Pascal/MT+™
Language
Reference Manual

Digital Research
P.O. Box 579
801 Lighthouse Avenue
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001

The Pascal/MT+ Language Reference Manual was prepared using the Digital Research TEX Text Formatter, and printed in the United States of America.

****
* First Edition: February 1983 *
****

Foreword

The Pascal/MT+™ language is a full implementation of standard Pascal as set forth in the International Standards Organization (ISO) standard DPS/7185. The Pascal/MT+ language also has several additions to standard Pascal. These additions make Pascal/MT+ more suitable for commercial programming, and increase its power to develop high-quality, efficiently maintainable software. The additions fall into four categories:

- enhanced I/O
- additional data types
- access to the run-time system
- modules and overlays

Pascal/MT+ is useful for both data processing applications and for real-time control applications.

The Pascal/MT+ system, which includes a compiler, linker, and programming tools, is implemented on a variety of operating systems and microprocessors. Because the language is consistent among the various implementations, Pascal/MT+ programs are easily transportable between target processors and operating systems. The Pascal/MT+ system can also generate software for use in a ROM-based environment, to operate with or without an operating system.

This manual describes the Pascal/MT+ language with emphasis o those features that are unique to Pascal/MT+. Information in this manual covers all language-related topics independent of the implementation.

Information about the compiler, linker, the Pascal/MT programming tools, and topics related to the operating system are contained in the version of the Pascal/MT+ Language Programmer' Guide pertinent to your specific implementation.

This manual assumes you are already familiar with the Pascal language in general. If you are not familiar with Pascal, refer t Appendix C for a bibliography of textbooks.

This manual uses Backus-Naur Form (BNF) notation to formally describe the syntax of Pascal statements. If you are not familiar with BNF notation, see Appendix B.

Table of Contents

Table of Contents
(continued)

Table of Contents
(continued)

Table of Contents
(continued)

Appendixes

Figures, Tables and Listings

Figures

Tables

Listings

Section 1
Pascal/MT+ Programs

1.1 Program Structure

Pascal/MT+ is a block-structured language. That is, you group one or more statements into logically related units called blocks. Every block has a heading, an optional declaration and definition section, and a set of statements. In every Pascal/MT+ program, the outermost block is the main program.

You can nest blocks inside your program. That is, you can put one block inside another block, but not overlap them. Inside blocks, you can also nest procedures and functions (see Section 6).
Figure 1-1 illustrates the typical block-structure of Pascal/MT+.

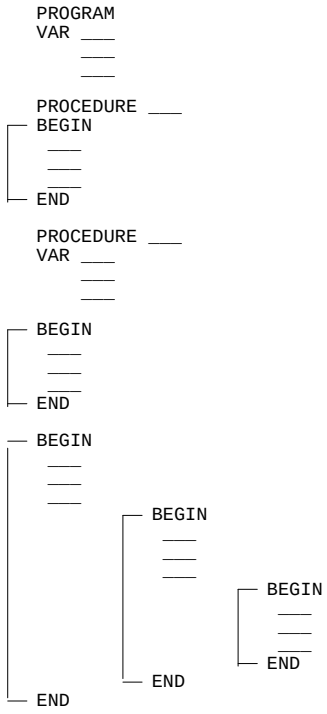


Figure 1-1. Block-structure in Pascal/MT+

Listing 1-1 shows a small Pascal/MT+ program containing a nested block.

```
PROGRAM FIRST_1;

CONST
  LIMIT   = 10;
  MESSAGE = 'TESTING PASCAL/MT+';

VAR
  NAME : STRING;


PROCEDURE RESPOND (ST : STRING);

VAR
  I : INTEGER;

BEGIN
  FOR I := 1 TO LIMIT DO
    BEGIN
      WRITELN (ST);
      ST := CONCAT  (‘ ‘, ST)     (* SHIFTS NAME TO RIGHT *)
    END
END;


BEGIN
  WRITELN (MESSAGE);
  WRITELN ('WHAT IS YOUR NAME?');
  READLN (NAME);
  RESPOND (NAME);
  WRITELN (‘FINISHED ‘, MESSAGE)
END.
```

Listing 1-1. Simple Pascal/MT+ Program

1.1.1 Program Heading

A program heading has the following form:

```
PROGRAM <program name> {(<Program parameters>)};
```

The <program name> has no significance inside the program, but you should not use the name for any other data item in the program. The optional <program parameters> have no special meaning in Pascal/MT+, as they do in some other versions of Pascal.

1.1.2 Declarations and Definitions

You must define an identifier before you use it in a program, unless the identifier is predefined by the language (see Appendix A). Listing 1-2 shows an example of the declaration and definition part of a program illustrating each of the major kinds of declarations as shown in the following list.

1) LABEL declarations
2) CONSTANT declarations
3) TYPE definitions
4) VAR declarations
5) PROCEDURE and FUNCTION definitions

Note that LABEL, CONSTANT, TYPE, and VAR declarations can be in any order, and there can be multiple occurrences of each type in a module. PROCEDURE and FUNCTION declarations must appear last, and there can be only one section of these per module.

Section 3 describes the various kinds of data type definitions.

```
LABEL
  34, 356, 755, 1000;


CONST
  TOP       = 100;
  BOTTOM  = -TOP;
  LIMIT   = 1.0E-16;
  MESSAGE   = 'THANK YOU FOR NOT SMOKING';


TYPE
  COLOR  = (RED, YELLOW, BLUE, GREEN, ORANGE);
  INDEX  = BOTTOM .. TOP;
  PERPT  = ^PERSON;
  PERSON = RECORD
             NAME,
             ADDRESS : STRING;
             PHONE   : STRING[8]
           END;

VAR
  COLR : COLOR;
  I, J : INTEGER;
  LIST : ARRAY [INDEX] OF PERPT;


PROCEDURE ECHO (ST : STRING);
BEGIN
  WRITELN (ST, ' ' ST)
END;
```

Listing 1-2. Declarations and Definitions

1.1.3 Statement Body

The words BEGIN and END surround the body of statements in a block, which can contain zero or more statements. If the block is the main program block, you must put a period after the word END. Within the statement body, separate each statement with a semicolon.

1.1.4 Modules

A module is a portion of a program that you compile separately, and then link to the main program. The general form of a module is the same as a program, except that a module does not have a main statement body. The only executable code in a module is contained in procedures and functions. The following example illustrates a simple single-procedure module.

```
MODULE SIMPLE;

PROCEDURE MARK (CALL_NUM : INTEGER);
BEGIN
  WRITELN ('IN MODULE SIMPLE, CALLED FROM: ', CALL_NUM)
END;

MODEND.
```

Notice that the word MODULE replaces the word PROGRAM and that the word MODEND replaces the main statement body.

Refer to the Pascal/MT+ Language Programmer's Guide for your implementation for more information about modules and modular programs.

1.2 Scope

Every identifier in a Pascal/MT+ program has a scope. The scope of an identifier is the set of all blocks where you can make a valid reference to the identifier. The normal scope of an identifier is anywhere inside its defining block, starting from its actual definition.

However, when a nested block redefines the same identifier, the outer variable is inaccessible from the inner block. When the same identifier has multiple definitions, the innermost definition is the one that applies.

This manual uses the terms global and local. The declarations at the outermost level in the program are the global declarations. Declarations in a block are local to that block. A variable is local to a block if its declaration is in that same block. Inside a nested block, a variable declared in a containing block is usable, but it is not local to that nested block. Within a contained block, a reference to a variable in a containing block is called an “up-level reference”.

Listing 1-3 shows a program containing nested blocks with multiple definitions for the same identifiers. The comments in the program explain which definitions apply at various points.

```
PROGRAM SHOWSCOPE;

VAR
  X, Y, Z : INTEGER;   (* X,Y,Z ARE GLOBAL *)

PROCEDURE PROC1;
VAR
BEGIN
  X := y / Z           (* Y & Z FROM MAIN BLOCK *)
END;

PROCEDURE PROC2;
VAR
  W : INTEGER;         (* W LOCAL TO PROC2 *)
  Y : STRING;          (* Y LOCAL TO PROC2 *)
BEGIN
  Y := 'ABCDEFG';
  W := X;              (* X FROM MAIN BLOCK *)
  Z := X DIV 3         (* X FROM MAIN BLOCK *)
END;

BEGIN
  Y := 35;             (* X, Y, & Z ARE ALL INTEGERS *)
  Z := 12;             (* IN THIS BLOCK *)
  PROC1;     (* CHANGES X *)
  PROC2;     (* CHANGES Z *)
  WRITELN (X, Y, Z)
END.
```

Listing 1-3. Example of Scope Rules

1.3 Comments

You can put a comment anywhere in a program that you can put a blank space; the compiler ignores comments. There are two ways to write a comment in a Pascal/MT+ program:

- Surround the comment with the characters { and }
- Surround the comment with the character pairs (* and *)

The compiler differentiates between the two sets of comment delimiters, so you can nest comments. You can use one set of delimiters for regular comments in your program, and use the other set of delimiters to comment out sections of code for debugging or development, as shown in the following program fragment.

```
PROCEDURE WALKTREE (TREE : TREEPT);

BEGIN
 WITH TREE^ DO
  BEGIN
    WALKTREE (LEFTREE); { PRE-ORDER WALK OF TREE }
    WRITELN (INFO.NAME);

    (* **** REMOVE THIS LINE FOR DIAGNOSTICS

    WRITELN ('**** IN WALKTREE ****');
    IF MARKED(NODE) THEN    { LOOK FOR LOOPS IN TREE }
      BEGIN
        WRITELN (‘  LINK ERROR IN TREE');
        TREEDUMP (TREE)  { WILL NOT RETURN }
      END
    ELSE
      MARK (NODE); { TREE OK SO FAR }

    ****** REMOVE THIS LINE FOR DIAGNOSTICS *)

    WALKTREE (RIGHTREE)
  END
END;
```

Listing 1-4. Example Program with Comments

End of Section 1

Section 2
Identifiers and Constants

This section describes Pascal/MT+ identifiers, and the rules for forming literal constants. It also describes how to define named constants.

