Prime Computer, Inc.

PRIME

Pascal
Rev. 19.2

```
X := LENGTH (STRING1);
REWRITE (OUTPUT, 'FILESTOR
FOR I := 1 TO X DO
 BEGIN
  STRING2 := SUBSTR (STRIN
  WRITELN (STRING2);
  WRITELN;
  N := N + 1
 END;
CLOSE (OUTPUT);
END.
```

The Programmer's Companion

FDR7095-192

PASCAL PROGRAMMER'S COMPANION

Revision 19.2
FDR7095–192

by

Stephen E. Alley

This document reflects the software
as of Master Disk Revision 19.2.

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

The Programmer's Companion is a series of pocket-size, quick-reference guides to Prime software products.

Published by Prime Computer, Inc.
Technical Publications Department
500 Old Connecticut Path
Framingham, Massachusetts 01701

Printing history: February 1984, First Printing

Credits

Design	Susan Windheim
Editing	Irene Rubin
Graphic Support	Marcella Gallardo Leo Maldonado
Technical Support	A. Paul Cioto
Data Entry	Hope Eldredge

TABLE OF CONTENTS

Note

Documents you will need with the **Pascal Programmer's Companion** include the **Pascal Reference Guide** and the **Prime User's Guide.** Secondary documentation includes the **New User's Guide to EDITOR and RUNOFF,** the **EMACS Primer,** the **EMACS Reference Guide,** the **SEG and LOAD Reference Guide,** the **Source Level Debugger User's Guide,** and the **Guide to Prime User Documents.** Use the following address to order Prime documentation:

Software Distribution
Prime Computer, Inc.
1 New York Ave.
Framingham, MA 01701
(617) 879-2960 x2053

PRIME DOCUMENTATION CONVENTIONS

The following conventions are used in command formats, in statement formats, in option formats, and in examples throughout this document. Pascal code may always be entered in either uppercase or lowercase.

Convention	*Explanation*
UPPERCASE	In command formats, words in uppercase indicate the actual names of commands, statements, and keywords. However, these words can appear in either uppercase or lowercase in the actual Pascal program. All program examples appear in uppercase for consistency.
lowercase	In command formats, words in lowercase indicate items for which the user must substitute a suitable value.
Brackets [ ]	Brackets enclose a list of one or more optional items. Choose none, one, or more of these items. When used in ARRAY and STRING declarations and in SET examples, brackets must be entered exactly as shown.
Ellipsis . . .	An ellipsis indicates that the preceding item may be repeated.

Parentheses ( )	In command, function, or statement formats, parentheses must be entered exactly as shown.
Hyphen –	Whenever a hyphen appears in a compiler option or in a compiler switch, it must be entered exactly as shown.
Vertical slash |	In command or statement formats, vertical slashes indicate a choice of one item or another.
Rust color	In examples of computer-user dialog, user input appears in rust color and system output does not. Command and compiler option abbreviations also appear in rust.
Underlining	Underlining indicates the default compiler options and switches.

COMPILING PROGRAMS

Prime supports two file naming conventions. Only the suffix filename convention, used for Rev. 18 and higher, will be described here and used in the following examples. (For information on the prefix method, see the **Pascal Reference Guide**.) Source files should be identified with a .PASCAL suffix. Thus, a program called TEST should be named TEST.PASCAL.

The command:

```
PASCAL TEST
```

will compile the example program TEST.PASCAL. For more information on the Pascal compiler see **THE PASCAL COMPILER** section. After compilation, two new files, besides the source file, are created and put into the directory to which you are attached. For example, after compilation of the TEST.PASCAL program these three files may exist:

TEST.PASCAL
Source program created and named by the user

TEST.BIN
Object file created upon compilation

TEST.LIST
Source listing file created upon compilation if –LISTING option was specified

After you have loaded your program (discussed in the next section), you should have this fourth file (the executable runfile) placed in your directory:

TEST.SEG
Executable runfile

LOADING PROGRAMS

The PRIMOS SEG utility loads and executes all Pascal programs. To invoke the Loader, type:

SEG –LOAD

The main program, the Pascal library, and the standard system libraries must all be loaded. External subprograms and/or other Prime libraries should also be loaded when needed. The QUIT or EXECUTE command returns control to PRIMOS level. Here is an example of loading the compiled Pascal program TEST.PASCAL:

`OK, SEG –LOAD`	Invoke the SEG Loader
`[ SEG rev x.x ]`	Loader returns with compiler revision level
`$ LOAD TEST`	Load the main program object file
`$ LIBRARY PASLIB`	Load the Pascal library
`$ LIBRARY`	Load the standard system libraries
`LOAD COMPLETE`	Loader signals load is complete
`$ QUIT`	Save executable file and return to PRIMOS
`OK,`	

When the loading process is finished an executable file, identified with a .SEG suffix, is created and put into the directory to which you are attached. In the above example the executable file would be:

TEST.SEG

EXECUTING PROGRAMS

After loading the program and returning to PRIMOS command level, the program may be executed with the SEG command. Here is an execution of the example program TEST.SEG:

SEG TEST

where TEST is assumed by the compiler to be the name of the program's executable runfile.

Execution During the Loading Process

Instead of typing the QUIT command immediately after the LOAD COMPLETE message during the loading process, the EXECUTE command may be entered to execute the program immediately. For example:

```
LOAD COMPLETE
$ EXECUTE
```

THE PASCAL COMPILER

The Pascal compiler is invoked by the command:

PASCAL filename [–option 1 . . . –option n]

The **filename** is the name of the source program to be compiled. The **filename** may be a pathname as well as a filename. The **options** are the names of various compiler functions. Multiple options are separated by spaces.

Pascal Compiler Options

Underlined options are the defaults in the following listing.

▶ –BIG / –NOBIG

Controls the production of boundary-spanning code for references to ARRAY or RECORD variable parameters.

▶ –BINARY [argument]

Controls the generation of an object (binary) file. The **argument** may be:

pathname	Object code is written to the file pathname.
YES	Object code is written to the program filename. (This is the default.)
NO	No object file is created.

▶ –DEBUG / –NODEBUG

Controls the generation of code for Prime's Source Level Debugger.

▶ –ERRTTY / –NOERRTTY

Controls the printing of error messages at the terminal.

▶ –EXPLIST / –NOEXPLIST

Controls the printing of pseudo-assembly code at the source listing.

▶ –EXTERNAL / –NOEXTERNAL

Controls the creation of an object file that can be linked to from other procedures and functions. Either this option, –EXTERNAL, or the $E+ compiler switch must be present when external subprograms are used.

▶ –FRN / –NOFRN

Controls the generation of floating-point round instructions.

▶ –INPUT filename

Obsolete. Identical to the compiler command **PASCAL filename.**

▶ –LISTING [argument]

Controls the creation of the source listing file. The **argument** may be:

pathname	Listing is written to the file pathname.
YES	Listing is written to the program filename.
TTY	Listing is printed at the terminal.
SPOOL	Listing is spooled directly to the line printer.
NO	No listing file is generated. (This is the default.)

▶ –MAP / NO_MAP

Controls the generation of a map of variables and their locations in memory that accompanies the listing of the program.

▶ –OFFSET / –NOOFFSET

Controls the creation of an offset map, which gives the hexadecimal offset in the object file of the first machine instruction generated for that statement.

