  VAR
    X : CHAR;
  BEGIN
   IF NOT(EOLN(INPUT)) THEN
     BEGIN
       READ(INPUT, X);
       WRITE(X);
       PUTPAL; {recursion happens here}
       WRITE(X);
     END;
  END; {of PUTPAL procedure}
BEGIN {main program}
  RESET(INPUT, 'indata');
  WHILE NOT (EOF(INPUT)) DO
    BEGIN
      PUTPAL;
      READLN(INPUT);
    END;
  WRITELN;
  CLOSE(INPUT)
END.
```

If the input file contains the characters NO, the palindrome program will recurse and print the word NOON at your terminal.

The following program is another good example of recursion:

```
PROGRAM Frog;

{Using a recursive procedure called JUMP, this program
calculates and writes out all of the ways a frog can jump to
the top of a 5-step flight of stairs, jumping one, two, or
three steps at a time. TOPSTEP is a constant that stands for
the fifth (top) step, and it flags the end of any of the frog's
series of jumps. WHICHSTEP is an array of type CURRENTSTEP
(0..5) that keeps track of the current step and which steps
were hit on the way up.}

CONST
  TOPSTEP = 5;
TYPE
  CURRENTSTEP = ARRAY[0..TOPSTEP] OF INTEGER;
VAR
  WHICHSTEP : CURRENTSTEP;
```

{The procedure is declared. N, which increments the WHICHSTEP position, is initially passed the value of 0 from the main program. LEAP is the index for the outer FOR loop and it controls the possible number of steps the frog can jump. I is the index for the inner FOR loop that controls the writing out of all the steps the frog hit on the way up.}

```
PROCEDURE JUMP(N : INTEGER);
  VAR
    LEAP : INTEGER;
    I : INTEGER;
  BEGIN {JUMP procedure}
```

{The outer FOR loop checks to see if the current step plus one more leap is greater than the TOPSTEP. If not, then the current step becomes the current step plus the leap.}

```
    FOR LEAP := 1 TO 3 DO
      BEGIN
        IF (WHICHSTEP[N] + LEAP < 6) THEN
          BEGIN
            WHICHSTEP[N + 1] := WHICHSTEP[N] + LEAP;
            IF WHICHSTEP[N + 1] = TOPSTEP THEN
              BEGIN
```

{The inner FOR loop writes out all of the steps the frog hit on the way up, if the frog has reached the TOPSTEP, with his allowed number of leaps.}

```
                FOR I := 1 TO (N + 1) DO
                  WRITE (WHICHSTEP[I]);
                WRITELN
              END
```

```
{The procedure calls itself — keeps jumping — if the frog
hasn't reached the TOPSTEP.}

            ELSE
              JUMP(N + 1)    {Recursion happens here}
          END
      END
  END; {of JUMP procedure}

{The main program writes a heading, initializes WHICHSTEP and N
to zero, then goes into the recursive routine.}

BEGIN {main program}
  WRITELN;
  WRITELN;
  WRITELN ('COMBINATION OF STEPS FROG CAN JUMP':43);
  WRITELN;
  WHICHSTEP[0] := 0;
  JUMP(0)
END.
```

When the program is executed, you will get the following output at your terminal:

```
COMBINATION OF STEPS FROG CAN JUMP

1       2       3       4       5
1       2       3       5
1       2       4       5
1       2       5
1       3       4       5
1       3       5
1       4       5
2       3       4       5
2       3       5
2       4       5
2       5
3       4       5
3       5
```

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:

```
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.)

> Caution
>
> When the -INTERACTIVE switch is on, you cannot use CONTROL-C as an end-of-file marker on terminal input. If you need an end-of-file in your program, remove the -INTERACTIVE switch or turn it off with the -TTY switch, explained below, before using CONTROL-C.

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:

```
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.

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.
- 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.

The RESET Procedure

The RESET procedure statement, which opens an input file, must be placed in the executable part of the program before data from the file is used. The format of RESET is:

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

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

Consider the following example:

```
PROGRAM Readfile;
VAR
  A, B, C : INTEGER;
  INFILE : FILE OF CHAR;
BEGIN
  RESET(INFILE, 'INDATA');
  READ(INFILE, A);
  READ(INFILE, B);
  C := A + B;
  WRITELN(C);
  CLOSE(INFILE)
END.
```

The name of the input file (the second parameter) can be either a simple filename, as shown above, or a pathname. For example:

```
RESET(INFILE, 'PAUL>HOMEWORK>INDATA');
```

The file INDATA resides in the subdirectory HOMEWORK within the directory PAUL. The pathname also must be in single quotes. (For more information on UFDs and sub-UFDs, see the Prime User's Guide.)

In the sample program above, notice that the file variable INFILE must be declared as a file. The CLOSE procedure must be used to close a data file. (CLOSE is discussed later in this chapter.)

You can also use a variable to represent a filename, and use that variable in the RESET procedure to open a file. For example:

```
VAR
  A : ARRAY[1..32] OF CHAR;
        .
        .
        .
  RESET(INFILE, A);
        .
        .
        .
```

Using a variable is particularly useful when you have to read data from different input files. For example, consider this program:

```
PROGRAM Pickafile;
VAR
  F : FILE OF CHAR;
  I : INTEGER;
  FILENAME : ARRAY[1..128] OF CHAR;
BEGIN
  WRITE('Please type in the name of file to be processed:');
  READLN(FILENAME);
  RESET(F, FILENAME);
  WHILE NOT EOF(F) DO
    BEGIN
      READLN(F, I);
      WRITELN(I)
    END;
  CLOSE(F)
END.
```

When you execute this program, it will ask you for the name of the data file, open that file, and perform its operations using the data in that file.

Note

RESET does an implicit GET.

If you create an input file using one of Prime's text editors, you should declare the file variable as FILE OF CHAR, regardless of the type of data values you're using — INTEGER, REAL, BOOLEAN, and so on. Because the editor interprets all data as ASCII characters, you would not be able to read the data in your input file if the file variables were declared FILE OF INTEGER, FILE OF REAL, or anything other than FILE OF CHAR.

A file that has been declared FILE OF CHAR is commonly called a textfile.

If you declare a file to be FILE OF INTEGER, FILE OF REAL, etc., the input file should be created by the Pascal compiler, not the text editor. You can accomplish this by making a program or subprogram generate the input file. Suppose you wanted to place five integers into an input file that is created by Pascal. You could read five integers from the terminal, write the integers out to an output file, and then make the outfile the input file. For example:

```
VAR
  A, B, C, D, E : INTEGER;
  DATAFILE : FILE OF INTEGER;
BEGIN
  READ(A, B, C, D, E);
  REWRITE(DATAFILE, 'DATA');
  WRITE(DATAFILE, A, B, C, D, E);
  RESET(DATAFILE);
  READ(DATAFILE, A, B, C, D, E);
  CLOSE(DATAFILE)
END.
```

The file variable is declared FILE OF INTEGER. Five integers are read from the terminal upon execution. An ouptut file named DATA is opened with the REWRITE procedure. The five integers are written out to that file. The same file is reopened as an input file with RESET. The five integers are read again — this time from the new input file, which is still named DATA. (The REWRITE procedure is discussed later in this chapter.)

Note

When a nontextfile is created using Pascal -- FILE OF INTEGER, FILE OF REAL, etc. — and not the text editor, you cannot modify the contents of that file with the editor because the data are stored in binary form and not as ASCII characters. The data in your input file created by Pascal would be unrecognizable to you. Your data, therefore, would have to be modified by a Pascal program or subprogram. Also, the standard procedures READLN and WRITELN can only be used on files of type FILE OF CHAR. The other standard procedures (READ, WRITE, GET, and PUT) can be used on files of any other type.

You can declare file types using structured types such as RECORD, ARRAY, and SET. For example:

```
TYPE
  IOREC = RECORD
            A: INTEGER;
            B: ARRAY [1..6] OF CHAR;
            C: (LEFT, RIGHT)
          END;
VAR
  F: FILE OF IOREC;
BEGIN
  RESET(F, 'F1');
    .
    .
    .
```

Remember that files not declared as FILE OF CHAR, such as the one above, cannot be modified with a text editor. They must be modified with a Pascal program or subprogram only.

Using the TEXT File Type

Standard Pascal has a standard FILE type called TEXT. It is identical to FILE OF CHAR. Whenever you are using CHAR type data in an input file — or whenever you declare any file as FILE OF CHAR to use Prime's text editors -- you can simply declare it as TEXT instead. For example, the following declarations are identical:

```
VAR
  F : FILE OF CHAR;

VAR
  F : TEXT;
```

For more information on TEXT, see Chapter 6.

Using the Standard Textfile INPUT

Standard Pascal also has a standard textfile called INPUT. This textfile does not have to be declared as a FILE type. For example:

```
VAR
  A : INTEGER;
BEGIN
  RESET(INPUT, 'INDATA');
  READLN(INPUT, A);
```

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;
BEGIN
  RESET(INPUT, 'INDATA');
  READLN(INPUT, A);
  RESET(INPUT, '-TTY');
  READ(B)
END.
```

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(OUTFILE, 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.

A variable can also represent an output filename, and you can use that variable in a REWRITE procedure to open a file:

```
VAR
  OUTFILE : TEXT;
  A : ARRAY[1..11] OF CHAR;
BEGIN
  A := 'PAUL>SAMPLE';
  REWRITE(OUTFILE, A)     {This is equivalent to}
                          {REWRITE(OUTFILE,'PAUL>SAMPLE')}
END.
```

Output files, like input files, can contain data of a structured type. For example:

```
TYPE
  A = ARRAY[1..10] OF CHAR;
VAR
  OUTDATA : FILE OF A;
```

Using the Standard Textfile OUTPUT

Pascal has a standard textfile called OUTPUT. This is similar to the standard textfile INPUT, which was explained earlier in this chapter. OUTPUT does not have to be declared as a file. For example:

```
VAR
  A : ARRAY[1..60] OF CHAR;
BEGIN
  REWRITE(OUTPUT, 'OUTDATA');
  WRITELN(OUTPUT, A);
  CLOSE(OUTPUT)
END.
```

When OUTPUT is used with an output file, the name of the file must still be given as the second parameter in the REWRITE procedure, as shown above.

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

```
WRITE(OUTPUT, A);

WRITE(A);
```

For more information on OUTPUT, see Chapter 6.

Switching from Standard OUTPUT File to Terminal

If you open an output data file with the standard textfile OUTPUT, and want to switch to outputting data to the terminal, use the -TTY switch in another REWRITE procedure. For example:

```
VAR
  A, B : INTEGER;
BEGIN
  REWRITE(OUTPUT, 'OUTDATA');
  WRITELN(OUTPUT, A);
  REWRITE(OUTPUT, '-TTY');
  WRITELN(B)
END.
```

18.3

The value of A will be written to an output data file named OUTDATA, and the value of B will be written to the terminal. The standard file OUTPUT is the first parameter with -TTY. The -TTY switch must be enclosed in single quotes.

The -TTY switch also works for RESET and the standard textfile INPUT.

I/O PROCEDURES AND FUNCTIONS

In addition to RESET and REWRITE, which are discussed earlier in this chapter, there are 10 other built-in I/O procedures and functions.

There are three input file-handling procedures:

- GET
- READ
- READLN

There are three output file-handling procedures:

- PUT
- WRITE
- WRITELN

There are two Boolean functions:

- EOF
- EOLN

There are two auxiliary procedures:

- PAGE
- CLOSE (a Prime extension)

All of these procedures and functions are briefly discussed in the paragraphs that follow. For more information on these procedures and functions (except CLOSE), consult a commercially available text. Other standard Pascal functions are summarized in Chapter 11.

Input File-handling Procedures

In addition to the RESET procedure, there are three input file-handling procedures — GET, READ, and READLN. (The RESET procedure is discussed earlier in this chapter.)

The GET Procedure: The GET procedure can be used to move the file pointer to the next element of the file.

The form of GET is:

```
GET(file);
```

GET advances the current file position to the next element, assigns the value of this element to the buffer variable file^, and leaves EOF(file) false. If no next element exists, EOF(file) becomes true, EOLN(file) becomes false, and file^ is a space. The parameter file is of a FILE type.