2.1 Identifiers

A Pascal/MT+ identifier can represent a variable, a type, a constant, a procedure or function, or an entire program. The same rules apply to all Pascal/MT+ identifiers, regardless of what kind of objects they represent.

A Pascal/MT+ identifier can be any length, as long as it fits on one line. However, the compiler uses only the first eight characters to distinguish one identifier from another. Only the first seven characters are significant in external identifiers.

Identifiers can contain any combination of letters, digits, and underscores. They must begin with a letter, and they cannot contain any blank spaces. The compiler ignores underscores and typecase. For example,

```
A_b__C
```

is the same as

```
abc
```

You can also use an @ as the first character in an identifier, as long as you do not use the @ compiler option. You cannot use the @ inside an identifier. The compiler allows the @ character, so you can access the run-time routines whose name begins with @.

However, if you use the @ compiler option, then the compiler interprets the @ character as the standard pointer character, and does not allow the @ as part of an identifier.

The following are examples of valid Pascal/MT+ identifiers:

```
x
@CPMRD
file_name
LA225prefix
Thisfile
Thisfile_for_91803_zip_only
```

The last two examples are indistinguishable to the compiler.

The following are examples of invalid identifiers:

X!2	Contains an illegal character
123x	Begins with a digit
program	Reserved word
STY@HM	@ not first character
X 22	Contains a blank space

You cannot use reserved words, such as BEGIN and IF, as identifiers. However, you can use predefined identifiers such as WRITELN and BOOLEAN, to name any object in your program. Predefined identifiers are defined one level above the global level in your program, so changing the definition of a predefined identifier makes the old object inaccessible from within the scope of the new definition.

Appendix A lists the Pascal/MT+ reserved words and predefined identifiers. The Pascal MT+ Language Programmer's Guide for your implementation contains the list of the run-time entry-point names, as well as information about external identifiers.

Note: if you inadvertently use a run-time entry-point name as an external identifier, your program might not link properly.

2.2 Constants

You can express a constant as a literal value, or you can give the constant a name and then use the name anywhere you need that value. Pascal/MT+ constants can be strings, integers, real numbers, or scalar types.

2.2.1 Numeric Literals

A numeric literal can be a decimal integer, a hexadecimal integer, a long integer, or a real number. The form of the constant determines its type.

Note: long integers are not available with the 8-bit versions of Pascal/MT+.

An integer literal is any whole number in the range -32768 to 32767. An integer literal cannot have a decimal point or any commas. To write an integer in hexadecimal, start it with a $. The following are examples of valid integer literals:

```
-3456
$FF00
32767
$EFFF
```

A long-integer constant must start with a pound sign, #. For negative numbers, put the minus sign before the #. The following are examples of long-integer literals:

```
#6234343
#0
-#678988
```

A real-number literal can be either in fixed- or floating-point format. In fixed-point format, at least one digit must precede and follow the decimal point. The form for a floating-point literal is a number with or without a decimal point, followed by an E, followed by an optionally signed integer. Neither format can contain any blanks or commas. The following are examples of valid real-number literals:

```
64.78E-13
-65.3
-33.677E+10
```

In floating-point format, the E is interpreted as "times 10 to the power of.” For example,

```
6.3E5
```

is 6.3 times ten to the power of five (10^5), or 630000.

2.2.2 String Literals

A string literal can contain any number of printable characters, as long as the string fits on one line. You write a string literal by enclosing it in single apostrophes. Everything between the apostrophes, including blanks, is part of the string. Use two single apostrophes to represent one single apostrophe inside a string. Inside strings upper- and lower-case letters are distinct. The following are examples of valid string literals:

```
'*** INVALID EDIT COMMAND ***’

'Steve’’s Program'
```

If you need to define a string that is longer than you can fit on one line, or if you need to put control characters in a string, use the string functions described in Section 6.

2.2.3 Named Constants

A constant definition defines an identifier as a synonym for a constant value. You can use a named constant anywhere that you can use a literal. The following is an example of a constant definition section:

```
CONST
   message       =   'VERSION 3.3';
   size          =   100;
   limit         =   -size;
   esc           =   $lB;
   conv_fact     =   3.27E-3;
   null_str      =   '';
```

Notice that Pascal/MT+ allows the null string.

End of Section 2

Section 3
Variables and Data Types

This section describes the data types supported by Pascal/MT+. There are two general categories of data types: simple and structured. Simple data types, also called scalar types, have only one element per data item. Integers, characters, and pointers are examples of simple types.

Structured types contain more than one element within a data item. Records, strings, and arrays are examples of structured types.

This section does not discuss files; see Section 7 for information about files.

3.1 Type Definition

The compiler uses a type definition to determine how to allocate space for a variable. The type definition section of a block associates names with specific type definitions, as in the following example:

```
TYPE
   NUMBERS = ARRAY [l..10] OF INTEGER;
   STRPT   = ^STRING;
   LETTER = 'A' .. 'Z';
```

3.2 Variable Declaration

A variable declaration establishes the type of a variable, and determines its scope. You must declare all variables before you can use them in a program. The following is an example of a variable declaration section in a block.

```
VAR
   X, Y, Z     : INTEGER;
   NAMES       : LIST;
   NUM1        : 0..200;
   NUM2        : 0..200;
```

Notice in the example above how you can group more than one name with a particular type definition, and that you can use an explicit type definition instead of just a type name.

If the compiler is using strong type checking, you must declare variables with the same type name if you want the variables to be compatible. Strong type checking requires that compatible

variables have exactly the same type, not just the same internal structure. In the above example, NUM1 and NUM2 are not compatible under strong type checking. To make them compatible, you could use the declaration,

```
NUM1,NUM2     :   0..200;
```

See the programmer's guide for more information about how the compiler performs type checking.

Pascal/MT+ supports absolute variables. That is, you can force a variable to be stored at a specific location using an absolute variable declaration. See the Programmer's Guide for details.

Pascal/MT+ also supports external variables. That is, you can declare variables in one module and reference them in other modules.

3.3 Simple Types

Pascal/MT+ has several predefined simple data types, summarized in Table 3-1. All of the simple data types, except the reals, are ordinal types. An ordinal type is one in which each possible value is countable with integers. The ASCII character set is an example of an ordinal type.

You can define your own enumerated or subrange data types. An enumerated type is an ordinal type whose complete set of values you explicitly specify. A subrange type is a contiguous portion of some other ordinal type.

Table 3-1. Predefined Data Types

Data type	Size		Range
CHAR	1	8-bit-byte	0 to 255
BOOLEAN	1	8-bit-byte	true or false
INTEGER	2	8-bit-bytes	-32768 to 32767
LONGINT	4	8-bit-bytes	2^{32}-1 to 2^{-32}
BYTE	1	8-bit-byte	0 to 255
WORD	2	8-bit-bytes	0 to 65535
BCD REAL	10	8-bit-bytes	see Programmer's
FLOATING REAL	8	8-bit-bytes	Guide

Pascal/MT+ provides four "pseudo-functions" or type conversion operators to convert from one simple type to another. These pseudo-functions do not generate any code, but simply direct the compiler to treat the following 8- or 16-bit item as a different type. The four pseudo-functions are

- CHR(X) returns the character whose ASCII value is the specified expression.

- ORD(X) returns the ordinal value of the expression. The ordinal value of a character is its ASCII numeric representation.

- ODD(X) returns the BOOLEAN value TRUE if the expression is odd, otherwise it returns the BOOLEAN value FALSE.

- WORD(X) directs the compiler to treat the specified expression as a native machine word.

3.3.1 BOOLEAN

The BOOLEAN type has two values: TRUE and FALSE. The ordinal value of FALSE is 0, and the ordinal value of TRUE is 1.