▶ –OPTIMIZE / –OPT1 / –NOOPT1 / –OPT3 –NOOPT3 / –NOOPTIMIZE

Controls the optimization phase of the compiler.

–OPT1	Optimizes less code but compiles faster than –OPTIMIZE. (–NOOPT1 is the default.)
–OPT3	Optimizes more code but compiles more slowly than –OPTIMIZE. (–NOOPT3 is the default.)
–NOOPTIMIZE	No optimization occurs. Compiles faster but executes more slowly than –OPTIMIZE.

▶ –PRODUCTION / –NOPRODUCTION

Controls the generation of alternative option-controlling code for the Source Level Debugger.

▶ –RANGE / –NORANGE

Controls checking for out-of-bounds values of array subscripts and character substring indexes.

▶ –SILENT / –NOSILENT

Controls the suppression of severity 1 error messages.

▶ –SOURCE filename

Obsolete. Identical to the compiler command **PASCAL filename.**

▶ –STANDARD / –NOSTANDARD

Controls checking of code syntax. If syntax is non-ANSI standard Pascal, the –STANDARD option causes a severity 1 error message to be generated.

▶ –STATISTICS / –NOSTATISTICS

Controls the listing of compilation statistics for disk usage, seconds elapsed, space used, I/O time used, and CPU time used.

▶ –UPCASE

Causes the compiler to map lowercase variables to uppercase.

▶ –XREF / –NOXREF

Controls the cross-reference listing for all variables. This listing gives the number of every line that references the variable.

▶ –64V / –32I

Determines the addressing mode of the object code.

Pascal Compiler Switches

Compiler switches specified within the source program also control some compiler functions. Switches must always be enclosed within comment delimiters (* *) or { } or /* */. A dollar sign ($) must be the first character. Underlining indicates the default. The available switches are:

▶ {$A+} / {$A–}

Controls the generation of code used to perform array bounds checking at runtime. A– suppresses the generation. A+ resumes it.

▶ {$L+} / {$L–}

Controls the printing of source lines to the listing file at compile time. L– suppresses the printing. L+ resumes it.

▶ {$E+} / {$E–}

Controls the definition of globally defined procedures, functions, and variables. E+ allows procedures and functions to be compiled separately. E+ and E– must also surround variables defined in the main program that are used in external subprograms. (See the **Pascal Reference Guide**.)

▶ {$P+} / {$P–}

Controls page breaks in the listing file. P+ causes printing to skip to the top of the next page.

Compiler Error Messages

The general format of an error message is:

line-number line-of-code

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

Elements of the error message format are:

line-number	The number of the line where the error occurred
line-of-code	The erroneous line of the Pascal program
xxx	The error code number
y	The severity code number
explanation	Description of the error and possible remedies
'filename'	The name of the %INCLUDE file

The severity code definitions are:

Severity	*Description*
1	Warning (will not prevent execution)
2	Error the compiler has attempted to correct (will not prevent execution)
3	Uncorrected error (prevents successful compilation)
4	Error that immediately halts compilation

LEGAL CHARACTER SET

Any ASCII character may appear in Pascal character data and I/O files. In program source statements, the legal characters are:

- The 26 uppercase letters A–Z
- The 26 lowercase letters a–z
- The 10 digits 0–9
- The special characters + – * / = < > [ ] . , : ; ^ ( ) { } ' & ! % $ _
- Blanks or spaces

IDENTIFIERS

Identifiers may be either user-defined or standard Pascal identifiers. Standard Pascal identifiers are listed in the following section. User-defined identifiers must conform to the following restrictions:

- They may not be keywords.
- They must begin with a dollar sign ($) or a letter.
- They can be composed of any combination of letters, digits, underscores, and dollar signs.
- They may not be more than 32 characters long (Prime restriction).

KEYWORDS AND STANDARD IDENTIFIERS

Keywords are special names with fixed meanings that *cannot* be redefined. **Standard identifiers** are special names with predefined meanings that *can* be redefined for another purpose. However, a redefined standard identifier cannot be used for its original purpose within the scope of the redefinition. Prime Pascal keywords and standard identifiers are listed in alphabetical order on the following page.

ABS
AND
ARCTAN *
ARRAY
BEGIN
BOOLEAN *
CASE
CHAR *
CHR *
CLOSE * #
CONST
COS *
DELETE * #
DISPOSE * #
DIV
DO
DOWNTO
ELSE
END
EOF *
EOLN *
EXP *
EXTERN * #
FILE
FOR
FORWARD *
FUNCTION
GET *
GOTO
IF
IN
INDEX * #
INPUT *
INSERT * #
INTEGER *
LABEL
LENGTH * #
LN *
LONGINTEGER * #
LONGREAL * #
LTRIM * #
MOD
NEW *
NIL
NOT
ODD *
OF
OR
ORD *
OTHERWISE #
OUTPUT *
PACKED
PAGE *
PRED *
PROCEDURE
PROGRAM
PUT *
READ *
READLN *
REAL *
RECORD
REPEAT
RESET *
REWRITE *
ROUND *
SET
SIN *
SQR *
SQRT *
STR * #
STRING * #
SUBSTR * #
SUCC *
TEXT *
THEN
TO
TRIM * #
TRUNC *
TYPE
UNSTR * #
UNTIL
VAR
WHILE
WITH
WRITE *
WRITELN *

* Identifier
\# Prime extension keyword or identifier

PASCAL PROGRAM STRUCTURE

The standard Pascal block structure consists of a program heading, a declaration part, and an executable part.

Program Heading

The program heading is optional (Prime extension). A program heading has the format:

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

The identifier is the name of the program and file-identifier-list is a list of files used by the program. The compiler only checks the program heading syntactically. (See the Pascal Reference Guide for information on files.)

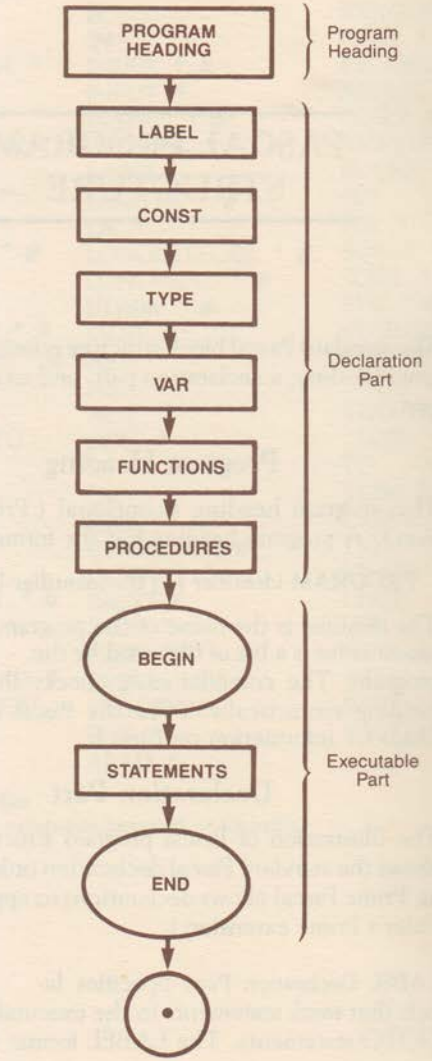
Declaration Part

The illustration of Pascal program structure above shows the standard Pascal declaration order. However, Prime Pascal allows declarations to appear in any order (Prime extension).