The READ Procedure: The READ procedure reads input data values and assigns these values, in order by position, to the variables in the READ parameter list. It has the form:

```
READ ([file,] variable-1 [,variable-2]...);
```

The variables can be of type CHAR, INTEGER, LONGINTEGER, REAL, LONGREAL, BOOLEAN, or a subrange of type CHAR, INTEGER, or LONGINTEGER, etc. The file is a file variable that is associated with an input data file. For example: 19.1

```
READ(INDATA, A, B);
```

Without a data input file, data is read from the standard textfile INPUT by default, whether INPUT is a file or the terminal:

```
READ(A, B);
```

When the variable is of type CHAR (or a subrange thereof), the call:

```
READ(file, variable);
```

is equivalent to:

```
variable := file^;
GET(file);
```

19.1 When the variable is of type INTEGER, LONGINTEGER, REAL, LONGREAL, or a subrange of type INTEGER or LONGINTEGER, the call:

```
READ(file, variable);
```

reads a sequence of characters that forms a number according to the rules for numeric constants (Chapter 4) and assigns the number to a variable. Successive numbers are separated by blanks, ends of lines, or commas.

A READ procedure may have several parameters. The call:

```
READ (file, variable-1,...,variable-n);
```

is equivalent to:

```
READ (file, variable-1);
        .
        .
        .
READ (file, variable-n);
```

In Prime Pascal, a READ automatically moves the pointer to the next character. For example, consider these statements:

```
READ(INFILE, CH);
IF EOLN(INFILE) THEN
     .
     .
     .
```

The READ procedure assigns a value to CH, then moves the file pointer to the next character on the line. The EOLN function then tests to see if the character that the pointer is now pointing to is a carriage return (end of line). Prime Pascal differs from many other languages because in those languages a READ assigns a value while the pointer remains at the character that was just assigned.

When you are inputting data from the terminal, you must still enter a carriage return so that an EOLN can be encountered. (The EOLN function is further discussed later in this chapter.)

The READLN Procedure: The READLN procedure is a special form of the READ procedure. It has the form:

```
READLN[([file|variable-1][,variable-2]...)];
```

READLN must only be applied to files that have been declared FILE OF CHAR or TEXT, or to data input at the terminal. If the first parameter file is omitted, then data is read from the standard textfile INPUT by default, whether INPUT is a file or the terminal.

A READLN statement by itself skips over characters until the end of the current line and places the current file position at the beginning of the next line. Thus, the call:

```
READLN(file);
```

is equivalent to:

```
WHILE NOT EOLN(file) DO
  GET(file);
GET(file);
```

With variables, READLN reads the input data values into the variables and then skips to the beginning of the next line. The call:

```
READLN(file, variable-1,...,variable-n);
```

is equivalent to:

```
READ(file, variable-1,...,variable-n);
READLN(file);
```

Output File-handling Procedures

In addition to the REWRITE procedure, there are three output file-handling procedures — PUT, WRITE, and WRITELN. (The REWRITE procedure is discussed earlier in this chapter.)

The PUT Procedure: After the REWRITE procedure opens an output file, the PUT procedure can write the output value into the file.

The form of PUT is:

```
PUT(file);
```

PUT appends and writes the value of the buffer variable file^ to the end of the file. EOF(file) remains true. PUT writes the value into the file, but does not assign the value to the variable, as WRITE does. Here is another example using PUT:

```
file^ := CH;
PUT(file);
```

The WRITE Procedure: The WRITE procedure writes the values of the text and/or expressions into the output textfile. It has the general form:

```
WRITE([file,] write-parameter-1 [,write-parameter-2]...);
```

The file is an output data file. For example:

```
WRITE(OUTFILE, X, Y, Z);
```

If file is omitted, data is written out to the standard textfile OUTPUT by default, whether OUTPUT is a data file or the terminal:

```
WRITE(X, Y, Z);
```

If the WRITE procedure has several parameters, the call:

```
WRITE(file, write-parameter-1,...,write-parameter-n);
```

is equivalent to:

```
BEGIN
  WRITE (file, write-parameter-1);
     .
     .
     .
  WRITE (file, write-parameter-n)
END.
```

A write-parameter is either a character string enclosed by a pair of apostrophes or an expression (represented by its variable name) with optional field-width parameters.

If the write-parameter is a character string, it is written on the output file exactly as it appears, without the delimiting apostrophe characters. For example:

```
WRITE('The ''blank character'' is significant in a string.');
```

will produce:

```
The 'blank character' is significant in a string.
```

The use of two consecutive quotes will produce a single quote in output.

If the write-parameter is a numeric expression, the value of the expression is written in the output file.

You can specify the field width of the expression's value with this format:

```
expression [ : total-width [: frac-digits]]
```

The expression may be of type INTEGER, LONGINTEGER, REAL, LONGREAL, BOOLEAN, or of a structured type. The total-width is the total number of character positions you want allocated to the value of the expression. The frac-digits value, which can only be applied to type REAL or LONGREAL, is the number of digits you want printed to the right of the decimal point. 19.1 19.1

If the total width is larger than the actual value, then the difference in the number of digits will be written as blanks to the left of the value. For example, if A=10, and you say WRITELN(A:6); four blanks will be written out to the left of the 10.

If the total width is omitted, a default field-width will be assumed according to the type of the expression. The default field-widths are as follows:

Data Type	Number of Character Positions
INTEGER, LONGINTEGER	10
REAL, LONGREAL	21
CHAR	1
BOOLEAN	4 (TRUE) 5 (FALSE)

A REAL value for which no frac-digits are specified will be written in floating-point (scientific) form:

```
d...d.dddddddE+|-dd
      ~~~~~~~~
      6-digit
```

The LONGREAL form looks like this:

```
d...d.ddddddddddddddD+|-dddd
      ~~~~~~~~~~~~~~~~
        12 digit
```

where each d denotes a decimal digit. The letter E is used for REAL exponents, and the capital letter D is used for LONGREAL exponents. If the default field-width is used, there should be 11 digits preceding the decimal point for reals. Otherwise, this number of digits should depend on the total-width specified. For example, if R and S are REAL variables with values 0.1 and 1.5 respectively, the WRITELN procedures:

```
WRITELN(R);

WRITELN(S:14);
```

will produce the following:

```
bbbbbbbbbb1.000000E-01
{Default case}

bb1.500000E+00
```

where each b is a blank.

frac-digits invokes fixed-point (or decimal) representation for a REAL or LONGREAL value and specifies the number of digits following the decimal point. For example, if R and S are REAL variables with values 1.5 and 112.123 respectively, the WRITE procedure: | 19.1

```
WRITE(R:7:2, S:10:1);
```

will produce the following:

```
bbb1.50bbbbbb112.1
```

where each b is a blank.

If the true field-width of the value is larger than the field-width specified, Pascal will automatically extend the specified total-width to a sufficient size. For example, if Y is a BOOLEAN variable with value FALSE and R is a REAL variable with value 112.12, the WRITE procedure:

```
WRITE(Y:1, R:3:2);
```

will produce the following:

```
FALSEb112.12
```

where b is a blank. A positive REAL or LONGREAL number is always written with one preceding blank, even if the specified field-width is smaller than the true field-width. This, however, does not apply to values of other types. For example, if R, S, and T are REAL variables with values 1.5, +1.5, and -1.5 respectively, Y is a BOOLEAN variable with value TRUE, and I is an INTEGER variable with value 6, the WRITE procedure: | 19.1

```
WRITE(R:7:5, S:7:5, T:7:4, Y:4, I:1);
```

will produce the following:

```
b1.50000b1.50000-1.5000TRUE6
```

where b is a blank.

If the expression is of type CHAR, the statement:

```
WRITE(file, expression);
```

is equivalent to:

```
file^ := expression;
PUT(file);
```

The WRITELN Procedure: The WRITELN procedure is a form of the WRITE procedure. It has the format:

```
WRITELN[([file|write-parameter-1][,write-parameter-2]...)];
```

For the description of write parameters, see the preceding section.

WRITELN must only be applied to files that have been declared FILE OF CHAR or TEXT. If the first parameter file is omitted, then data is written out to the standard textfile OUTPUT by default, whether OUTPUT is a data file or the terminal.

WRITELN writes a carriage return to the file. Thus, the call:

```
WRITELN(file,write-parameter-1,...,write-parameter-n);
```

is equivalent to:

```
WRITE(file,write-parameter-1,...,write-parameter-n);
WRITELN(file);
```

If a WRITELN procedure is called with a single parameter file or no parameter at all, WRITELN simply sends a carriage return to the output file.

BOOLEAN Functions

Pascal has two built-in BOOLEAN functions that manipulate I/O -- EOF (end of file) and EOLN (end of line).

The EOF Function: The EOF function tests for an end-of-file condition of a file. The form of EOF is:

```
EOF(file)
```

This function is true if the buffer variable file^ has moved beyond the end of the file. Otherwise it is false. If file is omitted, EOF is applied to the standard textfile INPUT, by default, whether INPUT is a data file or the terminal.

Note

The CONTROL-C character is the end-of-file marker for input that is read from the terminal. When you want your program to check for an end-of-file marker on terminal input, type CONTROL-C at the terminal. If CONTROL-C is not typed, the EOF condition will not be checked.

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 five types of standard functions -- arithmetic, transfer, ordinal, BOOLEAN, and STRING. | 19.2

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.

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

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.

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

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
```

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 error is generated. This also applies to REAL and LONGREAL. (See Chapter 6 for more information on LONGINTEGER and LONGREAL.)

ORDINAL FUNCTIONS

ORD(X) Gives the corresponding ordinal value of any character in Prime's character set. (See Appendix C.) X can be of any scalar type — except REAL and LONGREAL — including subrange and enumerated types. The result is an INTEGER value. For example: 19.1

```
ORD('F') yields 198
```

where 198 is the corresponding ordinal value of the character 'F' in the character set. Given an enumerated type:

```
VAR
  OPERATOR : (PLUS, MINUS, TIMES);
```

the ORD function can be used this way:

```
ORD(plus) yields 0

ORD(minus) yields 1

ORD(times) yields 2
```

The ORD function is also discussed in Chapter 6.

CHR(X) Gives the corresponding character value of any integer between 0 and 255 inclusive. CHR is the opposite of ORD. It yields a character element of Prime's character set. (See Appendix C.) X must be of type INTEGER. For example:

```
CHR(199) yields 'G'

CHR(225) yields 'a'

CHR(ORD('A')) yields 'A'

ORD(CHR(193)) yields 193
```

SUCC(X) Gives a value whose ordinal number is one greater than X. X can be of any scalar type — except REAL or LONGREAL — including any subrange or enumerated type. X and the result must be of the same type. For example: 19.1

```
SUCC(1) yields 2

SUCC('A') yields 'B'

SUCC(ORD('A')) yields 194
```

Given the following enumerated type:

```
TYPE
  COLORS = (RED, YELLOW, BLUE, GREEN);
```

the SUCC function can be used this way:

SUCC(YELLOW) yields BLUE

SUCC(ORD(BLUE)) yields 3

The SUCC function is also discussed in Chapter 6.

PRED(X) Gives a value whose ordinal number is one less than X. That is the only difference between PRED and SUCC. X can be of any scalar type — except REAL or LONGREAL — including any subrange or enumerated type. X and the result must be of the same type. For example:

19.1

PRED(2) yields 1

PRED('B') yields 'A'

PRED(ORD('B')) yields 193

Given the following enumerated type:

```
TYPE
  WEEKDAYS (SUN, MON, TUES, WED, THURS, FRI, SAT);
```

the PRED function can be used this way:

PRED(TUES) yields MON

PRED (ORD(FRI)) yields 4

The PRED function is also discussed in Chapter 6.

BOOLEAN FUNCTIONS

ODD(X)	X must be of type INTEGER or LONGINTEGER. The result is TRUE if X is odd and FALSE otherwise.
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.

STRING FUNCTIONS

There are nine functions that manipulate character strings. All of these functions, except the STR function, operate on values of the STRING data type. The STRING data type and all STRING functions are Prime extensions.

A brief description of each STRING function follows. For a complete explanation on all of these functions and the STRING data type, including program examples, see Chapter 6.

STR	Converts an ARRAY OF CHAR value to a STRING value.
UNSTR	Converts a STRING value to an ARRAY OF CHAR value.
LENGTH	Gives the operational length of the string. 19.2
INDEX	Determines if a first string contains a second string, and gives an integer that indicates the position in the first string where the second string begins.
SUBSTR	Gives a desired substring of any existing string.
DELETE	Deletes a specified substring of any existing string, and gives the resultant string.
TRIM	Removes all trailing blanks from a given string, and gives the resultant string.
LTRIM	Removes all leading blanks from a given string, and gives the resultant string.

APPENDIXES

A

Summary of Prime Extensions and Restrictions

This appendix lists Prime extensions and restrictions to standard Pascal. The extension or restriction and the chapter in which it is discussed appears in the left-hand column. A brief description appears in the right-hand column.

Prime Extensions

Extension and Chapter Reference	Description
The LONGINTEGER data type (Chapter 6 and Appendix B)	The LONGINTEGER type allows you to use 32-bit whole numbers without declaring a subrange. Variables declared as LONGINTEGER can have values that fall within the subrange -2147483648..+2147483647. (An INTEGER value is a 16-bit number that falls within the subrange -32768..+32767.) 19.1
The LONGREAL data type (Chapter 6 and Appendix B)	The LONGREAL type allows you to use 64-bit real numbers, as opposed to REAL values, which are 32-bit numbers.

19.1	The ARRAY OF CHAR enhancement (Chapter 6)	You can read an array of characters as one unit, instead of reading one character at a time.
18.3	Comment Delimiters /* */ (Chapter 4)	The sequence of symbols /* and */ can be used as comment delimiters in Pascal programs in addition to the standard delimiters { } and (* *).
19.1	The -INTERACTIVE switch for erase and kill characters (Chapter 10)	You can use Prime's "erase" and "kill" characters on data that is input at the terminal by using the -INTERACTIVE switch in a RESET procedure. With -INTERACTIVE, any character on the current line can be deleted before you type a carriage return.
18.3	The -TTY switch (Chapter 10)	Whenever you open and use input or output files with the standard textfiles INPUT and OUTPUT, and want to switch back to inputting or outputting data at the terminal, use the -TTY switch in another RESET procedure in your program. The -TTY switch can also turn the -INTERACTIVE switch off. That is, if you are using Prime's erase and kill characters with -INTERACTIVE, and want to go back to inputting data from the terminal without the use of erase and kill characters, use -TTY in another RESET procedure.
	Optional Program Heading (Chapter 5)	The program heading is optional in Prime Pascal. If present, the program heading is only checked syntactically by the Prime Pascal compiler.
	Order of Declarations (Chapter 5)	The LABEL, CONST, TYPE, VAR, PROCEDURE, and FUNCTION declarations can appear in any order in Prime Pascal. However, the standard order of declarations is strongly recommended.
	%INCLUDE files (Chapter 5)	The contents of a file can be included in a program unit at compile time with the %INCLUDE directive. %INCLUDE files can hold any legal Prime Pascal code — declarations as well as executable statements.

$ 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.

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.
The STRING data type (Chapter 6 and Appendix B)	The STRING data type makes it easy to manipulate character strings in Prime Pascal. It is similar to the CHARACTER VARYING type in PL/I Subset G. Unlike an array of characters, which must contain a precise number of character elements, STRING allows you to assign, compare, concatenate, read, write, and pass character strings that have a varying number of elements.
The STRING concatenation operator (+) (Chapters 6 and 7)	The concatenation operator (+) concatenates two strings into one. This operator works only on operands of the STRING data type.

19.2

Built-in STRING functions (Chapters 6 and 11)	There are nine built-in functions that manipulate character strings. All of these functions, except the STR function, operate on values of the STRING data type. The STR function converts an array of characters to a string. The UNSTR function converts a string to an array of characters. Other functions that manipulate strings are LENGTH, INDEX, SUBSTR, DELETE, INSERT, TRIM, and LTRIM.	19.2
The null string (Chapter 6)	The null string, which can only be assigned to character strings of the STRING data type, is a string that contains no characters. It is specified by two consecutive single quotes ''. Although the null string can be assigned to an existing string, it cannot be written out.	

Prime Restrictions

Restriction and Chapter Reference	Description
The keyword PACKED (Chapter 6)	The keyword PACKED is not supported in Prime Pascal. If PACKED is used, a severity 1 error will be generated at compile time.
The PACK and UNPACK procedures (Chapter 9)	The standard PACK and UNPACK procedures are not supported in Prime Pascal. An attempt to use PACK or UNPACK will generate severity 3 error and cause your program to fail.
Identifier length (Chapter 4)	Identifiers are limited to 32 significant characters. An identifier with more than 32 characters will generate a severity 1 error.
FILE OF CHAR (Chapter 10)	The standard procedures READLN and WRITELN can only be used on data contained in files of type FILE OF CHAR. The other standard procedures, READ, WRITE, GET, and PUT, can be used to do all I/O on files of any other type.

19.2

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)
Subrange
REAL
LONGREAL (Prime extension)
CHAR
BOOLEAN
Enumerated

Structured Data Types

ARRAY
RECORD
SET
FILE
STRING (Prime extension)

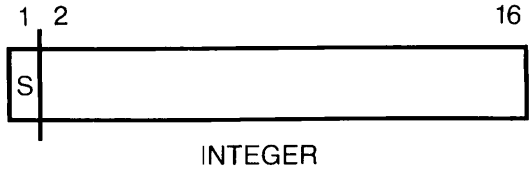
| 19.2

Pointer Data Type

Pointer

INTEGER TYPE DATA

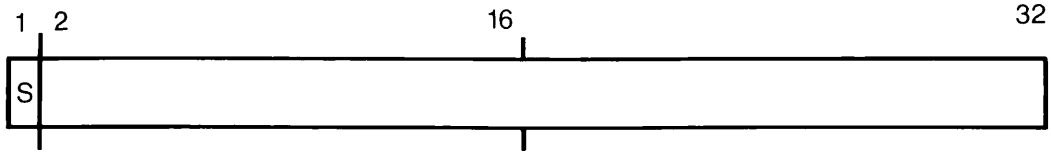
Integers are 16-bit (one word) twos-complement, fixed-point whole binary numbers. Integers can hold values within the range -32768 to +32767. Bit 1 is the sign bit, which indicates whether the integer is positive or negative. Bits 2-16 are the integer itself.



INTEGER

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.



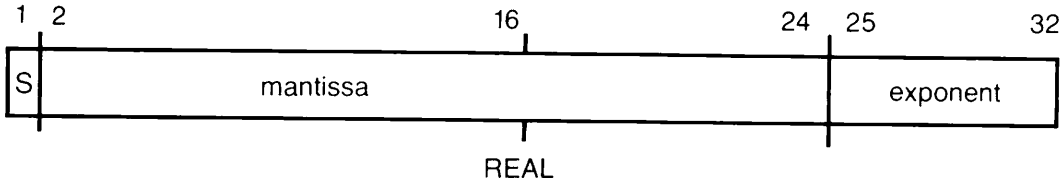
LONGINTEGER

SUBRANGE TYPE DATA

An INTEGER or LONGINTEGER subrange constant can either be 16-bit (one-word) or 32-bit (two-word) twos-complement, fixed-point whole binary number respectively. INTEGER subrange constants hold values in the range -32767..+32768. LONGINTEGER subrange constants can hold values within the range -2147483648..+2147483647. The first bit for every constant is the sign bit. These representations are the same as INTEGER and LONGINTEGER. Subrange values of types BOOLEAN and CHAR are represented the same way as BOOLEAN and CHAR values are. See the CHAR and BOOLEAN representations later in this appendix. For more information on subrange types for BOOLEAN, CHAR, and enumerated type data, see Chapter 6. 19.1 19.1 19.1

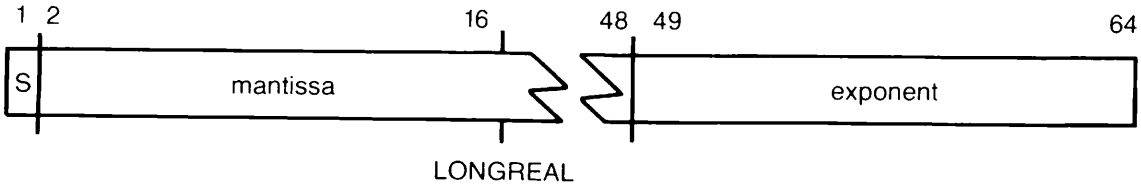
REAL TYPE DATA

REAL numbers are 32-bit (two-word) numbers. Bit 1 is the sign bit, which indicates whether the number is positive or negative. Bits 2-24 comprise the mantissa (fraction) in floating point (scientific) representation, and bits 25-32 comprise the exponent.



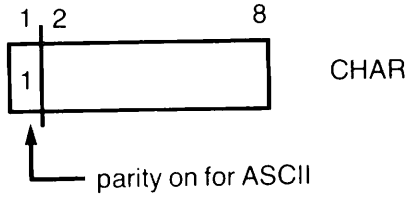
LONGREAL TYPE DATA

LONGREAL numbers are 64-bit (four-word) numbers. Bit 1 is the sign bit, which indicates whether the number is positive or negative. Bits 2-48 comprise the mantissa (fraction) in floating-point (scientific) representation, and bits 49-64 comprise the exponent. 19.1



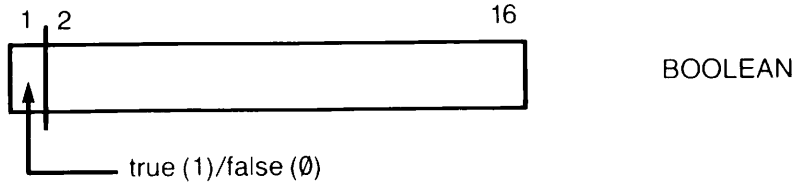
CHAR TYPE DATA

Each character is represented by one byte (eight bits). When an ASCII character value is represented, the parity bit is always on.



BOOLEAN TYPE DATA

A BOOLEAN value is stored in 16 bits (one-word). If Bit 1 is on (1) then the value is TRUE. If Bit 1 is off (0) then the value is FALSE. Bits 2-16 are ignored.



ENUMERATED TYPE DATA

Values of an enumerated type are stored as the ordinal numbers of those values. Each ordinal number of the enumerated type is a 16-bit (one-word) twos-complement, fixed-point whole binary number. For the internal representation of an ordinal number, see the INTEGER representation in this appendix. (Ordinal numbers of an enumerated type begin at 0 and increment positively.) The first element's ordinal number is 0, the second element's number is 1, etc.

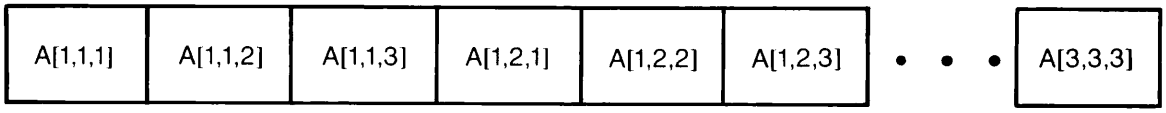
ARRAY TYPE DATA

The storage capacity and the internal representation of the ARRAY type data are determined by the index type (any scalar type except REAL) and the base type (any type) specified for the elements of the array.

A multidimensional array is represented internally by one row following another in straight linear fashion. This representation is commonly called "major order". For example, given the three-dimensional array:

```
A : ARRAY[1..3,1..3,1..3] OF INTEGER;
```

the internal representation would look like this:



MULTIDIMENSIONAL ARRAY

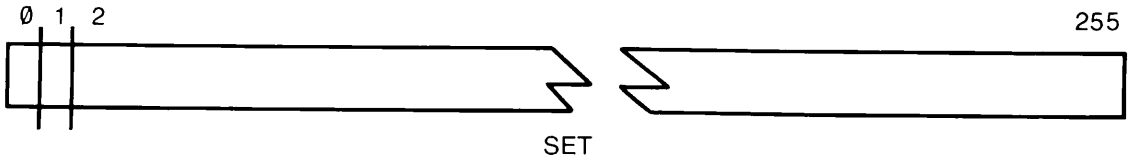
Each element is represented internally according to the index type and the base type specified for that element.

RECORD TYPE DATA

Storage of the RECORD type elements is allocated contiguously beginning with the first element (fields). Any non-CHAR type element of a record is stored in words while CHAR type elements are stored in bytes.

SET TYPE DATA

Each element of a set is stored in one bit. The ordinal values of SET elements range from 0 to 255. Therefore, the maximum number of bits used to represent a SET type is 256.

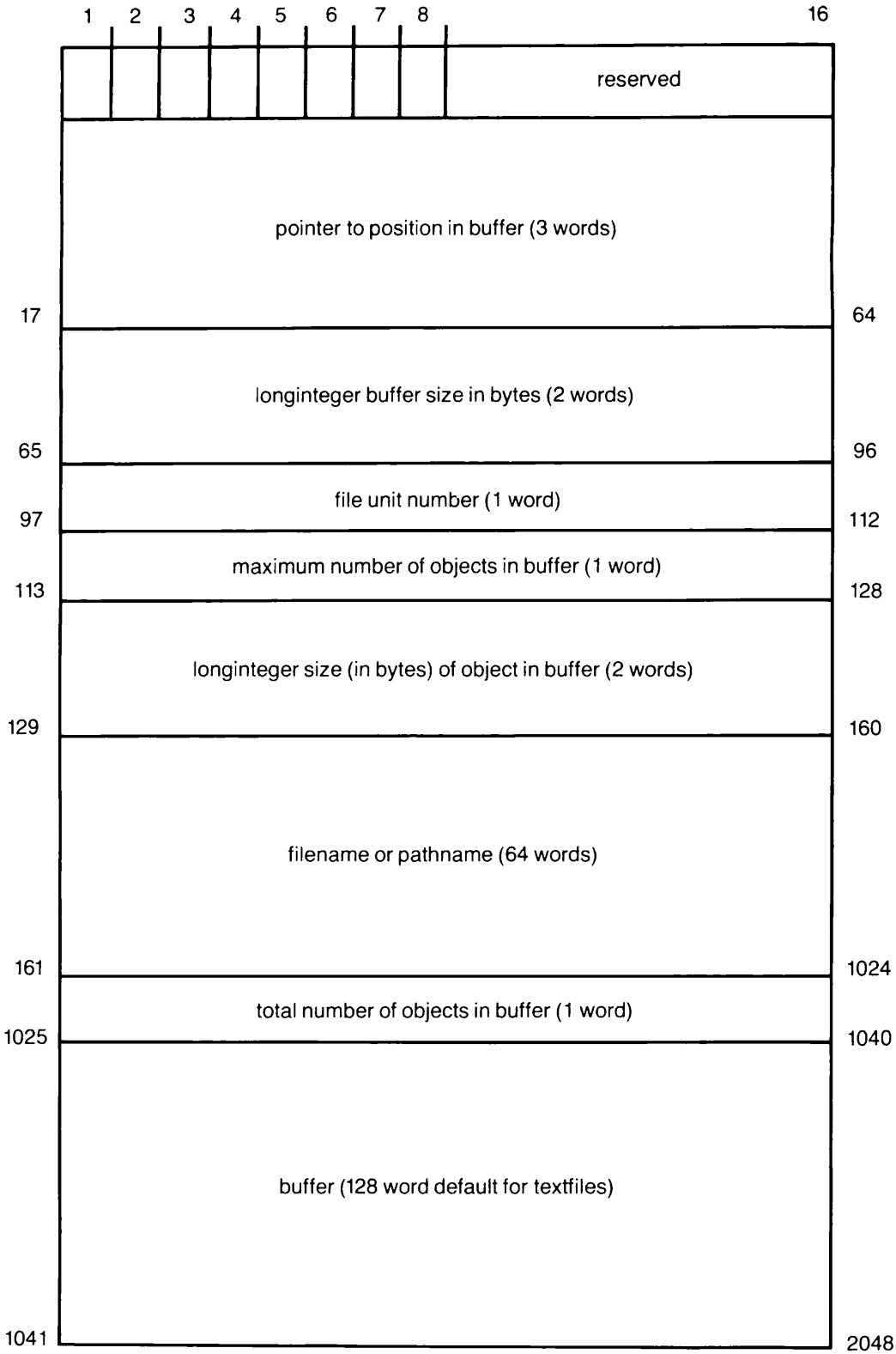


FILE TYPE DATA

FILE type data is stored and represented in a file control block. Within the file control block, a buffer stores data items from a file as they are input or output.

Specifically, the file control block consists of the following:

- Bit 1 indicates whether the data is input (1) or output (0).
- Bit 2 indicates whether the file is binary (0) or text (1).
- Bit 3 indicates the existence of end-of-line (EOLN). False is 0 and true is 1.
- Bit 4 indicates the existence of end-of-file (EOF). False is 0 and true is 1.
- Bit 5 indicates whether data is valid (1) or invalid (0).
- Bit 6 indicates whether the file control block is active (1) or inactive (0).
- Bit 7 indicates whether I/O is at the terminal (1) or a PRIMOS file (0).
- Bit 8 indicates whether the -INTERACTIVE switch is being used to allow erase and kill characters on terminal input (1) or not being used (0).
- Bits 9-16 are reserved for future use.
- Bits 17-64 are the pointer to the position in the buffer.
- Bits 65-96 contain the size of the buffer.
- Bits 97-112 are the file unit number.
- Bits 113-128 contain maximum possible number of objects in the buffer.
- Bits 129-160 contain the size of the object in the buffer.
- Bits 161-1024 contain the filename or pathname.
- Bits 1025-1040 contain the total number of objects in the buffer.
- Bits 1041 and on are the buffer itself. The size of the buffer varies according to the type of object in the buffer. The default size is 128 words (2048 bits) for textfile objects. This size can be much larger or smaller, depending on the type of object.



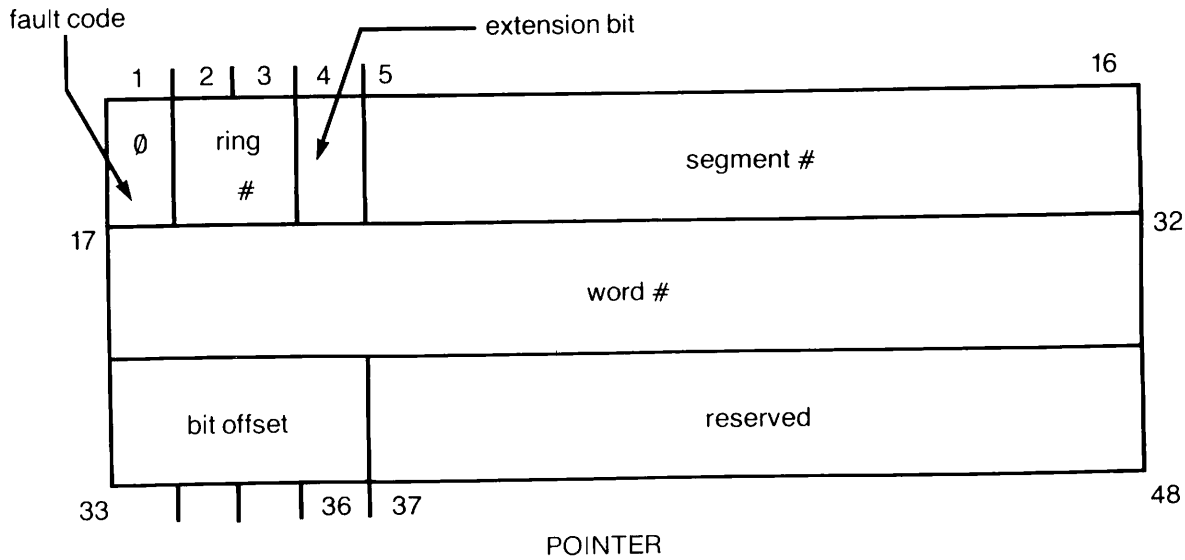
FILE CONTROL BLOCK

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.

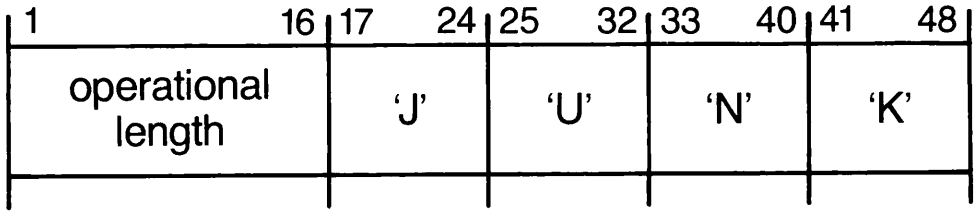


POINTER

STRING TYPE DATA

The first 16 bits in a string representation are the integer value for the operational length of the string. This integer is followed by the string itself. Each character in the string occupies 8 bits (one byte). The example below shows a string containing the value 'JUNK'. The operational length is 4. (The STRING data type is a Prime extension.)

19.2



STRING

C

ASCII Character Set

The standard character set used by Prime is the ANSI, ASCII 7-bit set with the 8 parity bit always on. Tables C-1 and C-2 present the ASCII nonprinting and printing character sets respectively.

PRIME USAGE

Prime hardware and software uses standard ASCII for communications with devices. The following points are particularly important to Prime usage:

- Output parity is normally transmitted as zero (space) unless the device requires otherwise, in which case software will compute transmitted parity. Some controllers (MLC) may have hardware to assist in parity generations.
- Input parity is always represented as a 1 by hardware and by standard software. Input drivers are responsible for making the parity bit suit the host software requirements. Some controllers (MLC) may assist in parity error detection.
- The Prime internal standard for the parity bit is one, that is, '200 is added to the octal value.

Notes to Table C-1

(1) Generally, CR is interpreted as .NL. at the terminal. In Pascal, however, CR (or LF) always returns from textfiles as a blank.

(2) .BREAK. at terminal. Relative copy in file; next byte specifies number of bytes to copy from corresponding position of preceeding line.

(3) Next byte specifies number of spaces to insert.

(4) Next byte specifies number of lines to insert.

C

ASCII Character Set

The standard character set used by Prime is the ANSI, ASCII 7-bit set with the 8 parity bit always on. Tables C-1 and C-2 present the ASCII nonprinting and printing character sets respectively.

PRIME USAGE

Prime hardware and software uses standard ASCII for communications with devices. The following points are particularly important to Prime usage:

- Output parity is normally transmitted as zero (space) unless the device requires otherwise, in which case software will compute transmitted parity. Some controllers (MLC) may have hardware to assist in parity generations.

- Input parity is always represented as a 1 by hardware and by standard software. Input drivers are responsible for making the parity bit suit the host software requirements. Some controllers (MLC) may assist in parity error detection.

- The Prime internal standard for the parity bit is one, that is, '200 is added to the octal value.

SPECIAL CHARACTERS

The following characters have special meanings in Prime usage:

CTRL-P	(octal 220)	is interpreted as a .BREAK.
.CR.	(octal 215)	is interpreted as a newline (.NL.)
"	(octal 242)	is the default for character erase
?	(octal 277)	is the default for line kill
\	(octal 334)	is interpreted as a logical tab (Editor)

KEYBOARD INPUT

Nonprinting characters may be entered into text using one of Prime's text editors with the logical escape character ^ and the octal value. The character is interpreted by output devices according to their hardware.

For example, typing ^207 will enter one character into the text.

Table C-1

ASCII Character Set (Non-Printing)
(Conforms to ANSI X3.4-1968)

Octal Value	ASCII Char	Comments/Prime Usage	Control Char
200	NULL	Null character - filler	^@
201	SOH	Start of header (communications)	^A
202	STX	Start of text (communications)	^B
203	ETX	End of text (communications)	^C
204	EOT	End of transmission (communications)	^D
205	ENQ	End of I.D. (communications)	^E
206	ACK	Acknowledge affirmative (communications)	^F
207	BEL	Audible alarm (bell)	^G
210	BS	Back space one position (carriage control)	^H
211	HT	Physical horizontal tab	^I
212	LF	Line feed; ignored as terminal input	^J
213	VT	Physical vertical tab (carriage control)	^K
214	FF	Form feed (carriage control)	^L
215	CR	Carriage return (carriage control) (1)	^M
216	SO	RRS-red ribbon shift	^N
217	SI	BRS-black ribbon shift	^O
220	DLE	RCP-relative copy (2)	^P
221	DC1	RHT-relative horizontal tab (3)	^Q
222	DC2	HLF-half line feed forward (carriage control)	^R
223	DC3	RVT-relative vertical tab (4)	^S
224	DC4	HLR-half line feed reverse (carriage control)	^T
225	NAK	Negative acknowledgement (communications)	^U
226	SYN	Synchronocity (communications)	^V
227	ETB	End of transmission block (communications)	^W
230	CAN	Cancel	^X
231	EM	End of Medium	^Y
232	SUB	Substitute	^Z
233	ESC	Escape	^[
234	FS	File separator	^\
235	GS	Group separator	^]
236	RS	Record separator	^^
237	US	Unit separator	^_

Notes to Table C-1

(1) Generally, CR is interpreted as .NL. at the terminal. In Pascal, however, CR (or LF) always returns from textfiles as a blank.

(2) .BREAK. at terminal. Relative copy in file; next byte specifies number of bytes to copy from corresponding position of preceeding line.

(3) Next byte specifies number of spaces to insert.

(4) Next byte specifies number of lines to insert.

Table C-2

ASCII Character Set (Printing)
(Conforms to ANSI X3.4-1968—
1963 Variances are noted)

Octal Value	ASCII Character	OCTAL Value	ASCII Character	OCTAL Value	ASCII Character
240	.SP (1)	300	@	340	` (9)
241	!	301	A	341	a
242	" (2)	302	B	342	b
243	# (3)	303	C	343	c
244	$	304	D	344	d
245	%	305	E	345	e
246	&	306	F	346	f
247	' (4)	307	G	347	g
250	(	310	H	350	h
251	)	311	I	351	i
252	*	312	J	352	j
253	+	313	K	353	k
254	, (5)	314	L	354	l
255	-	315	M	355	m
256	.	316	N	356	n
257	/	317	O	357	o
260	0	320	P	360	p
261	1	321	Q	361	q
262	2	322	R	362	r
263	3	323	S	363	s
264	4	324	T	364	t
265	5	325	U	365	u
266	6	326	V	366	v
267	7	327	W	367	w
270	8	330	X	370	x
271	9	331	Y	371	y
272	:	332	Z	372	z
273	;	333	[	373	{
274	<	334	\	374	|
275	=	335	]	375	}
276	>	336	^(7)	376	~ (10)
277	? (6)	337	_ (8)	377	DEL (11)

Notes to Table C-2

(1) Space forward one position

(2) Default terminal usage - erase previous character

(3) £ in British use

(4) Apostrophe/single quote

(5) Comma

(6) Default terminal usage - kill line

(7) 1963 standard ^; terminal use - logical escape

(8) 1963 standard <-; underscore "_"

(9) Grave

(10) 1963 standard ESC

(11) Rubout - ignored

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)
19.2 128-bit Float quad precision	—	—	—	REAL*16	—	—
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)
19.21 Varying character string	—	—	—	—	STRING[n]	Char(n) Varying
48-bits (three words)	—	—	—	—	^<type>	Pointer
256 bits	—	—	—	—	SET	Bit(256)

— Not available.

Note

Whenever you call an external subprogram from a Pascal program, you must use Prime's EXTERN attribute in the procedure or function heading. (Chapter 9 discusses external subprograms.)

INTERFACING INTEGER, BOOLEAN, AND ENUMERATED

Pascal's INTEGER, BOOLEAN, and enumerated type values are compatible with:

- PL/I-G's Fixed Bin(15)
- FORTRAN 77's INTEGER*2 and LOGICAL*2
- FORTRAN IV's INTEGER, INTEGER*2, and LOGICAL
- COBOL's COMP
- BASIC/VM's INT

Each value of any of these data types is stored in 16 bits (one word).

In an enumerated type, the ordinal INTEGER value of each element is stored. The first element's ordinal value is 0, the second is 1, etc. Given the following type:

```
TYPE
  COLORS = (RED, YELLOW, GREEN);
```

The ordinal numbers of RED, YELLOW, and GREEN are 0, 1, and 2 respectively.

Here is a Pascal program that passes an element of an enumerated type to a PL/I-G procedure:

```
PROGRAM Enumerate;
TYPE
  A = (RED, ORANGE, GREEN, BLUE, YELLOW, PURPLE);
VAR
  X : A;
PROCEDURE MODIFY_ENUMERATED(VAR Z : A); EXTERN;
BEGIN {main program}
  X := BLUE;
  IF X = BLUE THEN
    WRITELN('X equals BLUE');
  MODIFY_ENUMERATED(X);  {PL/I-G procedure is called here}
  IF X = YELLOW THEN
    WRITELN('X now equals YELLOW')
  ELSE
    WRITELN('X does not equal YELLOW')
END. {main program}
```

Here is the PL/I-G procedure that accepts the Pascal enumerated type value, changes it, and passes it back to the Pascal program:

```
MODIFY_ENUMERATED : PROCEDURE(DUMMY);
  DECLARE DUMMY FIXED BIN(15);
  DUMMY = DUMMY + 1;  /*ordinal value of BLUE changes to YELLOW*/
END; /*procedure PL1GPROC*/
```

INTERFACING LONGINTEGER

Prime Pascal's extension type LONGINTEGER is compatible with:

- PL/I-G's Fixed Bin(31)
- FORTRAN 77's INTEGER, INTEGER*4, LOGICAL, and LOGICAL*4
- FORTRAN IV's INTEGER, INTEGER*2, and LOGICAL
- BASIC/VM's INT*4

Each Prime Pascal LONGINTEGER value, and any value of these other data types, is stored in 32 bits (two words).

19.1

INTERFACING REAL

Pascal's REAL type values are compatible with:

- PL/I-G'S Float Bin(23)
- FORTRAN 77's REAL and REAL*4
- FORTRAN IV's REAL and REAL*4
- BASIC/VM's REAL

Each Pascal REAL value, and any value of these other data types, is stored in 32 bits (two words).

INTERFACING LONGREAL

Prime Pascal's extension type LONGREAL are compatible with:

- PL/I-G's Float Bin(47)
- FORTRAN 77's REAL*8
- FORTRAN IV's REAL*8
- BASIC/VM's REAL*8

19.1

Each Prime Pascal LONGREAL value, and any value of these other data types, is stored in 64 bits (four words).

INTERFACING CHAR AND ARRAY OF CHAR

Pascal's CHAR and ARRAY OF CHAR values are compatible with:

- PL/I-G's Char(n)
- FORTRAN 77's CHARACTER*n
- FORTRAN IV's INTEGER
- COBOL's DISPLAY(5), PIC A(n), PIC 9(n), and PIC X(n)
- BASIC/VM's INT

Each Pascal CHAR value, and any value of these other data types, is stored in eight bits (one byte). Values of an ARRAY OF CHAR, and the character array values of the other types, are stored as strings of bytes.

Here is a Pascal program that passes an ARRAY OF CHAR value to Char(n) in a PL/I-G procedure:

```
PROGRAM Change_Char_Array;
TYPE
  STRING5 = ARRAY[1..5] OF CHAR;
VAR
  CHARARRAY : STRING5;
PROCEDURE MODIFY_CHAR_ARRAY (VAR DUMMY : STRING5); EXTERN;
BEGIN {main program}
  CHARARRAY := 'ABCDE';
  MODIFY_CHAR_ARRAY (CHARARRAY);  {PL/I-G procedure is called here}
  IF CHARARRAY = 'ABCDZ' THEN
    WRITELN('The array of char was successfully modified')
  ELSE
    WRITELN('The array of char was not modified')
END.
```

Here is the PL/I-G procedure that accepts the Pascal ARRAY OF CHAR value, changes it, and passes it back to the Pascal program:

```
MODIFY_CHAR_ARRAY : PROCEDURE (DUMMY);
  DECLARE DUMMY CHAR(5);
  DUMMY = 'ABCDZ';
END;
```

INTERFACING POINTER

Pascal's pointer type, which is declared as ^type, is compatible with PL/I-G's POINTER type. Here is an example of a Pascal program that passes a pointer value to a PL/I-G procedure:

```
PROGRAM Test_Pointer;
TYPE
  POINTER_INTEGER = ^INTEGER;
VAR
  P : POINTER_INTEGER;
PROCEDURE ADD_1_TO_POINTER (VAR DUMMY : POINTER_INTEGER); EXTERN;
BEGIN {main program}
  NEW(P);
  P^ := 100;
  ADD_1_TO_POINTER(P); {PL/I-G procedure is called here}
  IF P^ = 101 THEN
    WRITELN('The integer has been changed')
  ELSE
    WRITELN('This does not work')
END.
```

Here is the PL/I-G procedure that accepts the Pascal pointer type value, changes it, and passes it back to the Pascal program:

```
ADD_1_TO_POINTER : PROCEDURE (DUMMY);
  DECLARE DUMMY POINTER;
  DECLARE I FIXED BIN BASED;
  DUMMY -> I = DUMMY -> I + 1;
END;
```

A pointer type value in Pascal and in PL/I-G is stored in 48 bits (three words).

INTERFACING SET

Values of Pascal's SET type are compatible with values of PL/I's Bit(256) type. Each element in a Pascal set is stored in one bit, with a 256-element limit for each set.

Here is an example of a Pascal program that passes a set to a PL/I-G procedure:

```
PROGRAM Pass_set;
TYPE
  SET_TYPE = SET OF 0..255;
VAR
  A, B, C : SET_TYPE;
PROCEDURE MODIFY_SET(VAR DUMMY : SET_TYPE); EXTERN;
BEGIN {main program}
  A := [10, 20, 30];
  B := A;
  C := A + [40];
  IF A = B THEN
    WRITELN('A contains 10, 20, and 30');
  MODIFY_SET(A); {PL/I-G procedure is called here}
  IF A = C THEN
    WRITELN('A now contains 10, 20, 30, 40')
END.
```

Here is the PL/I-G procedure that accepts the Pascal set, changes the contents of that set, and passes back the new set:

```
MODIFY_SET : PROCEDURE(DUMMY);
  DECLARE DUMMY (256)BIT;
  DECLARE I FIXED BIN(15);
  DUMMY(41) = '1'B;
END;
```

INTERFACING RECORD

Values of Pascal record fields can be compatible with the fields of records in other languages. Each field must be compatible with its counterpart in the other language. For example, if your Pascal record has an INTEGER type field and a REAL type field, it would have to match PL/I-G fields declared as Fixed Bin(15) and Float Bin(23) respectively.

Here is an example of a Pascal program that passes a record with an INTEGER field, a REAL field, and a Boolean field to a PL/I-G procedure:

```
PROGRAM Pascal_Modify_Record;
TYPE
  REC = RECORD
    A : INTEGER;
    B : REAL;
    C : BOOLEAN
  END;
VAR
  X : REC;
PROCEDURE MODIFY_RECORD (VAR DUMMY : REC); EXTERN;
BEGIN {main program}
  X.A := 100;
  X.B := 1000.0;
  X.C := TRUE;
  MODIFY_RECORD(X);  {PL/I-G procedure is called here}
  IF ((X.A = 101) AND
      (X.B = 1001.0) AND
      (X.C = FALSE)) THEN
        WRITELN('Record was correctly modified')
      ELSE
        WRITELN('Record was not correctly modified')
END.
```

Here is the PL/I-G procedure that accepts the Pascal record, changes the values of the INTEGER, REAL, and Boolean fields, and passes those new field values back to the Pascal program:

```
MODIFY_RECORD : PROCEDURE (DUMMY);
  DECLARE 1 DUMMY,
    2 A   FIXED BIN(15),
    2 B   FLOAT BIN(23),
    2 C   FIXED BIN(15);
  DUMMY.A = DUMMY.A + 1; /* 1 is added to the integer */
  DUMMY.B = DUMMY.B + 1; /* 1 is added to the real */
  DUMMY.C = 0; /* the Boolean is changed to FALSE */
END;
```

INTERFACING STRING

Prime Pascal's STRING type is compatible with PL/I Subset G's CHARACTER VARYING type. The STRING type is a Prime extension. Here is an example of a Pascal program that passes a STRING value to a PL/I-G procedure:

```
VAR
  ST6 : STRING[6];
PROCEDURE PASS_STRING (VAR STR : STRING[6]); EXTERN;
BEGIN {main program}
  ST6 := 'PA';
  PASS_STRING(ST6);  {PL/I-G procedure is called here}
  IF ST6 = 'PASCAL' THEN
    WRITELN('Pass')  {this will pass}
  ELSE
    WRITELN('Fail')
END.
```

19.2

Here is the PL/I-G procedure that accepts the Pascal STRING value, changes it, and passes the new value back to the Pascal program:

```
PASS_STRING : PROCEDURE (DUMMY6);
  DECLARE DUMMY6 CHARACTER(6) VARYING;
  DECLARE DUMMY4 CHARACTER(4) VARYING;
  DUMMY4 = 'SCAL';
  DUMMY6 = DUMMY6 || DUMMY4;
END;
```

Note

If your installation does not use Rev. 19.2 Pascal, see the Subroutines Reference Guide (Rev. 19 or higher) for interfacing Pascal character strings with PL/I-G CHARACTER VARYING strings.

INDEX

Index

$ and ! integer operators (Prime extension) 7-7, A-3

$ and _ in identifiers (Prime extension) 4-8, A-3

-32I compiler option 2-13

-64V compiler option 2-13

A compiler switch (Prime extension) 2-16, A-3

Abbreviations, compile option 2-13 to 2-15

ABS function 6-4, 11-1

Actual parameters 4-4, 9-1, 9-2

Allocating dynamic variables 6-32, 6-33

AND operator 7-6

ANSI standard 1-4, 2-12, 4-4, 6-8, C-1 to C-6

ARCTAN function 6-6, 11-2

Arithmetic operators 7-2, 7-3

ARRAY OF CHAR 6-17 to 6-19

ARRAY OF CHAR, (Prime extension) 6-18, 6-19

ARRAY storage format B-5

Array storage 6-16, B-5

ARRAY type 6-14L to 6-20, B-5

Arrays of characters, converting strings to 6-14C, 6-14D

Arrays:
external 6-16
multidimensional 6-19, 6-20, B-5

ASCII character set 4-4, 6-8 to 6-10, C-1 to C-6

Assignment compatibility 8-2, 8-3

Assignment statement 8-2, 8-3

Auxiliary procedures:
  CLOSE (Prime extension) 10-24, A-4
  PAGE 10-23, 10-24

BEGIN and END keywords 4-7, 8-4, 8-5

-BIG and -NOBIG compiler options 2-8

Binary (object) file 2-1, 2-4, 2-5, 2-8, 3-1 to 3-7

-BINARY compiler option 2-8

Blanks 4-11, 4-12

Block:
  declaration part 5-4 to 5-9
  definition 4-2
  description 5-3, 5-4
  executable part 5-9 to 5-11
  illustration 4-3

BOOLEAN operators 7-6

BOOLEAN storage format B-4

BOOLEAN type 6-8, B-4

Boundary-spanning object code 9-5

Call, recursive 9-19 to 9-22

Calling subprograms:
  external 9-16
  functions 9-14
  procedures 8-3, 8-4, 9-10, 9-11

Cardinality, data type 6-1

CASE and variant records 6-23 to 6-25

CASE statement 8-11 to 8-14

Changing compiler option defaults 2-6

Changing erase and kill characters 10-4

CHAR storage format B-4

CHAR type 6-8 to 6-10, B-4

Character set (See ASCII character set)

Character string constants 4-11, 6-9

Character strings 4-11, 6-14 to 6-14L, 6-17 to 6-19, D-5, D-6

CHR function 6-8 to 6-10, 11-3

CLOSE procedure (Prime extension) 6-30, 10-7, 10-12, 10-24, A-4

Closing data files 6-30, 10-7, 10-12, 10-24, A-4

Code, object:
  boundary-spanning 9-5
  ordinary 9-5

Collating sequence 6-8, C-3 to C-6

Command files 3-8

Command level, PRIMOS 2-2, 2-3, 2-5, 3-2 to 3-8

Command line:
  options 2-1, 2-2, 2-6 to 2-15
  Pascal compiler 2-2

Comments 4-11, 4-12

Compatibility with other languages 1-6, 9-1, 9-15 to 9-18, D-1 to D-9

Compile-time errors 2-1 to 2-4

Compiler switches:
  A switch 2-16, A-3
  E switch 2-9, 2-17, 6-16, 6-22, 9-16 to 9-18, A-3

L switch 2-16, A-3
overview 2-16, 4-12
P switch 2-17, A-3

Compiler:
error messages 1-4, 2-1 to 2-4
filename conventions 2-4, 2-5, 3-2 to 3-8
invoking 2-2
option abbreviations 2-13 to 2-15
options 2-1, 2-2, 2-6 to 2-15
PASCAL command 2-2
switches 2-9, 2-16, 2-17, 4-12, 6-16, 6-22, 9-16 to 9-18, A-3

Compiling programs 2-1 to 2-17

Compound statement 8-4, 8-5

Concatenation operator (Prime extension) 6-14E to 6-14F, 7-7, 7-7A, A-4A

Conditional statements:
CASE 8-11 to 8-14
IF 8-10, 8-11

CONST declaration 5-6

Constants:
BOOLEAN 4-9, 6-8
CHAR 4-11, 6-9
character string 4-11, 6-9
declared 5-6
enumerated 6-11, 6-12
INTEGER and LONGINTEGER 4-10, 6-4, 6-5
MAXINT 4-9, 6-3
NIL 6-32, 6-33, 7-1
numeric 4-8, 4-10
REAL and LONGREAL 4-10, 6-6, 6-7
standard 4-9
STRING 4-11
subrange 6-13, 6-14

Control (nonprintable) characters 6-8, 11-3, 11-4, C-5, C-6

Control statements:
CASE 8-11 to 8-14
FOR 8-8, 8-9
GOTO 8-14, 8-15
IF 8-10, 8-11
nested 8-5, 8-7, 8-9 to 8-11
REPEAT 8-6
WHILE 8-7, 8-8

Control-C end-of-file marker 10-22

Conventions, filename (See Filename conventions)

COS function 6-6, 11-1

CPL files 3-8

Creating data files:
input 10-6 to 10-11
output 10-11 to 10-14

Creating dynamic variables 6-32, 6-33

Data file I/O 6-30, 10-6 to 10-24

Data files:
closing 6-30, 10-7, 10-12, 10-24, A-4
creating 10-6 to 10-14
opening 6-30, 10-6 to 10-14

Data format (See Storage format)

Data type cardinality 6-1

Data types:
ARRAY 6-14 to 6-10, B-5
BOOLEAN 6-8, B-4
CHAR 6-8 to 6-10, B-4
enumerated 6-10 to 6-12, B-4
FILE 6-27 to 6-31, 10-6 to 10-24, B-6, B-7
illustration 6-2
INTEGER 6-3, 6-4, B-2
interfacing with other languages D-1 to D-8
LONGINTEGER (Prime extension) 6-4, 6-5, A-1, B-2
LONGREAL (Prime extension)

6-7, A-1, B-3
overview 6-1
pointer 6-31 to 6-33, B-8
REAL 6-6, B-3
RECORD 6-20 to 6-25, B-5
SET 6-25 to 6-27, B-5
standard scalar 6-2 to 6-10
storage formats B-1 to B-8
STRING (Prime extension) 6-14 to 6-14L, A-4A, B-9, D-9
structured 6-14 to 6-31
subrange 6-12 to 6-14, B-3
TEXT 6-29, 10-10
user-defined scalar 6-10 to 6-14

-DEBUG and -NODEBUG compiler options 2-8

Debugger utility 1-5, 2-6 to 2-8, 2-14

Decimal notation 4-10, 6-6, 6-7, 10-21

Declarations:
CONST 5-6
description 5-3, 5-4
LABEL 5-4, 5-5
order of (Prime extension) 5-3, A-2
PROCEDURE and FUNCTION 5-9
TYPE 5-6, 5-7
VAR 5-7 to 5-9

Default field widths 10-20

Default options 2-6

DELETE function (Prime extension) 6-14J to 6-14L, 11-5, A-4A

Delimiters, comment 4-12

Designator, function 9-14

Destroying dynamic variables 6-32, 6-33

Directives:
EXTERN 4-9, 9-15, A-3
FORWARD 4-9, 9-15

DISPOSE procedure 6-32

DIV operator 6-4, 7-3

Documents related to Pascal 1-4, 1-5

Dollar signs and underscores in identifiers 4-8, A-3

Dynamic allocation procedures:
DISPOSE 6-32
NEW 6-32

Dynamic storage 6-32, B-8

Dynamic variables 6-31 to 6-33

E compiler switch (Prime extension) 2-9, 2-17, 6-16, 6-22, 9-16 to 9-18, A-3

EDITOR 1-4, 1-5, 10-6, 10-8 to 10-10

Elements, Pascal language 4-1 to 4-12

EMACS editor 1-5, 10-6, 10-8 to 10-10

Empty record 6-25

Empty set 6-26

Empty statement 8-5

END and BEGIN keywords 4-7, 8-4, 8-5

End of File (EOF) condition 10-22, 10-23

End of Line (EOLN) condition 10-23

Enumerated storage format B-4

Enumerated type 6-10 to 6-12, B-4

EOF function 10-22, 11-5

EOLN function 10-23, 11-5

Erase and kill characters:
  changing 10-4
  overview 10-4
  using on terminal input 10-4 to 10-6
  with -INTERACTIVE switch 10-4 to 10-6

Erasing terminal input 10-4 to 10-6

Error messages:
  compile time 2-1 to 2-4
  for %INCLUDE files 2-3, 2-4
  for -INTERACTIVE switch 10-5
  for data types 6-5, 6-7, 6-12 to 6-14, 6-19, 6-27, 11-2
  for external subprograms 9-18
  for identifiers 4-8, A-5
  for keyword PACKED 6-14, A-5
  for labels 5-5
  for non-ANSI standard 2-12
  for PACK and UNPACK 9-12, A-5
  for parameters 9-4
  for standard functions 6-12, 11-2
  format of 2-2
  in listing file 2-10
  loading 3-3, 3-6
  overview 1-4
  runtime 2-11, 3-3, 3-6, 6-12, 10-5
  severity codes 2-3
  significance 2-3
  suppressing of 2-8, 2-12

-ERRITY and -NOERRITY compiler options 2-8

Executable (SEG) file 2-5, 3-2 to 3-8

Executable block part 5-9 to 5-11

Executable statements 5-9 to 5-11, 8-1 to 8-16

EXECUTE (load subprocessor) command 3-8

Executing programs 3-7, 3-8

EXP function 6-6, 11-2

-EXPLIST and -NOEXPLIST compiler options 2-9

Exponents 4-10, 6-6, 6-7, 10-20, B-3

Expressions 7-1 to 7-8

Extensions (See Prime extensions)

EXTERN (Prime extension) directive 9-15, A-3

-EXTERNAL and -NOEXTERNAL compiler options 2-9, 9-17

External arrays 6-16

External procedures and functions (See External subprograms)

External records 6-22

External subprograms:
  calling 9-16
  declaring 9-15, 9-16, 9-18
  EXTERN (Prime extension) directive 9-15, 9-16, A-3
  from libraries 9-19
  overview 9-1, 9-15
  written in other languages 9-18, D-1 to D-8
  written in Pascal 9-16 to 9-18

External variables 9-17

Field widths 10-18 to 10-22

Fields, variant 6-23 to 6-25

File control block B-6, B-7

File I/O 6-30, 10-6 to 10-24

FILE OF CHAR 6-28 to 6-31, 10-8 to 10-12

FILE OF CHAR, reading and writing of 10-9, A-5

FILE OF INTEGER 6-28, 6-29, 10-8, 10-9

FILE OF REAL 6-28, 6-29, 10-8, 10-9

FILE storage format B-6, B-7

File storage B-6, B-7

FILE type 6-27 to 6-31, 10-6 to 10-24, B-5

File variables 6-27 to 6-31, 10-6 to 10-24

File-handling functions:
EOF 10-22, 11-5
EOLN 10-23, 11-5

File-handling procedures:
CLOSE (Prime extension) 6-30, 10-7, 10-12, A-4, 10-2
GET 10-15
PAGE 10-23, 10-24
PUT 10-18
READ 10-15 to 10-17
READLN 10-17
RESET 10-6 to 10-10
REWRITE 10-11 to 10-13
WRITE 10-18 to 10-21
WRITELN 10-22

Filename conventions:
overview 2-4
prefix 2-5, 3-4 to 3-7
suffix 2-5, 3-2 to 3-4, 3-7
table 3-7

Filename in RESET and REWRITE (Prime extension) 10-6 to 10-14, A-4

Files, data (See Data files)

Files:
%INCLUDE 2-3, 2-4, 5-10, 5-11, A-2
(See also Textfiles)
closing 6-30, 10-7, 10-12, 10-23, A-4
of CHAR 6-28 to 6-31, 10-8 to 10-12
of INTEGER 6-28, 6-29, 10-8, 10-9