A BOOLEAN variable uses one byte, even in a packed structure (see Section 3.4). Within the byte, only the least-significant bit matters in determining the value. If the bit is set, the value of the variable is TRUE, if not, the value is FALSE. However, logical operations use the whole byte.

3.3.2 CHAR

Variables of type CHAR use one byte. The internal representation of a character is the ASCII value of the character. The range for CHAR variables is CHR(O) to CHR(255).

To express a CHAR value in a program, enclose the character in single apostrophes if it is a printable character, or use the CHR pseudo-function. Use two single apostrophes to represent the single apostrophe character.

The following example program demonstrates the CHR and ORD pseudo-functions.

```
PROGRAM CHR_ORD;

VAR
  I, J : INTEGER;
  C, D : CHAR;
  BELL : CHAR;

BEGIN
  I := 7;
  C := '8';
  D := CHR(I + ORD('0'));  (* ASCII VALUE OF '0' IS 48 *)
  J := ORD(C) - ORD('0');
  BELL := CHR(7)
END.
```

3.3.3 INTEGER and LONGINT

INTEGER variables are 2 bytes long. Integers can range from -32768 to +32767. An integer literal in the range 0 to 255 takes up only one byte in the code.

LONGINT variables are 4 bytes long. The range for long integers is 2^{-32} to 2^{32}-1. You can write a LONGINT literal only in decimal; write it like a regular integer literal, but start the number with the # character. For example,

```
#6234343
```

You can define LONGINT subranges, but you cannot use them as indexes for arrays.

There are three functions for converting between the LONGINT and other data types:

```
FUNCTION SHORT(L: LONGINT): INTEGER;
FUNCTION LONG (S: SHORT  ): LONGINT;
FUNCTION XLONG(S: SHORT  ): LONGINT;
```

A short data type is any 8- or 16-bit type, such as CHAR, BOOLEAN, INTEGER, or WORD. The function LONG pads the short value with zeros. The function XLONG sign-extends the short value into the high-order word.

See your programmer's guide for specific information about the internal representation of the INTEGER and LONGINT data types.

Note: the LONGINT type is not available in the 8-bit versions of Pascal/MT+.

3.3.4 REAL

Pascal/MT+ handles real numbers in two ways to support different applications:

- BCD for business applications
- Binary floating point for scientific and engineering applications.

A command-line option tells the compiler which format to use.

The internal representation and range of real numbers depends on the processor. See your programmer's guide for details about the internal representation of real numbers.

The following are examples of real-number literals, as explained in Section 2.

```
212.3E-16
-22.454
2.0E+4
```

3.3.5 BYTE and WORD

The BYTE data type uses a single byte. It is compatible in expressions and assignment statements with the CHAR and INTEGER types. BYTE accepts any bit pattern and is useful for handling control characters, and performing character arithmetic.

The WORD data type uses a native machine word, except in the 8-bit implementation where it uses two bytes. All arithmetic and comparison operations on WORD expressions are unsigned, whereas operations using INTEGER are signed.

3.3.6 User-defined Ordinal Types

You can define two kinds of ordinal types: enumerated types and subranges.

An enumerated type is one in which you explicitly list each value in the type. The names for the values must be valid Pascal/MT+ identifiers. The following example shows some type definitions for enumerated types.

```
TYPE
   COLOR = (RED, YELLOW, BLUE, GREEN, ORANGE);
   SCORE = (LOST, TIED, WON);
   SKILL = (BEGINNER, NOVICE, ADVANCED, EXPERT, WIZARD);
```

The ordinal value of an enumerated-type constant is the same as its position in the type definition. The first constant has an ordinal value of 0. In the example above, YELLOW has an ordinal value of 1, and EXPERT has an ordinal value of 3.

A subrange is a set of values ranging between two specified values of some previously defined ordinal type. The following are examples of subrange definitions.

```
TYPE
   GOOD    = ADVANCED .. WIZARD;
   PRIMARY = RED .. BLUE;
   NUMERAL = '0' .. '9';
   INDEX   = 1 .. 100;
```

Both bounds in a subrange definition must be either literals or named constants of the same ordinal type. The left constant must have an ordinal value less than that of the right constant.

3.3.7 Pointers

A pointer is a variable whose value is the address of a dynamically allocated variable of some specific type. To define a pointer type, use the pointer character, ^, followed by a type name, as in the following examples.

```
TYPE
   INTPT    : ^INTEGER;
   LINK     : ^TREE_NODE;
   NAMEPTR  : ^STRING;
```

You can assign the value NIL to any type pointer to represent a null pointer.

To reference the object whose address a pointer contains, follow the pointer's name with the ^ character, as in the following examples.

```
NEWREC    :=  NEXT^;
NAME^     :=  'ALPHA FIVE';
EMPLOYEE^.AGE := 32;
```

If the compiler is using strong type checking, two pointers must be of the same type to be compatible. When the compiler is using weak type checking, all pointer types are compatible, allowing you to treat the same object as more than one data type.

Note: if you use the @ compiler command-line option, the compiler accepts the character @ as a substitute for the ^ character.

3.4 Structured Types

Structured types are a composite of other types. A simple-type variable only has one value, whereas a structure-type variable can be a collection of values of different types. Arrays, records, sets, and files are the major kinds of structured types. Section 7 discusses filetypes.

When determining the internal layout of a structured type, the compiler sometimes leaves gaps between elements, putting the elements at word boundaries to speed up access. If you want to sacrifice speed for space, you can use the reserved word PACKED. In the context of a structure type definition, the word PACKED causes the compiler to eliminate any wasted space.

3.4.1 Arrays

An array is a collection of a fixed number of elements of the same type. Arrays can have any type element, including other structured types. An array type definition has the general format:

```
ARRAY [<index type> {,<index type> }] OF <element type>
```

The <index type> can be any subrange type except LONGINT. You can either use the name for a subrange type, or specify the bounds explicitly. For the <element type>, you can either use a type name, or define the type right in the array definition. The following are examples of array type definitions.

```
TYPE
   LIST  = ARRAY [FIRST .. LAST] OF STRING;
   GRID1 = ARRAY [1 .. 20] OF ARRAY [1 .. 20] OF INTEGER;
   GRID2 = ARRAY [1 .. 20, 1 .. 20] OF INTEGER;
   TABLE = PACKED ARRAY [INDEX] OF PERPT;
```

Note that the definitions for GRID1 and GRID2 are functionally identical.

You can use the reserved word PACKED in an array definition of the form:

```
PACKED ARRAY [1 .. n] OF CHAR;
```

In this context, the word PACKED causes the compiler to treat the array as a static string.

When accessing an array, the array's name by itself represents the entire array; the name followed by an index references an individual element in the array, as in the following example.

```
PROCEDURE WORTHLESS;

CONST
  FIRST = 1;
  LAST = 20;

TYPE
  LIST = ARRAY [l..20] OF STRING;

VAR
  I      : INTEGER;
  NAMESA : LIST;
  NAMESB : LIST;

BEGIN
  FOR I := FIRST TO LAST DO
    NAMESA[I] := ' ';
  NAMESB := NAMESA
END;
```

3.4.2 Strings

The predefined type STRING is like a packed array of characters in which byte 0 contains the dynamic length of the string and bytes 1 through n contain the characters. When you declare a string, the compiler allocates a predetermined number of bytes for the string. The default length is 80, but you can specify from 1 to 255 bytes. The dynamic length is the length of the string actually in use, not the total available space. To specify the maximum length of a string, put the length in square brackets, as in the following example:

```
VAR
   TITLE            : STRING[16]
   LINE              : STRING;
   LONGLINE         : STRING[255);
```

You can assign a string of any length to a string variable. You can also assign a CHAR value to a string. The length byte of the string variable reflects the new dynamic length, and the extra bytes are undefined. However, if the assigned string is longer than the maximum length of the string variable, errors can occur. Assigning individual characters to a string does not change the declared length.

To access individual characters in a string, you index the string like an array.

The predefined function LENGTH returns the dynamic length of a string. Section 6 describes several other predefined string routines.

Pascal/MT+ supports static strings, which have a preset, static length. To declare a static string, define it as:

```
PACKED ARRAY [l..n] OF CHAR
```

where n is an integer constant in the range 1 to 255.

Keep in mind the following points about static strings:

- You can assign a string literal to a static string if the string literal is exactly the same length as the static string.
- You can compare static strings to string literals of exactly the same length.
- You can write static strings to TEXT files using the WRITE and WRITELN procedures.

Pascal/MT+ stores string literals as dynamic strings, and the string routines work only with dynamic strings.

3.4.3 Sets

A set is a structured type that contains elements of the same base type. Unlike arrays or records, in which each element has a value, the elements of a set are only significant in their presence or absence from the set. Each element in a set has a corresponding bit. If an element is in a set, its bit is set, if the element is not in the set, its bit is 0.