LABEL Declaration Part: Specifies labels that mark statements in the executable part for GOTO statements. The LABEL format is:

LABEL label [,label]. . .;

The label is an unsigned integer consisting of up to four digits.



PASCAL PROGRAM STRUCTURE

CONST Declaration Part: Specifies all constants and the names used to represent these constants in a program. The CONST format is:

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

The **identifier** represents a constant and it is used in place of the constant throughout the program. A **constant** is a fixed numeric or character string value.

TYPE Declaration Part: Specifies data types that are associated with a variable or variables in the VARIABLE declaration part. The TYPE format is:

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

The **type-identifier** is the name used to represent a specific data-type. A **data-type** can be predefined or user-defined.

VAR Declaration Part: Specifies all variables used in a program. The VAR format is:

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

The **identifier** is the name of the variable associated with some data-type. A **data-type** may be a predefined, standard Pascal data-type or a type-identifier as defined in the preceding TYPE declaration part.

PROCEDURE and FUNCTION Declaration Parts: Specify every procedure and function used in a program. The format of a procedure declaration is:

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

The **identifier** is the name of the procedure. The **block** has the same general format as a program block except that there is no program heading. If a formal parameter is preceded by the keyword VAR, then it is a variable (pass-by-reference) parameter. Otherwise it is a value (pass-by-value) parameter.

Function declarations have the format:

FUNCTION identifier [(formal-parameter-list)] : result-data-type; block;

Functions are similar to procedures except that they produce a single output value, which is assigned to the function **identifier.** For this reason, the function identifier itself must be assigned a type, which is the **result-data-type.** The **block** has the same general format as the program block except that there is no program heading.

Executable Part

The executable part of a program contains a sequence of executable statements delimited by the keywords BEGIN and END. (See the section on STATEMENTS for more information.)

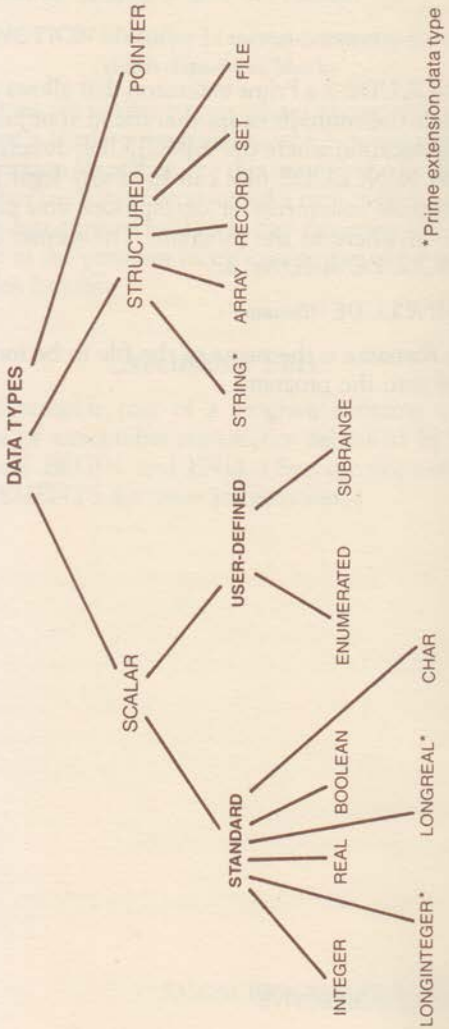
%INCLUDE DIRECTIVE

%INCLUDE is a Prime extension that allows you to include the contents of another file in your program at the location where the %INCLUDE directive appears. %INCLUDE files can hold any legal Pascal executable statements or declarations and can appear anywhere in the program. The format of the %INCLUDE directive is:

%INCLUDE 'filename';

The **filename** is the name of the file to be incorporated into the program.

DATA TYPES



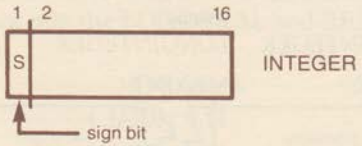
SCALAR DATA TYPES

Note

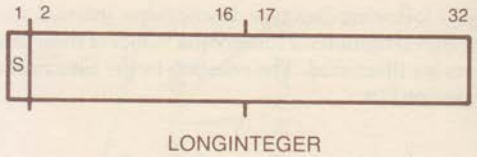
In the following data type descriptions, internal storage representations of some of the values of these data types are illustrated. The numbers in the illustrations represent bits.

Standard Scalar Data Types

The INTEGER Type: INTEGER comprises a subset of 16-bit whole numbers between –32768 and +32767, inclusive. Larger integers may be declared as LONGINTEGER (discussed later in this section) if you use a Rev. 19.1 or higher compiler. INTEGER constants must be written without a comma or decimal point. The sign is optional. MAXINT is a predefined Pascal INTEGER constant whose value is 32767. (See the table below for examples of valid integer constants.)



The LONGINTEGER Type: LONGINTEGER is a Prime extension that allows the use of 32-bit whole numbers, inclusive, within the range –2147483648 . . +2147483647. You must have a Rev. 19.1 or higher compiler to use LONGINTEGER. If you do not have a Rev. 19.1 or higher compiler, these integers must be declared as a subrange of type INTEGER within the above range. It is recommended that you do not mix the LONGINTEGER with the INTEGER type.



LONGINTEGER

VALID AND INVALID INTEGER REPRESENTATIONS

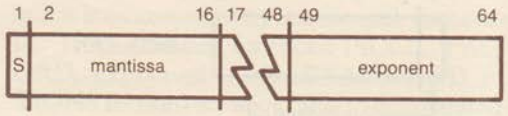
Valid INTEGER	*Invalid INTEGER*
23	4,568 (No comma allowed)
–100	40000 (LONGINTEGER)
MAXINT	32.00 (REAL)

Valid LONGINTEGER	*Invalid LONGINTEGER*
+40000	MAXINT
23	32.0 (REAL)
–100030	3143140000000 (Out of range)

The REAL Type: REAL is a subset of 32-bit real numbers with the range of approximately $-1 * 10^{38}$ to $+1 * 10^{38}$. Larger real numbers may be declared using LONGREAL as of Rev. 19.1. (See the LONGREAL discussion next.) Numbers of type REAL may be written in decimal notation or scientific notation. (See the table below for examples of valid REAL numbers.)

REAL

The LONGREAL Type: LONGREAL is a Prime extension as of Rev. 19.1. LONGREAL numbers are 64-bit numbers that can also be represented in either decimal or scientific notation. However, when longreals are represented in scientific notation, the letter D, instead of the letter E, signifies the exponent. Constants of more than six digits are assumed to be LONGREAL. It is recommended that you do not mix the LONGREAL and REAL types.