of REAL 6-28, 6-29, 10-8, 10-9
opening 6-30, 10-6 to 10-14
PRIMOS input/output 6-27 to 6-31, 9-16 to 9-18, 10-1, 10-5 to 10-24
standard INPUT 4-9, 6-31, 10-5, 10-10, 10-11, 10-16, 10-17, A-4, 10-22
standard OUTPUT 4-9, 6-31, 10-13, 10-14, 10-18, 10-22, 10-23, A-4
storage of data B-6, B-7

Fixed variant record part 6-23

FOR statement 8-8, 8-9

Formal parameters 4-4, 9-2, 9-3

Format, line 4-11

FORWARD directive 9-15

Forward procedures and functions 9-14, 9-15

-FRN and -NOFRN compiler options 2-9

FUNCTION declaration 5-9, 9-13, 9-15

Function designator 9-14

Functions:
(See also Subprograms)
declarations 9-13
external 9-15 to 9-19
forward 9-14, 9-15
heading 9-13, 9-15
I/O 10-14 to 10-24
invoking 9-14
overview 9-1, 9-12
recursive 9-19 to 9-22
standard 4-9, 6-4 to 6-12, 9-14, 10-1, 10-15, 10-22, 10-23, 11-1 to 11-5
STRING 6-14C, 6-14D, 6-14J to

6-14L, 11-5, A-4A

GET procedure 10-15

Global:
definition 4-2, 5-8, 9-10
external variables 9-17
illustration 4-3

GOTO statement 5-5, 8-14, 8-15

Graphic (printable) characters
6-8 to 6-10, 11-3, 11-4, C-5, C-6

Heading:
external 9-15
function 9-13, 9-15
procedure 9-9, 9-15
program 5-1 to 5-3

I/O at terminal 6-29, 6-30, 10-2 to 10-6

I/O procedures and functions:
CLOSE (Prime extension) 10-24
EOF 10-22, 11-5
EOLN 10-23, 11-5
GET 10-15
overview 10-1, 10-14
PAGE 10-23, 10-24
PUT 10-18
READ 10-15 to 10-17
READLN 10-17
RESET 10-6 to 10-10
REWRITE 10-11 to 10-13
WRITE 10-18 to 10-21
WRITELN 10-22

Identifier length (Prime restriction) 4-7, A-5

Identifiers:
dollar signs and underscores in (Prime extension) 4-8, A-3
standard 4-8, 4-9
user-defined 4-8

IF statement 8-10, 8-11

IN operator 6-27, 7-4

INCLUDE files 2-3, 2-4, 5-10, 5-11, A-2

INDEX function (Prime extension) 6-14J to 6-14L, 11-5, A-4A

Index, array 6-14, 6-15

INPUT and OUTPUT, use of 6-31, 10-11, 10-13, A-4

Input and output:
overview 10-1
procedures and functions 10-14 to 10-24
to and from data files 6-30, 10-6 to 10-24
to and from terminal 6-30, 10-2 to 10-6

-INPUT compiler option 2-9

INPUT, standard textfile 4-9, 6-31, 10-5, 10-10, 10-11, 10-16, 10-17, 10-22, 10-23

INSERT function (Prime extension) 6-14J to 6-14L, 11-5, A-4A

Integer (Prime extension) operators 7-7, A-3

INTEGER storage format B-2

INTEGER type 6-3, 6-4, B-2

-INTERACTIVE switch (Prime extension) 10-4 to 10-6, A-2, B-6

Interfacing Pascal to other languages:
ARRAY OF CHAR interface D-5
BOOLEAN interface D-3
CHAR interface D-5
compatibility table D-2
enumerated interface D-3
INTEGER interface D-3
LONGINTEGER interface D-4
LONGREAL interface D-5
overview 1-6, 9-1, 9-15 to 9-18, D-1, D-2
pointer interface D-6

REAL interface D-4
RECORD interface D-7 to D-8
SET interface D-7
STRING interface D-9

Internal representations (See Storage format)

Invoking external subprograms 9-16

Invoking functions 9-14

Invoking procedures 8-3, 8-4, 9-10, 9-11

Invoking the compiler 2-2

Keywords 4-7

Kill character (See Erase and kill characters)

L compiler switch (Prime extension) 2-16, A-3

LABEL declaration 5-4, 5-5

Language elements 4-1 to 4-12

Language interfaces 1-6, 9-1, 9-15 to 9-18, D-1 to D-9

LENGTH function (Prime extension) 6-14J to 6-14L, 11-5, A-4A

Libraries:
loading 3-1 to 3-6
Prime system 3-1 to 3-6, 9-19

LIBRARY (load subprocessor) command 3-2

Library, Pascal 3-1 to 3-6

Line format 4-11

-LISTING compiler option 2-10

Listing file (See Source listing file)

LN function 6-6, 11-1

LOAD (load subprocessor) command 3-2

-LOAD (SEG option) 3-2

Load subprocessor commands:
EXECUTE 3-8
LIBRARY 3-2
LOAD 3-2
QUIT 3-2

LOAD utility 1-5, 2-5, 3-1 to 3-8

Loading programs:
overview 3-1, 3-2
with prefix method 3-4 to 3-6
with suffix method 3-3, 3-4

Loading subprograms 3-1, 3-3 to 3-6, 9-18

Local:
definition 4-4, 5-8, 9-10
external variables 9-17
recursive variables 9-19

LONGINTEGER storage format (Prime extension) B-2

LONGINTEGER type (Prime extension) 6-4, 6-5, A-1, B-2

LONGREAL storage format (Prime extension) B-3

LONGREAL type (Prime extension) 6-7, A-1, B-2

LTRIM function (Prime extension) 6-14J to 6-14L, 11-5, A-4A

-MAP and -NO_MAP compiler options 2-10

MAXINT 4-9, 6-3

Messages:
end-of-compilation 2-2, 2-3
error (See Error messages)

MOD operator 6-4, 7-3

Multidimensional arrays 6-19, 6-20, B-5

Nested statements:
  defined 8-5
  FOR 8-9
  IF 8-10, 8-11
  WHILE 8-7

NEW procedure 6-32

NIL 6-32, 6-33, 7-1

Non-ANSI standard errors 2-12

Nonprintable (control) characters 6-8, 11-3, 11-4, C-5, C-6

NOT operator 7-6

Notation:
  decimal 4-10, 6-6, 6-7, 10-21
  scientific 4-10, 6-6, 6-7, 10-20, B-3

Null program 5-4

Null string 6-14B, A-4A

Numeric constants 4-8, 4-10

Object (binary) file 2-1, 2-4, 2-5, 2-8, 3-1 to 3-7

ODD function 11-5

-OFFSET and -NOOFFSET compiler options 2-10

Opening data files:
  input 6-30, 10-6 to 10-11
  output 6-30, 10-11 to 10-14
  RESET 10-6 to 10-10

Operands 7-1, 7-2

Operator precedence 7-7

Operator, string concatenation (Prime extension) 6-14E, 6-14F, 7-7, 7-7A, A-4

Operators, arithmetic:
  - 6-4, 6-6, 7-3
  * 6-4, 6-6, 7-3
  + 6-4, 6-6, 7-3
  / 6-6, 7-3
  DIV 6-4, 7-3
  MOD 6-4, 7-3

Operators, BOOLEAN:
  AND 7-6
  NOT 7-6
  OR 7-6

Operators, integer (Prime extension):
  ! 7-7
  & 7-7

Operators, relational:
  < 6-4, 6-8, 6-10, 6-12, 7-4
  <= 6-4, 6-8, 6-10, 6-12, 6-27, 7-4
  <> 6-4, 6-8, 6-10, 6-12, 6-27, 7-4
  = 6-4, 6-8, 6-10, 6-12, 6-27, 7-4
  > 6-4, 6-8, 6-10, 6-12, 7-4
  >= 6-4, 6-8, 6-10, 6-12, 6-27, 7-4
  IN 6-27, 7-4

Operators, SET:
  - 6-26, 7-5
  * 6-26, 7-5
  + 6-26, 7-5

Operators:
  arithmetic 7-2, 7-3
  BOOLEAN 7-6
  defined 7-1
  integer (Prime extension) 7-7
  order of evaluation 7-8
  precedence of 7-7
  relational 7-3, 7-4
  SET 6-26, 6-27, 7-4, 7-5
  string concatenation (Prime extension) 6-14E, 6-14F, 7-7, 7-7A, A-4

-OPT1 and -NOOPT1 compiler options 2-11

-OPT3 and -NOOPT3 compiler options 2-11

-OPTIMIZE and -NOOPTIMIZE compiler options 2-11

Optional program heading (Prime extension) 5-1, A-2

Options, compiler:
abbreviations 2-13 to 2-15
-BIG and -NOBIG 2-8
-BINARY 2-8
commonly used 2-7
-DEBUG and -NODEBUG 2-8
defaults 2-6
-ERRITY and -NOERRITY 2-8
-EXPLIST and -NOEXPLIST 2-9
-EXTERNAL and -NOEXTERNAL 2-9, 9-17
-FRN and -NOFRN 2-9
-INPUT 2-9
-LISTING 2-10
-MAP and -NO_MAP 2-10
not commonly used 2-7
-OFFSET and -NOOFFSET 2-10
-OPT1 and -NOOPT1 2-11
-OPT3 and -NOOPT3 2-11
-OPTIMIZE and -NOOPTIMIZE 2-11
-PRODUCTION and -NOPRODUCTION 2-11
-RANGE and -NORANGE 2-11
-SILENT and -NOSILENT 2-12
-SOURCE 2-12
-STANDARD and -NOSTANDARD 2-12
-STATISTICS and -NOSTATISTICS 2-12
-UPCASE 2-13
-XREF and -NOXREF 2-13

OR operator 7-6

ORD function 6-10, 6-12, 11-3

Order of declarations (Prime extension) 5-3, A-2

Order of evaluation 7-8

Ordinal values 6-8 to 6-12, 11-3, 11-4

OTHERWISE (Prime extension) clause 8-11, 8-14, 8-15, A-3

OUTPUT, standard textfile 4-9, 6-31, 10-13, 10-14, 10-18, 10-22, 10-23, A-4

P compiler switch (Prime extension) 2-17, A-3

PACK and UNPACK procedures (Prime restrictions) 9-12, A-5

Packed arrays 6-14, 6-17, A-5

PACKED keyword (Prime restriction) 6-14, 6-17, A-5

Page breaks in listing file 2-17, A-3

PAGE procedure 10-23, 10-24

Parameters:
actual 4-4, 9-2
array variable 9-5
formal 4-4, 9-2, 9-3
overview 4-4, 9-1, 9-2
procedures and functions passed as 9-6 to 9-9
record variable 9-5
value 9-3
variable 9-3, 9-4

PASCAL command 2-2

Pascal:
ANSI standard 1-4, 2-12, 4-4, 6-8, C-1 to C-6
arithmetic operators 7-2, 7-3
ASCII character set 4-4, 6-8 to 6-10, C-1 to C-6
blanks 4-11, 4-12
BOOLEAN operators 7-6
character strings 4-11, 6-14 to 6-14L, 6-17 to 6-19, D-5, D-6
comments 4-11, 4-12
compiler 2-1 to 2-17
data storage formats B-1 to B-8
data types 6-1 to 6-34
expressions 7-1 to 7-8

identifiers 4-7 to 4-9
input and output 10-1, 6-27 to 6-31
instruction books 1-1
integer operators 7-7
keywords 4-7
language elements 4-1 to 4-12
language interfaces D-1 to D-8, 1-6, 9-1, 9-15 to 9-18
library 3-1 to 3-6
line format 4-11
numeric constants 4-8
operands 7-1, 7-2
operator precedence 7-7, 7-8
operators 7-2 to 7-8
parameters 9-2 to 9-9
Prime extensions 1-2, A-1 to A-4
Prime Pascal 1-2
Prime restrictions 1-2, A-5
procedures and functions 9-1 to 9-22
program structure 5-1 to 5-15
punctuation symbols 4-5, 4-6
related documents 1-4, 1-5
relational operators 7-3, 7-4
separators 4-11
set operators 7-5
standard functions 4-9, 6-4 to 6-12, 9-14, 10-1, 10-15, 10-22, 10-23,
standard procedures 9-12, 10-1, 10-7, 10-11, 10-14 to 10-24
statements 8-1 to 8-16
storage requirements 6-17, 6-22, 6-32, B-1 to B-9, D-1, D-2, 6-16

Pass-by-reference parameters 9-3, 9-4

Pass-by-value parameters 9-3

PMA (See Prime Macro Assembler)

Pointer data type 6-31 to 6-33, B-8

Pointer storage format B-8

Precedence of operators 7-7

PRED function 6-10, 6-12, 11-4

Prefix:
executing file 3-7
filename conventions 2-5, 3-4 to 3-7
loading procedure 3-4 to 3-7