Set operations are the standard mathematical operations like union, intersection, and difference. Section 4 describes the set operators and expressions.

A set type definition has the general form:

```
SET OF <base type>
```

In Pascal/MT+, the <base type> can be any ordinal type. The ordinal value of the upper and lower bounds of the base type must be in the range 0 to 255. A set-type variable always takes up 32 bytes.

Listing 3-1 is an example program that uses sets.

```
PROGRAM USE_SETS;

VAR
  LOWER, UPPER   : SET OF CHAR;
  DIGIT, DELIMIT : SET OF CHAR;
  I, NUMLETS, NUMDIGS : INTEGER;
  LINE : STRING;

BEGIN
  LOWER := [‘a’..‘z’];
  UPPER := ['A’..'Z’];
  DIGIT := [‘0'..'9’];
  DELIMIT := [‘ ', '.', ‘,’, ‘;’, ‘:’, ‘!’, ‘?’];
  NUMLETS := 0;
  NUMDIGS := 0;
  READLN(LINE);

  FOR I := 1 TO LENGTH(LINE) DO
    IF LINE[I] IN (LOWER + UPPER) THEN
      BEGIN
        NUMLETS := NUMLETS + 1;
        IF LINE[IL IN LOWER THEN (* MAKE UPPERCASE *)
          LINE[IL := CHR(ORD(LINE[I]) - 32)
      END
    ELSE
      IF LINE[I] IN DIGIT THEN
        NUMDIGS := NUMDIGS +1
      ELSE
        IF LINE[I] IN DELIMIT THEN
          LINE[I] := ‘*’
END.
```

Listing 3-1. Program Using Sets

3.4.4 Records

A record is a collection of distinct elements called fields, each of which can be of any type. Records are useful for describing logically related data items that are of different types.

Pascal/MT+ records can either be variant, or nonvariant. Any two nonvariant records of a particular type always have the same internal structure whereas variant records can vary in internal structure.

The type definition for a nonvariant record has the general form:

```
RECORD
  <field list> : <field type> {;
  <field list> : <field type> }
END;
```

The <field list> consists of one or more identifiers separated by commas. Within any given record, each field name must be a unique identifier. Outside the record, the field names can be used for different identifiers. Therefore, two different record types can have identical field names.

The following is an example of a nonvariant record definition:

```
TYPE
   PART = RECORD
      NAME, SOURCE : STRING[10];
      ID_NUMBER    : INTEGER;
      PRICE        : REAL
   END;

VAR
   PARTLIST : ARRAY [NUMPARTS] OF PART;
   NEWPART  : PART;
```

Notice that the field definitions have the same format as variable declarations.

You can reference each element in record by its field name using the following form:

```
<record name>.<field name>
```

where the dot operator connects the record name and field name. For example,

```
NEWPART.PRICE := 29.95;
WRITELN(PARTLIST[I].NAME);
```

A variant record is a record whose internal structure varies depending on how you use the record. That is, you can have two or more records of the same type that have different types of fields.

The variant part of the record's definition acts like a CASE statement (see Section 5.2) because each option in the definition is labeled with one or more values, and the only option whose label matches the value of a selector is used.

The variant part of a record must follow the nonvariant part, and a record can have only one variant part. However, a field within the variant can also be a variant record, so it is possible to nest variants.

The type definition for a variant record has the general form

```
RECORD
   {<field name list> : <field type>;}
   CASE <case selector> OF
      <case label list> : (<field list>) {;
      <case label list> : (<field list>) }
```

where the <field name list> is identical in form, to the list of fields in a record definition and can have a variant part. If a

field has a variant part, it must be the last field in the list. To indicate that a variant has no fields, use an empty parentheses pair.

The <case selector> is either a <tag field> or simply a type name. In either case, the type must be some previously defined simple (scalar) type. The case labels are constants of the type of the selector. If there are more than one, separate them with commas.

If the <case selector> is a <tag field>, it has the form:

<field name> : <type name>

and is one of the regular fields in the record. The field list, or variant with the correct case label, is selected depending on the value of the <tag field>.

The following example shows a variant record definition:

```
RECORD
  NAME: RECORD
          FIRST : STRING[15]
          MID   : CHAR;
          LAST  : STRING[15]
        END;
AGE, BIRTH : INTEGER;
SEX        : CHAR;
CASE EMPLOYED : BOOLEAN OF (* START OF VARIANT PART *)
  FALSE : ( );
  TRUE : (SALARY : REAL;
          CASE EMP_BY : EMP_TYPE OF
            SELF : (YEARS : INTEGER);
            GOV, BUSI : (TITLE : STRING[12];
                         NUMYRS : INTEGER )
          )
END;
```

Both the main variant and the nested variant in the preceding example have a field that controls which variant applies. It is also possible to use a type name to control the variant, as in the following example. This kind of variant is called a free variant.

```
RECORD
  CASE INTEGER OF
    1 : (A, B, C, D : CHAR);
    2 : (X, Y       : INTEGER);
    3 : (Z          : LONGINT)
END;
```

Every field name in a record must be distinct, even if the fields are in different variants. Surround each variant with parentheses; if there are no fields in the variant for a given label, use empty parentheses, ( ).

End of Section 3

Section 4
Operators and Expressions

Pascal/MT+ provides a large assortment of operators for building expressions in several general categories. Table 4-1 briefly describes each of the operators.

Pascal/MT+ evaluates every expression to result in a value of some specific type. The type of the result depends on the operator and the kind of operands in the expression.

The simplest expression is a single operand, which can be a constant, variable, function call, or sub-expression. In an expression with more than one operator, the precedence of the operators determines how Pascal/MT+ evaluates the expression. If two or more operators have the same precedence, they are evaluated from left to right unless you use parentheses to override the normal order of evaluation. For example,

```
4 - 3 + 1 = 2        whereas    4 - (3 + 1) = 0
```

Table 4-1. Summary of Pascal/MT+ Operators

Operator	Operation	Operands	Result	Precedence
		Arithmetic		
+	unary identity	integer or real	same as operand	3rd highest
+	addition,	integer,real or pointer	same as operand	3rd highest
-	unary sign inversion	integer or real	same as operand	3rd highest
-	subtraction,	integer or real	same as operand	3rd highest
*	multiplication	integer or real	integer	2nd highest
div	integer division	integer	integer	2nd highest
/	real division	integer or real	real	2nd highest
mod	modulus	integer	integer	2nd highest

Table 4-1. (continued)

Operator	Operation	Operand	Result	Precedence
		Relational		
=	equality	scalar,string set, pointer record	boolean	lowest
<>	inequality	scalar, string set, pointer record	boolean	lowest
< >	less than greater than	scalar or string	boolean	lowest
<=	less or equal	scalar or string	boolean	lowest
	or set inclusion	set	boolean	lowest
>=	greater or equal	scalar or string	boolean	lowest
	or set inclusion	(see 4.4)	boolean	lowest
IN	set membership	(see 4.4)	boolean	lowest
		Boolean		
NOT	negation	boolean	boolean	highest
OR	disjunction	boolean	boolean	3rd highest
AND	conjunction	boolean	boolean	2nd highest
		Logical		
? - or \	one’s complement of operand	integers and pointers	same as operand	highest
! or |	logical OR	integers and pointers	same as operand	3rd highest
&	logical AND	integers and pointers	same as operand	2nd highest
		Set		
+	union	set	set	3rd highest
-	set difference	set	set	3rd highest

*	intersection	set	set	3rd highest

4.1 Arithmetic Expressions

Pascal/MT+ has operators for addition, subtraction, multiplication, and division. There is no operator for exponentiation.

The arithmetic operators work with integers and reals, and you can mix integers with reals. If both operands are integers, the result is an integer, except with division. Otherwise, the result is a real. A long integer mixed with a regular integer produces a long integer. In an expression, the compiler treats an integer subrange type like an integer.

Be careful with multiplying large numbers, particularly integers. The results of overflows are unpredictable.

The real-number division operator, /, always produces a real-number result. For integer division, use the DIV and MOD operators. DIV gives the integer quotient, and MOD gives the remainder. For example,

```
6 / 3 = 2.0              (* REAL RESULT *)
6 DIV 3 = 2              (* INTEGER RESULT *)
44 DIV 7 = 6
44 MOD 7 = 2
-3 MOD 2 = -1
```

DIV and MOD work with regular and long integers.

4.2 Boolean Expressions

Boolean expressions have either the Boolean value TRUE or FALSE. Two kinds of operators form Boolean expressions:

- Relational operators produce Boolean results, but take operands of many different types.
- Boolean operators work only with Boolean operands.

The relational operators for equality and inequality work with any type except files. The operators that test for ordering only work with simple types and strings. Some relational operators also have special meanings in the context of set expressions, which are described in Section 4.4.