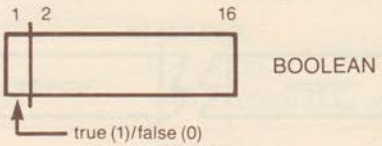
LONGREAL

VALID AND INVALID REAL REPRESENTATIONS

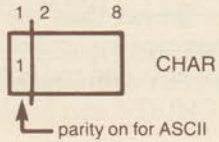
Valid REAL	*Invalid REAL*
–12.1	.12 (No digit to the left of the decimal point)
5E–8	5. (No digit to the right of the decimal point)
–7.0E+6	–8.2E–6.3 (Only whole number exponents allowed)

Valid LONGREAL	*Invalid LONGREAL*
+1.234567	1,234D+4 (No comma allowed)
2.1D01	16E44 (Must use the letter D to signify exponent)
5D+5	3.0D+4.0 (Only whole number exponents allowed)
1.1	

The BOOLEAN Type: BOOLEAN has two standard constant values: TRUE and FALSE. When these values are compared, TRUE is greater than FALSE. Three BOOLEAN functions (ODD, EOF, and EOLN) return BOOLEAN values. (See the **STANDARD FUNCTIONS** and **INPUT AND OUTPUT** sections for more information on these functions.)



The CHAR Type: CHAR consists of a group of characters that includes both printable and nonprintable characters: the ANSI, ASCII 7-bit character set. See the ASCII character set table at the back of this companion for a list of these characters and their corresponding ordinal values. Several built-in character-manipulating functions (CHR, ORD, PRED, SUCC) are explained in the **STANDARD FUNCTIONS** section of this companion.



User-defined Scalar Data Types

The Enumerated Type: Enumerated is a data type you create by sequentially listing an ordered group of values. To create an enumerated type, use the following type definition:

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

The **type-identifier** is the name of the new type, and **identifiers** are the values of this type. For example:

COLOR = (RED, YELLOW, GREEN, BLUE, PINK);

The ordinal number of the first (leftmost) value is 0 and it is incremented by one for each successive value. Three standard functions (SUCC, PRED, and ORD, explained in the **STANDARD FUNCTIONS** section) are applicable to enumerated types. The values of one enumerated type may not appear in any other enumerated type except when these values define a subrange.

The Subrange Type: Subrange is a data type you create based upon a specified range of any other already defined scalar data type, except types REAL and LONGREAL. The subrange type is defined by:

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

The **type-identifier** is the name of the subrange type, and **lower-bound** and **upper-bound** are values of the same standard scalar data type or previously defined enumerated type. The lower-bound value must not be greater than the upper-bound value. The subrange type comprises all those values between the lower-bound and upper-bound values, inclusive.

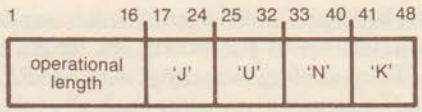
STRUCTURED DATA TYPES

The STRING Type

STRING is a Prime extension that allows you to manipulate character strings with a varying number of elements. A variable of type STRING can be declared like this:

```
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 known as the **maximum length** of the string. The default string length is 80. The number of characters contained in a string at any one time, which may be less than or equal to the maximum length specified by **n** is known as the **operational length** of the string.



STRING

STRING Operations and Functions

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

the operational length of the sending string is less than or equal to the maximum length of the receiving string, the entire string value along with the operational length is assigned. If the operational length of the sending string is greater than the maximum length of the receiving string, only the number of characters equal to the maximum length of the receiving string is assigned.

Assigning Arrays and Strings to Each Other: Two functions (STR and UNSTR), which are Prime extensions, allow arrays and strings to be assigned to one another. These are discussed in the **STRING FUNCTIONS** section of this companion.

Comparing Strings: Strings are compared from left to right according to ordinal values of the characters in the string. If the operational lengths of the strings are different, blanks will be assumed to follow the shorter string.

Concatenating Strings: Two strings may be concatenated into one string with the use of the concatenation operator (+), a Prime extension. The length of the newly formed string equals the sum of the lengths of the two concatenated strings.

Reading Strings: To read a string, enter any number of characters up to the maximum length of the string. 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.

Writing Strings: When writing a string, the default field width is the operational length of the string. If you specify the field width and it is greater than the operational length of the string, then the string is right-justified. If you specify the field width and it is

smaller than the operational length of the string, only the specified number of characters will be printed.

STRING Functions: The built-in Prime extension STRING functions are listed in the **STRING FUNCTIONS** section.

The ARRAY Type

The ARRAY type has the format:

```
VAR array-variable : ARRAY [ index-type ] OF
    data-type;
```

The **index-type** must be a scalar type other than REAL, LONGREAL, or LONGINTEGER. However, a subrange of type LONGINTEGER is acceptable as an index-type. The **data-type** of the array can be any data type.

Reading Arrays: On Rev. 19.1 or higher systems, Prime allows you to read an array of characters as one unit instead of reading one character at a time in a FOR loop. If variable A is declared as type ARRAY [1..10] OF CHAR, the statement READLN (A) reads all 10 characters at once.

Multidimensional Arrays: If the data type of an array is itself an array, the array is multidimensional. Multidimensional arrays are created according to the following type definition:

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

where **t1** and **t2** are index types and **base-type** is any data type.

The RECORD Type

A **record** is a structure consisting of a fixed number of elements, called fields, which may be of different data types. A record may be defined using the following type definition:

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

where **record-identifier** is the name of the entire record. The **field-identifiers** and their associated **data-types**, which can be any data type, are delimited by the keywords RECORD and END.

A **variant record** uses a variant part that is determined at execution time according to the value of the **tag-field**. The **fixed part** must precede the variant part. Here is an example of a RECORD type, STAFF, which uses both a fixed part and a variant part:

```
TYPE STAFF = RECORD
               LNAME, FNAME : ARRAY
                 [ 1..20 ] OF CHAR;
               AGE : 0..100;
               SEX : ( MALE, FEMALE );
               CASE MARRIED :
                 BOOLEAN OF
                 TRUE : ( SPOUSE_NAME
                   : ARRAY [ 1..20]
                   OF CHAR;
                   SPOUSE_AGE : 0..100 );
                 FALSE : ( )
             END;
```


MARRIED, the tag-field, should be assigned a value (TRUE or FALSE) before this record is used. If MARRIED is true, two additional tag-fields will exist — SPOUSE_NAME and SPOUSE_AGE.

You access a particular record element this way:

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

The WITH statement makes it easy to access several records at one time. Its format is:

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

Record elements within **statement** are referred to by **field-identifiers** only.

The SET Type

A **set** is a collection of elements (up to 256) of the same data type, termed the base-type. A **base-type** can be any scalar data type except REAL and LONGREAL. Use the following type definition to create a SET type:

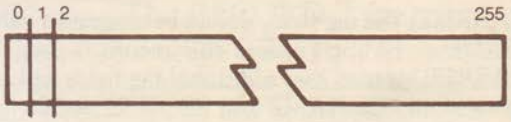
```
TYPE type-identifier = SET OF base-type;
```

The **type-identifier** is the name of the SET type. For example:

```
TYPE LETTERS = SET OF 'A'..'Z';
```

Variables of a SET type such as LETTERS are themselves sets whose members are chosen from the base-type. Set members are always enclosed in brackets [ ]. If VOWELS, EMPTY, and LIST are variables of type LETTERS, they can be assigned values this way:

```
VOWELS := ['A','E','I','O','U'];   { Arbitrary order }
EMPTY := [ ] ;                     { Empty set — no
                                     members }
LIST := ['F'..'P'];                { Consecutive
                                     values }
```



SET

Three SET operators and five SET relational operators are explained in the **OPERATORS** section.

The FILE Type

A **file** is a collection or receptacle of data values that is external to the program. The format of the file definition is:

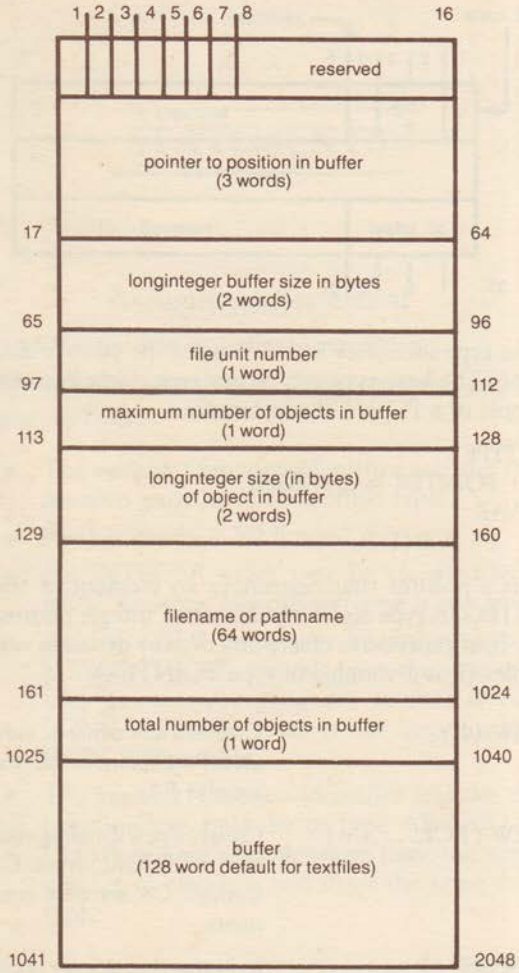
TYPE type-identifier = FILE OF base-type;

base-type must not be a FILE type or a structured type with a FILE component. Variables of type FILE must be used to open (RESET, REWRITE), close (CLOSE), and read and write (READ, READLN, WRITE, WRITELN) data files. For more information, see the **INPUT AND OUTPUT** section. FILE type data is stored and represented in a **file control block.**

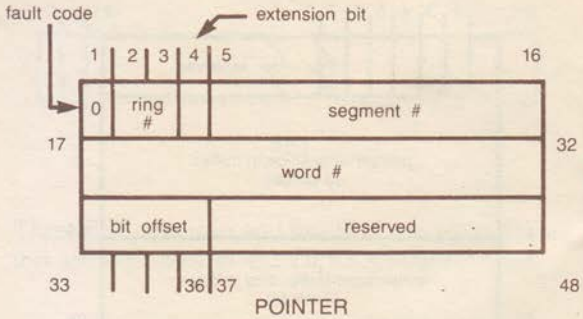
The POINTER Type

A **pointer** is a type of variable that references or points to a storage location in memory. It is the address of a dynamic variable, or a variable that is created and destroyed during execution. POINTER types are declared with this format:

TYPE type-identifier = ^base-type;



FILE CONTROL BLOCK



POINTER

The **type-identifier** is the name of the pointer data type. The **base-type** can be any type. Here is an example of a POINTER declaration:

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

P is a pointer that references an element of the INTEGER type and P^ is the actual integer pointed to. Four procedures create and destroy dynamic variables (P is a variable of type POINTER):

NEW (P)	Creates a new dynamic variable. P will point to the new variable P^.
NEW (P,C1,. . . ,Cn)	Creates a new dynamic variable of RECORD types. C1 through Cn are case constants.
DISPOSE (P)	Indicates that storage occupied by the variable P^ is destroyed. P is undefined.
DISPOSE (P, C1,. . . ,Cn)	Indicates that storage occupied by P^, of RECORD type, is destroyed. P is undefined.

STATEMENTS

Assignment Statement

variable | function-identifier := expression;

Here are some rules governing the use of the assignment statement:

- The **variable | function-identifier** and the **expression** must be of compatible types.
- Neither can be a FILE type.
- An INTEGER value may be assigned to a REAL variable, but the converse is not true.
- An INTEGER value may be assigned to a LONGINTEGER variable, but the converse may cause your program to fail if the LONGINTEGER value is too large for the INTEGER variable.
- The **variable | function-identifier** and the **expression** can both be of type ARRAY OF CHAR as long as both arrays have the same number of elements and share the same data type.

Procedure Statement

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

The **procedure-identifier** is the name of the called procedure. The **parameters** are used to pass values to and from the main program and the procedure.

Compound Statement

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

A compound statement can appear anywhere a single statement is allowed.

REPEAT Statement

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

REPEAT loops are executed at least once. Note that REPEAT loops do not need BEGIN and END delimiters for the loop body.

WHILE Statement

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

The **boolean-expression** is evaluated at the beginning of each cycle.

FOR Statement

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


The **control-variable** is increased by 1 up to or decreased by 1 down to the next value in the loop. The **initial-value** and **final-value** must be of a type compatible with the control-variable's type. The control-variable is undefined upon completion.

If Statement

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

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

Never put a semicolon immediately before ELSE. IF statements are *nested* when they appear as **statement-1** or **statement-2** or part of statement-1 or statement-2.

CASE Statement

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

If the value of **expression** matches any of the **case-constants**, then the corresponding **statement** is executed. The expression can be 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.

The expression must match one of the constant values unless you use Prime's OTHERWISE extension. OTHERWISE executes an alternative statement if no other statement corresponding to a case-constant-list has been selected. It must appear immediately before the keyword END. No colon separates the OTHERWISE clause from its statement.

GOTO Statement

GOTO label;

The **label** is an unsigned integer up to four digits long, which must be declared in the LABEL declaration part. (See the **PASCAL PROGRAM STRUCTURE** section.) The statement you wish to transfer control to must be prefixed with the integer label followed by a colon.

WITH Statement

The WITH statement makes it easy to access record fields. (See **The RECORD Type** in the **STRUCTURED DATA TYPES** section.)

OPERATORS

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 or REAL/LONGREAL	Same as operand

RELATIONAL OPERATORS

Operator	*Operation*	*Type of Operands*
=	Equality	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
>=	Greater or equal	

SET RELATIONAL OPERATORS

Operator	*Operation*	*Type of Operands*
=	Equality	SET
<>	Inequality	
<=	Set inclusion (is contained in)	SET
>=	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.

SET OPERATORS

Operator	*Operation*	*Result*
+	Union	All the members of both sets.
-	Difference	All the members of the first set that are not also members of the second set.
*	Intersection	All values that belong to both sets.

BOOLEAN OPERATORS

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 (Prime extension)

Operator	*Operation*
&	Performs BOOLEAN AND operation on two integers.
!	Performs BOOLEAN OR operation on two integers.

STRING CONCATENATION OPERATOR (Prime extension)

Operator	*Operation*
+	Concatenates two strings together. (See the **DATA TYPES** section.)

OPERATOR PRECEDENCE

1. Operations in parentheses — Highest precedence (done first)
2. Not, unary – and +
3. *, /, DIV, MOD, AND, &
4. +, –, OR, !
5. =, <>, <, >, <=, >=, IN — Lowest precedence (done last)

Operations at the same level of precedence are performed from left to right.

STANDARD FUNCTIONS

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*).

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 to an integer. 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. 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
ROUND (3.5) yields 4

Ordinal Functions

ORD (X)
Gives the corresponding ordinal value for integers, BOOLEAN values, subrange values, enumerated values, and any character in Prime's character set. (See the **ASCII CHARACTER SET** at the back of this companion.) The result is an INTEGER value. For example:

ORD ('F') yields 198

With enumerated and subrange types, the ordinal number for the first constant value is 0 and it is incremented by one for each successive value.

CHR (X)
Gives the corresponding character value of any integer between 0 and 255, inclusive. CHR is the opposite of ORD. X must be of type INTEGER. For example:

CHR (199) yields 'G'

SUCC (X)
Gives a value whose ordinal number is one greater than X. X can be of any scalar type (except REAL and LONGREAL) including any subrange or enumerated type. X and the result must be of the same type. For example:

SUCC (1) yields 2
SUCC ('A') yields 'B'

PRED (X)
Gives a value whose ordinal number is one less than X. X can be of any scalar type (except REAL and LONGREAL) including any subrange or enumerated type. X and the result must be of the same type. For example:

PRED (2) yields 1
PRED ('B') yields 'A'

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)
See the **INPUT AND OUTPUT** section.

EOLN (F)
See the **INPUT AND OUTPUT** section.

STRING FUNCTIONS

The following STRING functions are Prime extensions.

The STR Function: Converts an ARRAY OF CHAR value or a single character to a STRING value.