Prime extensions to standard Pascal:
$ and _ in identifiers 4-8, A-3
%INCLUDE files 2-3, 2-4, 5-10, 5-11, A-2
& and ! integer operators 7-7, A-3
A compiler switch 2-16, A-3
ARRAY OF CHAR enhancement 6-18, 6-19, A-2
CLOSE procedure 6-30, 10-7, 10-12, 10-23,
comment delimiters /* */ 4-11, A-2
E compiler switch 2-9, 2-17, 6-16, 6-22,
EXTERN directive 9-15, A-3
Filename in RESET and REWRITE 10-6 to 10-24, A-4
-INTERACTIVE switch 10-4 to 10-6, A-2, B-6
L compiler switch 2-16, A-3
LONGINTEGER type 6-4, 6-5, A-1
LONGREAL type 6-7, A-1, B-3
optional program heading 5-1, A-2
order of declarations 5-3, A-2
OTHERWISE clause 8-11, 8-14, 8-15, A-3
P compiler switch 2-17, A-3
string concatenation operator 6-14E, 6-14
STRING data type 6-14 to 6-14L, A-4, A-4A, D-9
STRING functions 6-14C, 6-14D 6-14J to 6-14L, 11-5, A-4A
string, null 6-14B, A-4A
-TTY switch 10-5, 10-11, 10-14, A-2

Prime Macro Assembler 1-6, 2-9, 9-18

Prime Pascal:
ASCII character set 6-8 to 6-10, C-1 to C-6
compiler 2-1 to 2-17
defined 1-2
extensions 1-2, A-1 to A-4A
library 3-1 to 3-6, 9-19
related documents 1-4, 1-5
restrictions 1-2, A-5

Prime restrictions to standard Pascal:
FILE OF CHAR, reading/writing of 10-9, A-5
identifier length 4-7, A-5
PACK procedure 9-12, A-5
PACKED keyword 6-14, 6-17, A-4
UNPACK procedure 9-12, A-5

Prime:
debugging utility 1-5, 2-6 to 2-8, 2-14
documents related to Pascal 1-4, 1-5
filename conventions 2-4, 2-5, 3-2 to 3-8
high-level languages 1-4, 1-6, 9-1, 9-15 to 9-18, D-1 to D-8
input and output 10-1 to 10-24
libraries 3-1 to 3-6, 9-19
SEG loading utility 1-5, 3-1 to 3-8, 2-5
subroutines 1-5, 9-19
text editors 1-4, 1-5, 10-6, 10-8 to 10-10

PRIMOS:
command level 2-2, 2-3, 2-5, 3-2 to 3-8
data files 6-27 to 6-31, 10-1, 10-5 to 10-24, 9-16 to 9-18
erase and kill characters 10-4 to 10-6
file variables 10-7, 10-8, 10-13
PASCAL command 2-2, 9-17
SEG command 3-2
subroutines 1-5, 9-19
user file directories 6-27, 10-6 to 10-8, 10-12

Printable (graphic) characters 6-8 to 6-10, 11-3, 11-4, C-5, C-6

PROCEDURE declarations 5-9, 9-9, 9-10, 9-15

Procedure statement 8-3, 8-4, 9-10, 9-11

Procedures:
(See also Subprograms)
declarations 5-9, 9-9, 9-10
dynamic allocation 6-32
external 9-15 to 9-19
forward 9-14, 9-15
heading 9-9, 9-15
I/O 10-14 to 10-24
invoking 8-3, 8-4, 9-10, 9-11
overview 9-1, 9-9
recursive 9-19 to 9-22
standard 4-9, 6-32, 9-12, 10-1, 10-7, 10-11, 10-14 to 10-24

-PRODUCTION and -NOPRODUCTION compiler options 2-11

Program definition 4-2

Program heading:
definition 4-2
description 5-1 to 5-3

Program structure:
declaration part 5-3 to 5-9
executable part 5-9 to 5-15
heading 5-1 to 5-3
overview 5-1

Program unit definition 4-2

Program, null 5-4

Punctuation symbols 4-5, 4-6

PUT procedure 10-18

QUIT (load subprocessor) command 3-2

-RANGE and -NORANGE compiler options 2-11

READ procedure 10-15 to 10-17

Reading arrays 6-14 to 6-19

READLN procedure 10-17

REAL storage format B-3

REAL type 6-6, B-3

RECORD storage format B-5

Record storage 6-22, B-5

RECORD type 6-20 to 6-25, B-5

Records:
  empty 6-25
  external 6-22
  using WITH 6-22, 6-23
  variant 6-23 to 6-25

Recursive procedures and functions 9-19 to 9-22

Relational operators 7-3, 7-4

REPEAT statement 8-6

Repetitive statements:
  FOR 8-8, 8-9
  REPEAT 8-6
  WHILE 8-7, 8-8

RESET procedure 10-6 to 10-10

Restrictions (See Prime restrictions)

ROUND function 6-4, 11-2

RUNOFF utility 1-4, 1-5

Runtime errors 2-11, 3-3, 3-6, 6-12, 10-5

Scalar data types:
  standard 6-2 to 6-10
  user-defined 6-10 to 6-14

Scientific notation 4-10, 6-6, 6-7, 10-20, B-3

Scope, definition 4-4

SEG command 3-2

SEG loading utility 1-5, 2-5, 3-1 to 3-8

Separators 4-11

SET operators 6-26, 6-27, 7-4, 7-5

SET storage format B-5

SET type 6-25 to 6-27, B-5

Set, empty 6-26

Severity codes 2-3

-SILENT and -NOSILENT compiler options 2-12

SIN function 6-6, 11-1

-SOURCE compiler option 2-12

Source listing file 2-1, 2-2, 2-4, 2-5, 2-10, 3-4 to 3-7

Source program file 2-1 to 2-6, 3-2, 3-4, 3-6, 3-7

SQR function 6-4, 11-1

SQRT function 6-6, 11-2

-STANDARD and -NOSTANDARD compiler options 2-12

Standard constants 4-9

Standard functions (See Functions)

Standard identifiers 4-8, 4-9

Standard procedures (See Procedures)

Standard scalar data types 6-2 to 6-10

Standard textfiles (See INPUT and OUTPUT)

Statements, declaration (See Declarations)

Statements, executable:
  assignment 8-2, 8-3
  compound 8-4, 8-5
  control 8-5 to 8-15
  empty 8-5
  function designator 9-14
  overview 8-1
  procedure 8-3, 8-4
  WITH 8-16

Statements, nested (See Nested statements)

Static variables 6-31

-STATISTICS and -NOSTATISTICS compiler options 2-12

Storage format:
  ARRAY B-5
  CHAR B-4
  enumerated B-4
  file control block B-6, B-7
  FILE B-6, B-7
  INTEGER B-2
  LONGINTEGER (Prime extension) B-2
  LONGREAL (Prime extension) B-3
  pointer B-8
  REAL B-3
  RECORD B-5
  SET B-5
  STRING (Prime extension) B-9
  subrange B-3

Storage:
  array capacity 6-16
  compatibility D-1 to D-8
  data formats B-1 to B-8
  dynamic 6-32
  illustrations B-1 to B-8
  in other languages D-1
  record capacity 6-22

STR function (Prime extension) 6-14C, 6-14D, 6-14J to 6-14L, 11-5, A-4A

STRING data type (Prime extension) 6-14 to 6-14L, A-4, A-4A, D-9

STRING functions (Prime extension):
  DELETE 6-14J to 6-14L, 11-5, A-4A
  INDEX 6-14J to 6-14L, 11-5, A-4A
  INSERT 6-14J to 6-14L, 11-5, A-4A
  LENGTH 6-14J to 6-14L, 11-5, A-4A
  LTRIM 6-14J to 6-14L, 11-5, A-4A
  STR 6-14C, 6-14D, 6-14J to 6-14L, 11-5, A-4A
  SUBSTR 6-14J to 6-14L, 11-5, A-4A
  TRIM 6-14J to 6-14L, 11-5, A-4A
  UNSTR 6-14C, 6-14D, 6-14J to 6-14L, 11-5, A-4A

STRING storage format B-9

Strings (Prime extension):
  (See also Character strings)
  arrays converted to 6-14C, 6-14D
  assigning 6-14B to 6-14D
  comparing 6-14E
  concatenating 6-14E, 6-14F, 7-7, 7-7A
  converting arrays to 6-14C, 6-14D
  declaring 6-14A
  interfacing other languages with D-9
  null 6-14B, A-4A
  passing to procedures and functions 6-14I, 6-14J
  reading and writing 6-14G to 6-14I
  vs. arrays of characters 6-17

Structured data types 6-14 to 6-31

Subprograms, external (See External subprograms)

Subprograms:
(See also Procedures and functions)
defined 4-2, 9-1
external 9-15 to 9-19
forward 9-14, 9-15
from libraries 9-19
recursive 9-19 to 9-22
written in other languages 9-18, D-1 to D-8

Subrange storage format B-3

Subrange type 6-12 to 6-14, B-3

Subroutines 1-5, 9-19

SUBSTR function (Prime extension) 6-14J to 6-14L, 11-5, A-4A

SUCC function 6-10 to 6-12, 11-3, 11-4

Suffix:
executing file 3-7
filename conventions 2-5, 3-2 to 3-4, 3-7
loading procedure 3-2 to 3-4

Suppressing error messages 2-8, 2-12

Switches (Prime extension):
-INTERACTIVE 10-4 to 10-6, A-2, B-6
-TTY 10-5, 10-11, 10-14, A-2

Switches, compiler (See Compiler switches)

Terminal I/O 6-29, 6-30, 10-2 to 10-6

Text editors, Prime 1-4, 1-5, 10-6, 10-8 to 10-10

TEXT type 6-29, 10-10

Textbooks, Pascal instruction 1-1

Textfiles:
closing 6-30, 10-7, 10-12, 10-23, A-4
defined 6-29, 10-8
opening 6-30, 10-6 to 10-14
standard INPUT 4-9, 6-31, 10-5, 10-10, 10-11, 10-16, 10-17, 10-22,
standard OUTPUT 4-9, 6-31, 10-13, 10-14, 10-18, 10-22, 10-23, A-4

TRIM function (Prime extension) 6-14J to 6-14L, 11-5, A-4A

TRUNC function 6-4, 11-2

-TTY switch (Prime extension) 10-5, 10-11, 10-14, A-2

TYPE declaration 5-6, 5-7

Types (See Data types)

Unconditional GOTO statement 8-14, 8-15

Underscores and dollar signs in identifiers 4-8, A-3

UNPACK and PACK procedures (Prime restrictions) 9-12, A-5

UNSTR function (Prime extension) 6-14C, 6-14D, 6-14J to 6-14L, 11-5, A-4A

-UPCASE compiler option 2-13

User-defined identifiers 4-8

User-defined scalar data types 6-10 to 6-14

Value parameters 9-3

VAR declaration 5-7 to 5-9

Variable parameters 9-3, 9-4

Variables:
defined 4-7, 4-8, 5-7, 5-8
dynamic 6-31 to 6-33
external 9-17
file 6-27 to 6-31, 10-6 to 10-24
PRIMOS file 10-7, 10-8, 10-13
static 6-31

Variant fields 6-23 to 6-25

Variant records 6-23 to 6-25

WHILE statement 8-7, 8-8

WITH statement 6-22, 6-23, 8-16

WITH used with records 6-22, 6-23

Write parameters 10-18 to 10-22

WRITE procedure 10-18 to 10-21

WRITELN procedure 10-22

-XREF and -NOXREF compiler options 2-13

READER RESPONSE FORM

DOC4303-191 Pascal Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

   ___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

   Readability: ___hard to understand ___average ___very clear

   Technical level: ___too simple ___about right ___too technical

   Technical accuracy: ___poor ___average ___very good

   Examples: ___too many ___about right ___too few

   Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

   _____

   _____

   _____

4. What faults or errors gave you problems? _____

   _____

   _____

   _____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760