All the relational operators have the same meaning that they do in standard algebraic equations. When testing structures for equality, both structures must have identical contents to be equal.

4.2 Boolean Expressions

Leading and trailing blanks are significant. For example,

'THIS ' <> 'THIS' and ‘XXZZY’ <> ‘ XXZZY’

When testing strings for ordering, the evaluator checks character by character, from left to right until it either reaches the end of a string or finds two characters that do not match. The ordering is based on the ASCII values of the characters. For example,

‘AAAB’ > ‘AAAAAAAAA’

The ordering for enumerated types is based on the ordinal values of the items. For example,

FALSE < TRUE

‘c’ > ‘C’

Remember that relational operators have the lowest precedence. You often have to use parentheses around relational expressions to make them evaluate the way you want. Failure to do so is a common cause of compilation errors. For example, the compiler interprets the expression

X < 3 OR X > 15

as

X < (3 OR X) > 15

which is an invalid expression. The proper way to write the expression is

(X < 3) OR (X > 15)

The Boolean operators AND, OR, and NOT have the same effect as in standard Boolean algebra. Table 4-2 shows the results from Boolean operations. T and F stand for TRUE and FALSE.

Table 4-2. Boolean Operations

A	B	A AND B	A OR B	NOT A
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

4.3 Logical Expressions

Logical expressions perform bitwise logical operations on simple data items. Table 4-3 shows the three logical operators.

Table 4-3. Logical Operators

Operator	Use
&	logical AND
!(or |)	logical OR
~ (or ? or\)	one's complement NOT

The following example uses the logical operators to invert four bits in a variable.

```
MIDBITS  :=  ~(FLAGS &  $00F0);   (* ISOLATE AND INVERT *)
FLAGS    :=  FLAGS & $FF0F;       (* MASK OUT BITS      *)
FLAGS    :=  FLAGS ! MIDBITS;     (* PUT IN NEW FIELD   *)
```

4.4 Set Expressions

There are two classes of operators for sets. One class of operator forms relational expressions that produce Boolean results. The other class of operator forms expressions that build sets.

To form valid expressions, the sets must be of compatible types. Sets are of compatible types if either they are the same type or if the base types for the sets are assignment compatible, as described in Section 5.

The set constructor,

[<member list>]

specifies the values of a set. The <member list> can be any combination of individual elements and closed intervals, separated by commas. The following examples demonstrate the set constructor

```
[1, 3, 5, 7..20, 22, 34]
[l..10, x..y, i+j]
[89, 3, 54, 4..13]
[] (* THIS IS THE EMPTY SET *)
```

The members do not have to be in any order, and they do not have to be constants. You can specify individual members and intervals with variables or expressions. All of the members listed must be in the declared range of values for the set, and the left-hand bound of an interval must not be greater than the right-hand bound.

There are three operators that build sets from other sets:

- The + operator produces the union of two sets.
- The * operator produces the intersection of two sets.
- The - operator produces a set equal to the set on the left, minus all the elements that are in the set on the right.

The following examples demonstrate these set operators:

```
[RED, YELLOW, BLUE] * [RED, GREEN] = [RED]

[l..20] + [3, 5, ll..34] = [l..34]

LETTERS := ['A'..'z’];
CLOSED  := ['A', 'B', 'D', ‘O'..'R’];
OPENED  := LETTERS - CLOSED;
```

There are five relational operators that operate on sets:

- The IN operator tests for membership of an individual item in a set. The item on the left must be of a compatible type with the base type of the set.
- The = operator tests for equality of two sets. Both sets must have exactly the same members.
- The <> operator tests for inequality.
- The <= operator tests for inclusion of the set on the left in the set on the right.
- The >= operator tests for inclusion of the set on the right in the set on the left.

Listing 4-1 demonstrates several of the set operators.

```
PROCEDURE CHECKLINE (ST : STRING);

VAR
  CH : CHAR;
  I : INTEGER;
  ALLOWED, FOUND : SET OF CHAR;

BEGIN
  ALLOWED := [‘A’..‘Z’, ‘0’..’9’, ‘.’, ‘ ‘];
  FOUND := [];
  FOR I := 1 TO LENGTH(ST) DO
     FOUND := FOUND + [ST[I]];
  IF FOUND = ALLOWED THEN
     WRITELN ('ALL USED, NO EXTRAS’)
  ELSE
    IF FOUND <= ALLOWED THEN
      BEGIN
        WRITELN ('NO EXTRA CHARACTERS IN STRING, BUT');
        WRITELN ('THE FOLLOWING CHARACTERS ARE MISSING:');
        FOR CH := CHR(32) TO CHR(126) DO
          IF (CH IN ALLOWED) AND NOT (CH IN FOUND) THEN
            WRITELN (CH)
        END
      ELSE
        IF FOUND >= ALLOWED THEN
          BEGIN
            WRITELN ('ALL CHARACTERS USED, BUT SOME EXTRA:');
            FOR CH := CHR(32) TO CHR(126) DO
              IF (CH IN FOUND) AND NOT (CH IN ALLOWED) THEN
                WRITELN (CH)
            END
          ELSE
            WRITELN ('NOT EVEN IN THE BALLPARK!’)
   END;
```

Listing 4-1. Set Expressions

End of Section 4

Section 5
Statements

This section describes the syntax for each of the Pascal/MT+ statements in alphabetical order. Anywhere in a syntax description that

<statement>

appears, you can use one of the statements described in this section, or you can use a procedure call or compound statement. A compound statement is zero or more statements enclosed by a BEGIN and an END.

5.1 The Assignment Statement

An assignment statement assigns a value to a variable. The general form is

<variable> := <expression>

The assignment statement evaluates the expression on the right and gives that value to the variable on the left. The statement does not change the value of the variable until it evaluates the whole expression. If you use the same variable on both sides of the assignment operator, the statement uses the old value in the expression.

The expression assigned can be of any type. The left and right sides of the assignment statement must be of the same type, with the following exceptions:

- If the variable is REAL the right can be an INTEGER or INTEGER subrange expression.
- The variable's type can be a subrange of the expression as long as the assigned value is in the range of the variable.
- You can assign different set types if all members of the right set can be members of the left set.
- You can assign expressions of type CHAR to variables of type STRING or BYTE.
- You cannot assign files or structures containing files.

Examples:

```
COUNT  := COUNT + 1;

LETTER  :=   ['a'..'z', 'A'..'Z'];

LIST[I]^.VALUE := 163000.0;
```

5.2 The CASE Statement

The CASE statement is a multiple-path branch. The general form is

```
CASE <expression> OF
  { <constant> {, <constant> } : <statement> ; }
END

or

CASE <expression> OF
  { <constant>  {, <constant> } : <statement> ;}
ELSE
  <statement>
END
```

The CASE statement evaluates the <expression> and executes the <statement> that is labeled with the matching value. If no label matches, the <statement> after the ELSE executes. If there is no match and there is no ELSE part, the program flow continues at the next statement after the CASE statement.

The constants labeling the selectable statements must be the same type as the expression, which can be any ordinal type. The same value cannot label more than one path.

The CASE labels are different from declared labels. The scope of a CASE label is confined to the body of the CASE statement. Note also that you cannot reference CASE labels in a GOTO statement.

Examples:

```
CASE CH OF
  ‘a’, ‘A’ : WRITELN ('A');
  ‘q’, ‘Q’ : WRITELN (‘Q’);       (* SEMICOLON OPTIONAL *)
ELSE
  WRITELN ('NOT A OR Q’)
END


CASE COMPARE(N[I], N[I+l]) OF
  LESS    : ; (* DO NOTHING *)
  SAME    : DUPLICATES := DUPLICATES + 1;
  GREATER :
    BEGIN
      SWITCHED := SWITCHED + 1;
      INTERCHANGE(N[I], N[I+l])
    END
END
```

5.3 The Empty Statement

A semicolon by itself is a valid Pascal/MT+ statement called the empty statement. However, if you misplace a semicolon, you can end up with a program that acts differently than you expect. For example, in the following program fragment, the semicolon after the reserved word DO causes an infinite loop. Because the semicolon is misplaced, the only statement in the WHILE loop is the empty statement, and the control variable never changes.

```
WHILE LIST[I] <> ‘ ‘ DO;     (* MISPLACED SEMICOLON *)
  BEGIN
    WRITELN (LIST[I]);
    I := I + 1
  END;
```

The correct form is to omit the semicolon after DO. In general, it is incorrect to put a semicolon before a BEGIN statement.

5.4 The FOR Statement

The FOR statement repeats an action a specified number of times. The general form is

```
FOR <control variable> := <expression> TO <expression> DO
  <statement>
```

or

```
FOR <control variable> := <expression> DOWNTO <expression> DO
  <statement>
```

The FOR statement assigns a succession of values to the <control variable> and executes the statement body once for each value of the variable. In FOR TO statements, the value of the <control variable> increments by one after each repetition.