```
string-variable := STR ( array );
```

The UNSTR Function: Converts a STRING value to an ARRAY OF CHAR value. The result of the UNSTR function will have the same number of characters as the receiving array of characters. (Either blanks will be added or characters will be deleted.)

```
array-variable := UNSTR ( string );
```

The LENGTH Function: Takes a string as an argument and returns an integer that is the operational length of the string.

```
integer-variable := LENGTH ( string );
```


The INDEX Function: Takes two strings as arguments. It searches the first string to determine if it contains the second string. 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.

```
integer-variable := INDEX ( string-1, string-2 );
```

The SUBSTR Function: Takes three arguments—a string and two integers. It yields a substring of the first argument, which is a string, starting at the position of the second argument (integer) in that string. The third argument is the desired length of the substring.

```
string-variable := SUBSTR ( string, integer-1,
integer-2 );
```

The DELETE Function: Takes three arguments—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.

```
string-variable := DELETE ( string, integer-1,
integer-2 );
```

The INSERT 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.

```
string-variable := INSERT ( string-1, string-2,
integer );
```

The TRIM Function: Takes a string as an argument and returns the string after removing all trailing blanks.

```
string-variable := TRIM ( string );
```

The LTRIM Function: Takes a string as an argument and returns the string after removing all leading blanks.

```
string-variable := LTRIM ( string );
```


INPUT AND OUTPUT

Procedures READ, READLN, WRITE, WRITELN

The READ Procedure: READ reads input data values from input files or from the terminal and assigns these values, sequentially, to the variables in the READ parameter list. It has the format:

```
READ ( [ file, ] variable-1 [ ,variable-2 ] . . . );
```

The **file** is a variable that identifies the input data file. Several variables may be assigned values in one READ statement. Numeric input data values must be separated by spaces and/or commas, but input data values of type CHAR must not be separated at all.

The READLN Procedure: READLN is similar to the READ statement except that the READLN statement skips to the beginning of the next line of input after reading input data into the variable or variables. A READLN statement without an argument skips over the current line of input and moves the file pointer to the beginning of the next line. A READLN statement has the format:

```
READLN [ ( [ file | variable-1 ] [ ,variable-2 ] . . . ) ];
```

READLN must only be applied to files that have been declared FILE OF CHAR or TEXT.

The WRITE Procedure: WRITE copies the values denoted by the parameters into an output textfile or to the terminal. It has the format:

WRITE ([file,] write-parameter -1
[,write-parameter-2] . . .);

The **file** is a variable that identifies the output data file. The **write-parameter** can be either a character string enclosed in single quotes or an expression, in which case the value of the expression is written to the output data file. Two consecutive apostrophes will produce a single quote in a character string.

The WRITELN Procedure: WRITELN is similar to the WRITE statement except that WRITELN skips to the next line after writing the output values. A WRITELN statement without an argument functions exactly as a carriage return. WRITELN has this format:

WRITELN [(file | write-parameter -1]
[,write-parameter -2] . . .)];

WRITELNs must only be applied to files of type FILE OF CHAR or TEXT. The **write-parameter** is an expression, including character strings, whose output format may be specified with field-width parameters.

The Field Width: Field width specifies the number of character positions allocated for an expression's value in WRITE and WRITELN statements. Its format is:

expression [: total-width [: frac-digits]]

The **total-width** specifies the total number of character positions allocated for the expression. The **frac-digits** value applies only to expressions of type REAL and LONGREAL and it specifies the number of digits to the right of the decimal point to be printed. Several possibilities regarding field width specifications follow:

- If the total-width specified is greater than the actual width of the expression, the expression is right-justified.
- If the actual field width of the expression is larger than the total-width specified, Pascal automatically extends the specified total-width to a sufficient size.
- If the total-width is omitted, a default width is assumed. (See the **Pascal Reference Guide** for default width information.)
- If no frac-digits are specified for real numbers, the real numbers are written in floating-point (scientific) form.
- Positive REAL and LONGREAL numbers are always preceded by a blank.

Input and Output at the Terminal

If no input data file is specified in a READ or READLN statement and the standard textfile INPUT (discussed later in this section) has not been used to open an input file in a RESET statement, you can input data at the terminal.

If no output data file is specified in a WRITE or WRITELN statement and the standard textfile OUTPUT (discussed later in this section) has not been used to open an output file in a REWRITE statement, you can output data at the terminal.

Using Prime's Erase and Kill Characters: Two special characters, the erase and kill characters, provided by PRIMOS, allow you to correct mistakes when you are entering data. The default erase character is the double quote ("), and the default kill character is the question mark (?). However, these characters may have been changed to other characters on your system. These characters may be used while inputting data at the terminal with the –INTERACTIVE switch in the RESET statement in your program. –INTERACTIVE must be enclosed in single quotes and can only be used with the stan-

dard Pascal textfile identifier INPUT. The –INTERACTIVE switch can only be used with READLN statements, not READ statements. The –INTERACTIVE switch may be turned off with the –TTY feature. This allows you to go back to inputting data from the terminal without the use of Prime's erase and kill characters. Here is an example program illustrating the use of the –INTERACTIVE and –TTY switches:

```
PROGRAM EXAMPLE;
VAR
  A, B, C, D : CHAR;
  X, Y : INTEGER;
BEGIN
  WRITELN ( 'ENTER 4 LETTERS:' );
  RESET ( INPUT, '–INTERACTIVE' );
  READLN ( A,B,C,D );      {Read with use of
                            –INTERACTIVE}
    RESET ( INPUT, '–TTY' );
  READLN ( X,Y );          {Read without use of
                            –INTERACTIVE}
    CLOSE ( INPUT )
END.
```

If you mistakenly typed PQRX during execution instead of PQRS for the first READLN statement in the above example, you could correct your mistake like this:

```
OK, SEG EXAMPLE
ENTER 4 LETTERS:
PQRX"S
```

Input and Output With PRIMOS Data Files

The RESET Procedure: RESET must be used to open an input file. The RESET statement must appear before data from the input file is used. The format of a RESET statement is:

RESET (file,'filename');

The **file** is a variable that identifies that file type of the input file, and **filename** is the actual name of the input file. The following program reads input data (integers) from a file called NUMBERLIST:

```
PROGRAM FILEXAMPLE;
VAR
  A : INTEGER;
  INFILE : FILE OF CHAR;
BEGIN
  RESET ( INFILE, 'NUMBERLIST' );
  READ ( INFILE, A );
  CLOSE ( INFILE )
END.
```

The name of the input file can be a pathname as well as a simple filename. A variable may also be used to represent an input file, in which case no quotes are placed around the variable in the RESET procedure. Notice that the variable INFILE is closed by the CLOSE procedure. (This is discussed later in this section.) Also notice that the variable INFILE is declared as FILE OF CHAR.

Note

You should always declare the file variable as FILE OF CHAR or TEXT regardless of the type of data values you use (such as INTEGER, BOOLEAN, REAL) if you create the input file using Prime's text editors. (See the **Pascal Reference Guide** to find out when, why, and how to declare a file variable other than FILE OF CHAR.)

The TEXT File Type: TEXT is identical to the file type FILE OF CHAR and may be used in its place at any time.

The Standard Textfile INPUT: INPUT is of type FILE OF CHAR and does not have to be declared in the variable declaration part of a Pascal program. The above program example could be rewritten using INPUT like this:

```
PROGRAM FILEXAMPLE;
VAR
  A : INTEGER;
BEGIN
  RESET ( INPUT, 'NUMBERLIST' );
  READ ( INPUT, A );
  CLOSE ( INPUT )
END.
```

If a file is not specified in a READ or READLN statement, the standard textfile INPUT is assumed. In the above example, READ (INPUT, A); can also be written as READ (A); because the standard textfile INPUT, which has been declared to be the file NUMBERLIST, is assumed. If INPUT had not been explicitly declared, the READ statement would have defaulted to the terminal.

Switching From Standard INPUT File to the Terminal: This can be done by using the –TTY switch in another RESET procedure. –TTY allows you to input at the terminal after inputting from a data file. Example:

```
VAR
  A, B : INTEGER;
BEGIN
  RESET ( INPUT, 'NUMBERLIST' );
  READLN ( A );      {Read from input data file
                      NUMBERLIST}
  RESET ( INPUT, '-TTY' );
  READLN ( B );      {Read from terminal}
  CLOSE ( INPUT )
END.
```


The REWRITE Procedure: REWRITE allows you to open an output data file and output data to it. Its format is:

REWRITE (file, 'filename');

The **file** is a variable that identifies the file type of the output file, and **filename** is the actual name of the output file. The filename can also be a pathname. A variable may also represent the filename, in which case no quotes are placed around the variable in the REWRITE procedure. You should not create a PRIMOS output file, **filename**, beforehand. It is created for you when your program executes. For example, if the variable OUTFILE is declared as FILE OF CHAR or TEXT, the statement:

```
REWRITE ( OUTFILE, 'OUTDATA' );
```

opens the file OUTDATA for output. If OUTDATA had not previously been created, the REWRITE procedure would have created this file upon execution. If you open an existing file for output, the output data are written over any code that was contained in the file before it was opened. Output files must be closed (explained later in this section) just like input files.

The Standard Textfile OUTPUT: OUTPUT is very similar to the standard textfile INPUT explained earlier in this section. The REWRITE format above could be rewritten as:

```
REWRITE ( OUTPUT, 'OUTDATA' );
```

No file variable needs to be declared. If no file is specified in a WRITE or WRITELN statement, the standard textfile OUTPUT is assumed.

Switching From Standard OUTPUT File to the Terminal: This can be done using the –TTY switch in a REWRITE procedure similar to the –TTY switch and the RESET procedure that works for the standard textfile INPUT. This statement:

```
REWRITE ( OUTPUT, '–TTY' );
```

causes output to be written to the terminal instead of being written out to an output file opened with the standard textfile OUTPUT.

Other I/O Procedures and Functions

The GET Procedure: GET is used to move the file pointer to the next element of a file. Its format is:

GET(file)

After advancing the file pointer to the next element, GET assigns the value of this element to the buffer variable file and adjusts the value of EOF and EOLN (discussed later in this section).

The PUT Procedure: PUT writes an output value into an output file. Its format is:

PUT(file)

PUT writes the value of the buffer variable file to the end of **file**. EOF (file) remains true.

The EOF Function: EOF is a BOOLEAN function that tests for an end-of-file condition of a file. Its format is:

EOF(file)

EOF is true if the file pointer has moved beyond the end of the file. Otherwise it is false. If **file** is omitted, EOF is applied to the standard textfile INPUT.

Note

The CONTROL-C character is the end-of-file marker for input that is read from the terminal. It cannot be used with the –INTERACTIVE switch on.

The EOLN Function: EOLN is a BOOLEAN function that tests for an end-of-line condition in a textfile. Its format is:

EOLN(file)

EOLN is true if the file pointer's position is at a line separator. If **file** is omitted, EOLN is applied to the standard textfile INPUT.

The PAGE Procedure: PAGE is an auxiliary procedure that generates a skip to the top of the next page before the next line of the output textfile **file** is written. Its format is:

PAGE(file)

If **file** is omitted, PAGE is applied to the standard textfile OUTPUT.

The CLOSE Procedure: CLOSE is a Prime extension used to close all input and output data files after these files have been opened. Its format is:

CLOSE(file)

EXTERNAL AND FORWARD PROCEDURES AND FUNCTIONS

External Procedures and Functions

External procedures and functions are independent, separately compiled subprograms, which may be written in Pascal or any other Prime language. To use external subprograms written in Pascal, these rules must be followed:

1. The external subprogram's heading followed by the word EXTERN must appear in the main program's declaration section.
2. Any external subprogram variables that are referenced by the main program must be declared in the main program's VAR declaration section with the {$E+} compiler switch above them and the {$E–} switch below them.
3. Any external variables described above must also be declared in the subprogram file *before* the subprogram heading and not in the variable declaration section.
4. The {$E+} compiler switch must be put at the top of every external subprogram file. Do not put a {$E–} switch at the bottom of the external file.

Here is an example of a Pascal program that uses an external subprogram:

```
PROGRAM MAIN;
VAR
    I, J : INTEGER;
    {$E+}
    ADDSUM : INTEGER;      {external variable}
    {$E-}                  {declared in main}
PROCEDURE SUB ( A,B : INTEGER ); EXTERN;
BEGIN
    •
    •
    •
END.
```

Here is the external subprogram SUB :

```
{$E+}
VAR
  ADDSUM : INTEGER;   {external variable must}
                      {also be declared here}
PROCEDURE SUB (A, B : INTEGER);
  BEGIN
    •
    •
    •
  END;      {no E- switch here}
```

You can use the –EXTERNAL option whenever you compile a subprogram file instead of the {$E+} switch. (See the **Pascal Reference Guide.**)

Compiling and Loading Subprograms: Remember to compile and load each external subprogram separately.

Forward Procedure and Function Declarations

The FORWARD declaration allows subprograms to call other subprograms that have not been fully declared. The heading of the undeclared subprogram followed by the word FORWARD is placed before the subprogram that calls it. The rest of the subprogram appears later. Example:

```
FUNCTION X (M, N : INTEGER) : INTEGER;
                                FORWARD;
PROCEDURE TEST (VAR A, B : INTEGER);
  BEGIN
    •
    • {function X is called here}
    •
  END;
FUNCTION X;
  VAR R : INTEGER;
  BEGIN
    •
    •
    •
  END;
```

PRIME EXTENSIONS

The following summarizes Prime extensions and restrictions to standard Pascal.

The LONGINTEGER data type
Allows use of 32-bit whole numbers.

The LONGREAL data type
Allows use of 64-bit real numbers.

The ARRAY OF CHAR enhancement
Can read an array of characters as one unit instead of one character at a time.

Comment delimiters /* */
Can be used in addition to standard delimiters.

The –INTERACTIVE switch for erase and kill characters
Allows use of "erase" and "kill" characters on data input at the terminal.

The –TTY switch
Used to switch from inputting and outputting with data files to inputting and outputting at the terminal. Also can be used to turn the –INTERACTIVE switch off.

Optional program heading
Program heading is optional.

Order of declarations
Declarations can appear in any order.

%INCLUDE files
Used to include the contents of one file within another file.

$ and _ in identifiers
Can be used in identifiers, but the underscore (_) cannot be the first character.

The & and ! INTEGER operators
Perform BOOLEAN AND and OR operations, respectively, on integers. These are not synonymous with the BOOLEAN operators AND and OR.

The OTHERWISE keyword
Used as an alternative condition within a CASE statement. Evaluates to true if no other conditions within the CASE statement have been met.

The EXTERN attribute
Must appear in the main program at the end of the declaration heading of any external, separately compiled subprogram.

The {$E} compiler switch
Allows external Pascal subprograms to be compiled separately. Also used to single out subprogram variables declared in a main program's declaration section.

The {$A} compiler switch
Performs array bounds checking at runtime.

The {$L} compiler switch
Controls the printing of source lines to the listing file.

The {$P} compiler switch
Controls page breaks in program listings.

The second parameter in RESET and REWRITE procedures
Must specify the name of the PRIMOS file that is to be opened for reading or writing.

The CLOSE procedure
Is used to close data files after they have been opened by a RESET or a REWRITE procedure.

The standard data files INPUT and OUTPUT
When used in a RESET or REWRITE procedure without the second parameter 'filename' will default to I/O to and from the terminal. If no file is specified in a READ or READLN statement, the standard textfile INPUT is assumed, whether INPUT is a file or the terminal. This also applies to WRITE, WRITELN, and the standard textfile OUTPUT.

The STRING data type
Allows character strings with a varying number of elements to be manipulated.

The STRING concatenation operator (+)
Concatenates two strings into one.

Built-in STRING functions
Used with the STRING data type. All are described in the **STRING FUNCTIONS** section of this companion.

The null string ('')
String that contains no characters.

PRIME RESTRICTIONS

The keyword PACKED
Not supported in Prime Pascal. Generates a severity 1 warning message.

The PACK and UNPACK procedures
Will generate a severity 3 error and cause your program to fail if used.

Identifier length
Identifiers are limited to 32 significant characters. Generates a severity 1 error message if longer identifiers are used.

INTERFACING PASCAL TO OTHER LANGUAGES

The following table offers guidelines for interfacing Pascal data types with compatible data types of other Prime languages.

COMPATIBLE DATA TYPES

Generic Unit/PMA	BASIC/VM	COBOL	FORTRAN IV
1 bit	—	—	—
16 bits (one word)	INT	COMP	INTEGER INTEGER * 2 LOGICAL
32 bits (two words)	INT * 4	—	INTEGER * 4
64 bits (four words)	—	—	—
32-bit Float single precision	REAL	—	REAL REAL * 4
64-bit Float double precision	REAL * 8	—	REAL * 8
128-bit Float quad precision	—	—	—
Byte string (Max. 32767)	INT	DISPLAY (5) PIC A(n) PIC 9(n) PIC X(n)	INTEGER
Varying character string	—	*	*
48-bits (three words)	—	—	—
256	—	—	—

—Not available.
* See **Subroutines Reference Guide.**

COMPATIBLE DATA TYPES (Continued)

FORTRAN 77	Pascal	PL/I Subset G
—	—	Bit Bit (1)
INTEGER * 2 LOGICAL * 2	INTEGER BOOLEAN ENUMERATED	Fixed Bin Fixed Bin (15)
INTEGER INTEGER * 4 LOGICAL LOGICAL * 4	LONGINTEGER	Fixed Bin (31)
—	—	—
REAL REAL * 4	REAL	Float Binary Float Bin (23)
REAL * 8	LONGREAL	Float Bin (47)
REAL * 16	—	—
CHARACTER * n	CHAR ARRAY [1..n] OF CHAR	Char(n)
*	STRING[n]	Char(n) Varying
—	<type>	Pointer
—	SET	Bit (256)

ASCII CHARACTER SET

ASCII CHARACTER SET (NONPRINTING)
(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

Octal Value	ASCII Char	Comments/Prime Usage	Control Char
216	SO	Shift out (to alternate character set)	^N
217	SI	Shift in (to standard character set)	^O
220	DLE	Data Link escape (2)	^P
221	DC1	Device control 1 (3)	^Q
222	DC2	Device control 2	^R
223	DC3	Device control 3 (4)	^S
224	DC4	Device control 4	^T
225	NAK	Negative acknowledgement (communications)	^U
226	SYN	Synchronization (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

(1) Generally, CR is interpreted as NL (new line) at the terminal. In Pascal, however, CR (or LF) always returns from text files as a blank.

(2) At terminal: abort (quit) program, return to PRIMOS. Within a file: relative copy; next byte specifies number of bytes to copy from corresponding position of preceding line.

(3) Next byte specifies number of spaces to insert.

(4) Next byte specifies number of lines to insert.

ASCII CHARACTER SET (PRINTING) (CONFORMS TO ANSI X3.4-1968— 1963 VARIANCES ARE NOTED)

OCTAL Value	ASCII Character	OCTAL Value	ASCII Character
240	space (1)	300	@
241	!	301	A
242	" (2)	302	B
243	# (3)	303	C
244	$	304	D
245	%	305	E
246	&	306	F
247	' (4)	307	G
250	(	310	H
251	)	311	I
252	*	312	J
253	+	313	K
254	, (5)	314	L
255	–	315	M
256	.	316	N
257	/	317	O
260	0	320	P
261	1	321	Q
262	2	322	R
263	3	323	S
264	4	324	T
265	5	325	U
266	6	326	V
267	7	327	W
270	8	330	X
271	9	331	Y
272	:	332	Z
273	;	333	[
274	<	334	/
275	=	335	]
276	>	336	^(7)
277	? (6)	337	_ (8)

OCTAL Value	ASCII Character	OCTAL Value	ASCII Character
340	' (9)	360	p
341	a	361	q
342	b	362	r
343	c	363	s
344	d	364	t
345	e	365	u
346	f	366	v
347	g	367	w
350	h	370	x
351	i	371	y
352	j	372	z
353	k	373	{
354	l	374	|
355	m	375	}
356	n	376	~(10)
357	o	377	DEL (11)

Notes

(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: ↑ (up-arrow)
(8) Underscore "_"; 1963 standard: ← (back-arrow)
(9) Grave
(10) 1963 standard ESC
(11) Rubout