In FOR DOWNTO statements, the value of the <control variable> decrements by one after each repetition. Note that the value of the <control variable> is undefined after the last repetition.

The expressions that control the FOR statement must be of the same ordinal type as the <control variable>. In the FOR TO statement, if the first <expression> is greater than the second <expression>, the statement body does not execute. The same thing happens in a FOR DOWNTO statement if the first <expression> is less than the second.

The FOR statement evaluates both expressions and stores the values before it executes the statement body. It evaluates the first <expression> before it evaluates the second <expression>. If the first <expression> contains a function reference that changes the value of a variable in the second <expression>, the new value is the one that applies. Evaluating the second <expression> has no effect on the first <expression>.

The <control variable> must be a simple (scalar) variable; it cannot be a pointer-referenced variable or an element of a structure. The scope of the <control variable> must be local to the block containing the FOR statement, and its value must not change inside the statement body.

Examples:

```
FOR CH := ' ' TO 'z' DO
  WRITELN(ORD(CH):3,  ' ', CH)


FOR I := LENGTH(LINE) DOWNTO 1 DO
  WRITE(LINE[I])


FOR X := LEFT TO RIGHT DO
  FOR Y := BOTTOM TO TOP DO
  IF GRID[X, Y] IN ['*', '+'. ':'] THEN
    BEGIN
      STORELOC(X, Y);
      CHECKPATTERN(X, Y)
    END
```

5.5 The GOTO Statement

The GOTO statement transfers program control to a labeled statement. The general form is

```
GOTO <label>
```

The label can be any positive integer literal of one to four digits. You must declare the label in the label declaration section of the block that includes both the GOTO statement and the labeled statement.

The labeled statement must be in the same block as the GOTO statement or at a higher nesting level. The Pascal/MT+ run-time system can transfer control out of routines and structures, including deeply nested recursive routines, to any higher level that meets the scope requirements for the label. However, transferring control into procedures, functions, or structured statements produces unpredictable results.

Examples:

```
PROGRAM USE_GOTO;

LABEL
  9999;

CONST
  MAGIC_WORD = 'QUIT';

VAR
  INP : STRING;

PROCEDURE BAILOUT (INST : STRING);
BEGIN
  IF INST <> MAGIC_WORD THEN
    WRITELN(‘NO, THAT’’S NOT RIGHT')
  ELSE
    GOTO 9999
END;

BEGIN
  WHILE TRUE DO         (* INFINITE LOOP *)
    BEGIN
      WRITELN('WHAT IS THE MAGIC WORD?');
      READLN(INP);
      BAILOUT(INP)
    END;
  9999 :
END.
```

5.6 The IF Statement

The IF statement controls program flow based on the value of a Boolean expression. The general form is

```
IF <Boolean expression> THEN
  <statement>
```

or

```
IF <Boolean expression> THEN
  <statement>
ELSE
  <statement>
```

If the <Boolean expression> is TRUE, the first statement executes. If the <Boolean expression> is FALSE and there is an ELSE part, the second statement executes. If the <Boolean expression> is FALSE and there is no ELSE part, the program flow continues at the next statement.

In a statement of the form,

```
IF <exp> THEN
  IF <exp> THEN
    <statement>
  ELSE
    <statement>
```

the compiler associates the ELSE part with the closest IF.

Examples:

```
IF HELP_REQUEST THEN
 BEGIN
   HELP_DISP;
   GET_LEVEL (LEV);
   MESG_DISP(LEV)
 END


IF SCORE < 60 THEN
 GRADE := 'F'
ELSE
 IF SCORE < 70 THEN
   GRADE := 'D'
 ELSE
   IF SCORE < 80 THEN
     GRADE := 'C'
   ELSE
     IF SCORE < 90 THEN
       GRADE := 'B'
     ELSE
       GRADE := 'A'
```

5.7 The REPEAT Statement

The REPEAT statement executes a group of statements repeatedly until the exit condition is true. The general form is

```
REPEAT
  <statement> {;
  <statement> }
UNTIL <Boolean expression>
```

The REPEAT statement executes the statement body before it evaluates the <Boolean expression> in the UNTIL part. If the <Boolean expression> is TRUE, the REPEAT statement is finished. Note that if the controlling condition does not change in the statement body, the statement loops indefinitely.

Notice that a BEGIN-END pair is not required around the statement body.

Examples:

```
REPEAT
  READLN(INP);
  WRITELN (F, INP);
  LINECNT := LINECNT + 1
UNTIL INP = '.'
```

5.8 The WHILE Statement

The WHILE statement repeatedly executes its statement body, as long as the controlling condition is true. The general form is

```
WHILE <Boolean expression> DO
  <statement>
```

The WHILE statement evaluates the <Boolean expression> before it executes the statement body. If the <Boolean expression> is initially FALSE, the statement body does not execute. As long as the <Boolean expression> is TRUE, the statement body executes.

Examples:

```
WHILE NOT EOF(FN) DO
  BEGIN
    READLN(FN, INP);
    SCAN(INP)
  END


WHILE (I < LENGTH(ST)) AND NOT FOUND DO
  BEGIN
    FOUND := ST[I) = '.';
    I := I + 1
END
```

5.9 The WITH Statement

The WITH statement creates a context for referencing record fields by their individual names. The general form is

```
WITH <record variable> {, <record variable> } DO
  <statement>
```

Inside the statement body, you can reference any field of a specified <record variable> by the field's name. For example, the WITH statement,

```
WITH EMPLOYEE DO
  BEGIN
    NAME  := 'John Doe';
    AGE   := 47;
    TITLE := 'Programmer IV'
  END
```

is equivalent to the three assignment statements,

```
EMPLOYEE.NAME  := 'John Doe';
EMPLOYEE.AGE   := 47;
EMPLOYEE.TITLE := 'Programmer IV';
```

A WITH statement having more than one <record variable> is equivalent to a series of nested WITH statements with one <record variable> specified at each level. A <record variable> can be a field in a previously specified record. For example, the single WITH statement:

```
WITH R1, R2, R3 DO
  <statement>
```

is equivalent to:

```
WITH R1 DO
  WITH R2 DO
    WITH R3 DO
      <statement>
```

If you specify more than one record, and if two records have a field with the same name, the compiler associates the field name with the innermost <record variable>.

Example:

```
PROGRAM SHOW_WITH;

TYPE
  FULLNAME  = RECORD
                FIRST, LAST : STRING[15]
                MIDDLE  : CHAR
              END;
  MEMBER    = RECORD
                NAME   : FULLNAME;
                JOINED : STRING[8];
                ID     : INTEGER
              END;

VAR
  NEWMEM : MEMBER;

BEGIN
  WITH NEWMEM, NAME DO
    BEGIN
      FIRST := 'JOHN';
      MIDDLE := 'Q';
      LAST  := 'PUBLIC';
      JOINED := ‘02/27/53’;
      ID := 0
    END
END.
```

End of Section 5

Section 6
Procedures and Functions

Pascal/MT+ is a block-structured, procedure-oriented language. It contains all the necessary control structures you need to write understandable, and maintainable code. The underlying concept of any procedural language is designing the program as a series of small, logically distinct units that are easy to code, debug, and maintain.

Procedures and functions are essential building blocks in a structured programming language. A procedure is like a parameterized statement, and a function is like a parameterized expression.

In Pascal/MT+, you call (invoke) a procedure by simply using its name. That is, a procedure call is the procedure name, followed by the required parameters. A procedure call is like any valid statement. Anywhere that you can use a statement, you can use a procedure call.

You can put a function reference anywhere that you can put an expression. The function reference is part of the process of evaluating the expression. A function reference, like a procedure call, is just the function name, followed by the required parameters.

Pascal/MT+ functions and procedures can be recursive. They can contain calls to themselves. They can also be mutually recursive. Two procedures or functions can reference each other.

Pascal/MT+ also supports a special type of procedure called an interrupt procedure. See your programmer's guide for details.

In the rest of this section, the word procedure refers to both functions and procedures, unless the context makes it exclude functions.

6.1 Procedure Definitions

A procedure definition, like a program, has a heading followed by a declaration section and a statement body. The following is an example of a procedure definition.

```
PROCEDURE INTERCHANGE(VAR I, J : INTEGER);

VAR
  TEMP : INTEGER;

BEGIN
  TEMP  := I;
  I := J;
  J := TEMP
END;
```

A function definition is like a procedure definition, with the following additions:

- You must specify the data type for the function.
- At least once in the statement body, you must have a special assignment statement that returns the function value.

The data type for a function must be a simple or string type. Put the type name after a colon at the end of the function heading.

To specify the value that a function returns, use an assignment statement with the function name on the left side. You can put more than one of the special assignment statements in the function body, in which case the last value assigned before the function returns control is the value the function returns. The following is an example of a function definition.

```
FUNCTION MIN (L, R : INTEGER) : INTEGER;

BEGIN
  IF L < R THEN
    MIN := L
   ELSE
    MIN := R
END;
```

If you have to reference a procedure before its definition, use a FORWARD declaration, that has the following form:

```
<procedure heading> ; FORWARD;
```

The definition of the procedure, later in the program, does not have the parameter list in the heading. Listing 6-1 is an example of a program with a FORWARD declaration. The two functions are mutually recursive.

```
PROGRAM RECURSE;

VAR
  I : INTEGER;

FUNCTION G (X : INTEGER) : INTEGER; FORWARD;

FUNCTION P (X : INTEGER) : INTEGER;
BEGIN
  IF X < 2 THEN
    F := 1
  ELSE
    F := F(X-1) + G(X-2)
END;

FUNCTION G; (* NO PARAMETER LIST OR FUNCTION TYPE *)
BEGIN
  IF X < 2 THEN
    G := 1
  ELSE
    G := (X*X) + G(F(X-1) MOD X)
END;

BEGIN     (* MAIN PROGRAM *)
  FOR I := 1 TO 10 DO
    WRITELN ('F(', I:2, ') = ', F(I))
END.
```

Listing 6-1. FORWARD Declarations

6.2 Parameters

The parameters in the procedure heading are called formal parameters. The parameters in the procedure call are called actual parameters. There are two types of formal parameters in Pascal/MT+: value and variable parameters. The difference between the two is the way that the parameters are passed at run-time.

A value parameter is like a local variable in the procedure. During a procedure call, the value of the actual parameter passes into the procedure. If you change the value of the formal parameter inside the procedure body, it does not effect the value of the actual parameter. In the procedure call, the actual parameter can be any expression whose type is compatible with the formal parameter.

Changing a variable parameter inside a procedure body changes the actual parameter. During a procedure call, the address of the formal parameter, instead of its value, passes into the procedure. The actual parameter in the procedure call must be a variable whose type is compatible with the formal parameter. A variable parameter cannot be a constant or an element of a packed structure. A file parameter must be a variable parameter.

The following example demonstrates the difference between variable and value parameters. Listing 6-2a shows the program and Listing 6-2b shows the output from the program.

```
PROGRAM VALVAR;

VAR
  XVAL, XVAR : INTEGER;

PROCEDURE MUDDLE (MVAL : INTEGER; VAR MVAR : INTEGER);

BEGIN      (* MUDDLE *)
  MVAL := 11;
  MVAR := 33;

  WRITELN('IN MUDDLE AT END      ', MVAL, MVAR)
END;

BEGIN    (* MAIN PROGRAM *)
  XVAL := 1;
  XVAR := 2;
  WRITELN('IN MAIN BEFORE CALL   ', XVAL, XVAR);
  MUDDLE (XVAL, XVAR) ;
  WRITELN('IN MAIN AFTER CALL    ', XVAL, XVAR)
END.
```

Listing 6-2a. Parameter Passing Program

```
IN MAIN BEFORE CALL 1 2
IN MUDDLE AT END    11 33
IN MAIN AFTER CALL  1 33
```

Listing 6-2b. Output from VALVAR Program

To specify that a parameter is a variable parameter, place the word VAR in the parameter declaration. The VAR applies to all of the parameters grouped together with one type name. In the following procedure heading,

```
PROCEDURE X (VAR I, J, K : INTEGER; M, N : INTEGER);
```

I, J, and K are all variable parameters, and M and N are value parameters.

Besides passing values and variables into procedures, you can also pass procedures and functions. The declaration for a procedural parameter has the same form as a procedure heading. The parameter names in the procedural parameter declaration have no scope outside of the declaration. The formal name for the procedure is the name that the main procedure uses in the statement body.

A procedure or function passed as a parameter can only have value parameters and must be declared in the outermost block.

Listing 6-3 shows a program that uses procedures as parameters.

```
PROGRAM PASSPROC;

TYPE
  REC = RECORD
          NAME, PHONE : STRING
        END;
  PTR = ^REC;
  LST = ARRAY [l..10] OF PTR;

VAR
  LIST : LST;
  J    : INTEGER;

PROCEDURE INIT (PT : PTR);
BEGIN
  WRITELN(‘ENTER A NAME');
  READLN(PT^.NAME);
  WRITELN(‘PHONE NUMBER?');
  READLN(PT^.NUMBER)
END;

PROCEDURE DISPLAY (P : PTR);
BEGIN
  WRITELN(P^.NAME, ‘ : ‘, P^.NUMBER)
END;

PROCEDURE WALKLIST (VAR LS : LST); PROCEDURE WORK(A:PTR));
VAR
  I : INTEGER;
BEGIN
  FOR I := 1 TO 10 DO
    WORK(LS[I])        (* FORMAL PROCEDURAL PARAMETER *)
END;

BEGIN (* MAIN PROGRAM *)
  FOR J := 1 TO 10 DO
    NEW (LIST [J] ) ;
  WALKLIST(LIST, INIT);
  WALKLIST(LIST, DISPLAY)
END.
```

Listing 6-3. Procedural Parameters

6.3 Conformant Arrays

You can define an array parameter for a procedure without specifying the upper- or lower-bounds of the array. This lets you pass different sized arrays to the same procedure. The arrays must have the same number of dimensions, the same element type, and compatible index types.

The declaration for a conformant array is like the declaration for a static array parameter, except that it must be a VAR parameter, and you do not specify the upper- and lower-bounds. Instead, you supply variables that hold the values when the procedure is called. A conformant array declaration has the following form:

```
VAR <name> : ARRAY [<low>..<high>:<type>] OF <type>
```

Inside the procedure body, you can use the boundary variables to control access to the array. Listing 6-4 is an example of a procedure that has a conformant array.

```
PROGRAM DEMOCOM;

VAR
  Al : ARRAY [l..10] OF INTEGER;
  A2 : ARRAY [2..20] OF INTEGER;

PROCEDURE DISPLAYIT
  (VAR AR1 : ARRAY [LOW..HI : INTEGER] OF INTEGER);

(* THE DECLARATION ABOVE DEFINES THREE VARIABLES: *
 * AR1 : THE PASSED ARRAY                         *
 * LOW : LOWER BOUND OF AR1, PASSED AT RUN TIME   *
 * HI  : UPPER BOUND OF AR1, PASSED AT RUN TIME   *)

VAR
  I : INTEGER;

BEGIN   (* DISPLAYIT *)
  FOR  I := LOW TO HI DO
    WRITELN(‘INPUT ARRAY[', I,  ‘] =‘, AR1[I])
END;


BEGIN (* MAIN PROGRAM *)
  WRITELN('DISPLAYING UNINITIALIZED ARRAY A1’);

  DISPLAYIT(Al);               (* PASS Al EXPLICITLY, PASS
                                  1 AND 10 IMPLICITLY *)

  WRITELN('DISPLAYING UNINITIALIZED ARRAY A2');

  DISPLAYIT(A2)                (* PASS A2 EXPLICITLY, PASS
                                  2 AND 20 IMPLICITLY *)
END.
```

Listing 6-4. Conformant Array Example

6.4 Predefined Functions and Procedures

This section describes the predefined functions and procedures of Pascal/MT+. Table 6-1 summarizes these predefined routines.

Note: in the parameter lists for the routines, NUM is an integer or real expression.

Table 6-1. Predefined Functions and Procedures

Arithmetic Functions		
Function	Parameter List	Returns
FUNCTION ABS	(NUM)	REAL
FUNCTION ARCTAN	(NUM)	REAL
FUNCTION COS	(NUM)	REAL
FUNCTION EXP	(NUM)	REAL
FUNCTION LN	(NUM)	REAL
FUNCTION SIN	(NUM)	REAL
FUNCTION SQR	(NUM)	REAL
FUNCTION SQRT	(NUM)	REAL

Bit and byte manipulation routines		
Function	Parameter List	Returns
PROCEDURE CLRBIT	(BASIC_VAR, BIT_NUM)	
FUNCTION HI	(BASIC_VAR)	INTEGER
FUNCTION LO	(BASIC_VAR)	INTEGER
PROCEDURE PACK	(ARRAY, INTEGER, ARRAY)	
PROCEDURE SETBIT	(BASIC_VAR, BIT_NUM)	
FUNCTION SHL	(BASIC_VAR, INTEGER)	INTEGER
FUNCTION SHR	(BASIC_VAR, INTEGER)	INTEGER
FUNCTION SWAP	(BASIC_VAR)	INTEGER
FUNCTION TSTBIT	(BASIC_VAR, BIT_NUM)	BOOLEAN
PROCEDURE UNPACK	(ARRAY, INTEGER, ARRAY)	

Byte and Character manipulation routines	
Function	Parameter List
PROCEDURE FILLCHAR	(DESTINATION, LENGTH, CHARACTER)
PROCEDURE MOVE	(SOURCE, DESTINATION, NUM_BYTES)
PROCEDURE MOVELEFT	(SOURCE, DESTINATION, NUM_BYTES)
PROCEDURE MOVERIGHT	(SOURCE, DESTINATION, NUM_BYTES)

Table 6-1. (continued)

Dynamic allocation routines		
Function	Parameter List	
PROCEDURE DISPOSE	(POINTER, TAG, TAG, ...)	
PROCEDURE NEW	(POINTER, TAG, TAG, ...)	
Input/Output routines		
Function	Parameter List	Returns
PROCEDURE ASSIGN	(FILE, NAME)	
PROCEDURE BLOCKREAD	(FILE, BUF, IOR, NUMBYTES, RELBLK)	
PROCEDURE BLOCKWRITE	(FILE,BUF,IOR,NUMBYTES,RELBLN)	
PROCEDURE CLOSE	(FILE, RESULT)	
PROCEDURE CLOSEDEL	(FILE, RESULT)	
FUNCTION EOF	(FILE)	BOOLEAN
FUNCTION EOLN	(FILE)	BOOLEAN
PROCEDURE GET	(FILE)	
FUNCTION GNB	(FILE)	CHAR
FUNCTION IORESULT		INTEGER
PROCEDURE OPEN	(FILE, TITLE, RESULT)	
PROCEDURE OPENX	(FILE, TITLE, RESULT, EXTENT)	
PROCEDURE PAGE	(FILE)	
PROCEDURE PURGE	(FILE)	
PROCEDURE PUT	(FILE)	
PROCEDURE READ	(FILE, VARIABLE, VARIABLE, ...)	
PROCEDURE READHEX	(FILE, VAR, SIZE);	
PROCEDURE READLN	(FILE, VARIABLE, VARIABLE, ...)	
PROCEDURE RESET	(FILE)	
PROCEDURE REWRITE	(FILE)	
PROCEDURE SEEKREAD	(FILE, RECORD_NUMBER)	
PROCEDURE SEEKWRITE	(FILE, RECORD_NUMBER)	
FUNCTION WNB	(FILE, CHAR)	BOOLEAN
PROCEDURE WRITE	(FILE, VARIABLE, VARIABLE, ...)	
PROCEDURE WRITEHEX	(FILE, EXPRESSION, SIZE)	
PROCEDURE WRITELN	(FILE, VARIABLE, VARIABLE, ...)	
PROCEDURE LWRITEHEX	(FILE, EXPRESSION, SIZE) *	

* does not apply to the 8080 implementation

Table 6-1. (continued)

String handling routines		
Function	Parameter List	Returns
FUNCTION CONCAT	(SOURCE1, SOURCE2,...,SOURCEn)	STRING
FUNCTION COPY	(SOURCE, LOCATION, NUM_BYTES)	STRING
PROCEDURE DELETE	(TARGET, INDEX, SIZE	
PROCEDURE INSERT	(SOURCE, DESTINATION, INDEX)	
FUNCTION LENGTH	(STRING)	INTEGER
FUNCTION POS	(PATTERN, SOURCE)	INTEGER
Transfer Functions		
Function	Parameter List	Returns
FUNCTION CHR	(INTEGER)	CHAR
FUNCTION ODD	(ORDINAL)	BOOLEAN
FUNCTION ORD	(ORDINAL)	INTEGER
FUNCTION ROUND	(NUM)	INTEGER
FUNCTION TRUNC	(NUM)	INTEGER
Miscellaneous routines		
Function	Parameter List	Returns
FUNCTION @BDOS	(INTEGER, WORD)**	INTEGER
FUNCTION @BDOS86	(INTEGER, POINTER)*	INTEGER
FUNCTION @CKD		PTR_TO_STRING
PROCEDURE @ERR	(INTEGER)	
FUNCTION @HERR		
PROCEDURE @HLT		
FUNCTION @MRK		INTEGER
FUNCTION @RLS	(INTEGER)	
FUNCTION ADDR	(VARIABLE REFERENCE)	INTEGER
PROCEDURE CHAIN		
PROCEDURE EXIT		
PROCEDURE INLINE	(see Programmer's Guide)	
FUNCTION MAXAVAIL		INTEGER
FUNCTION MEMAVAIL		INTEGER
FUNCTION PRED	(X)	same type as X
FUNCTION RIM85		** BYTE
FUNCTION SIZEOF	(VARIABLE OR TYPE NAME)	INTEGER
PROCEDURE SIM85	(VAL : BYTE)	**
FUNCTION SUCC	(X)	same type as X
PROCEDURE WAIT	(PORTNUM , MASK, POLARITY)	**

* does not apply to the 8080 implementation
** does not apply to the 8086 implementation

ABS Function

Syntax:

```
FUNCTION ABS(X);
```

Explanation:

ABS returns the absolute value of X. X must be a real or integer expression. The result has the same type as X.

Examples:

```
ABS(-5.789) = 5.789

ABS(56)     = 56
```

ADDR Function

Syntax:

```
FUNCTION ADDR(VARIABLE OR ROUTINE) : POINTER;
```

Explanation:

ADDR returns the address of a variable, function, or procedure. Variable references can include subscripted variables and record fields. ADDR does not work with constants, user-defined ordinal types, or any item that does not take code or data space.

You can reference externals, including those in overlays. However, you must keep in mind the scope of the referenced item. For example, you cannot use ADDR in the main program to find the address of a variable you declare in a nested procedure.

Example:

```
PROCEDURE ADDR_DEMO(PARAM : INTEGER);
VAR
  REC : RECORD
          J : INTEGER;
          BOOL : BOOLEAN;
        END;
ADDRESS : INTEGER;
R : REAL;
S1   ARRAY[l..10] OF CHAR;
P : ^INTEGER;

BEGIN
  P := ADDR(ADDR DEMO);
  P := ADDR(PARAM);
  P := ADDR(REC);
  P := ADDR(REC.J);
END;
```

ARCTAN Function

Syntax:

```
FUNCTION ARCTAN(X);
```

Explanation:

ARCTAN returns the angle, expressed in radians, whose tangent is X. X must be a real or integer expression. The result is real number.

Example:

```
ARCTAN(L) = 0.     (* THE ANGLE IS PI / 4 *)
```

ASSIGN Function

Syntax:

```
PROCEDURE ASSIGN( FILE, NAME );
```

Explanation:

ASSIGN attaches an external filename to a file variable before using a RESET or REWRITE procedure. FILE is a filename; NAME is a literal or a variable string containing the name of the file to create. FILE can be of any type, but must be of type TEXT to use the special device names listed in Table 6-2.

Pascal/MT+ implements the Pascal local file facility using temporary filenames in the form

```
PASTMPxx.$$$
```

where xx is sequentially assigned, starting at zero, from the beginning of each program.

If an ASSIGN does not precede an external file REWRITE, a temporary filename attaches before creation. Locally declared files cannot be used as temporary files unless you initialize the file with ASSIGN(<file>,'').

The following table defines the device names supported in the CP/M® run-time environment.

Table 6-2. Device Names

Name	Definition
CON:	As input, echoes input characters, CR as CR/LF, and backspace [CHR(8)] as backspace, space, backspace As output, echoes CR as CR/LF and CP/M expands tabs to every 8 character positions. Line-feed cannot be output.
KBD:	CP/M console, input device only. No echo or interpretation. Cannot be used with CON: input or output.
TRM:	CP/M console, output device only. No interpretation
LST:	CP/M printer, output device only. No interpretation, including no tab expansion.

Table 6-2. (continued)

Name	Definition
RDR:	CP/M reader, input device only. Call auxiliary input routine in the BIOS via the BDOS, using Function 3.
PUN:	CP/M punch, output device only. Call auxiliary output routine in the BIOS via the BDOS, using Function 4.

Note that using CON: and KBD: together can create problems because of the way they are implemented. To implement CTRL-S, CP/M checks for typed characters when performing BDOS Function 2, writing to CON:. If you type a character other than CTRL-S, CP/M stores it internally, anticipating a subsequent call using Function 1.

Function 6, used by KBD:, goes directly to the BIOS for input, ignoring any character in this internal buffer. Therefore, your program might appear to be losing characters when in fact CP/M is storing them internally.

Examples:

```
ASSIGN(CONIN, ‘CON:’)
ASSIGN(KEYBOARD,’KBD:');
ASSIGN(CRT,'TRM:');
ASSIGN(PRINTFILE,'LST:');
```

BLOCKREAD, BLOCKWRITE Function

Syntax:

```
BLOCKREAD (F:FILEVAR; BUF:ANY; VAR IOR:INTEGER; SZ,RB:INTEGER);
BLOCKWRITE(F:FILEVAR; BUF:ANY; VAR IOR:INTEGER; SZ,RB:INTEGER);
```

Explanation:

These procedures enable direct disk access. FILEVAR is an untyped file (FILE;). BUF is any array variable large enough to hold the data. It can be indexed. IOR is an integer that receives the returned value from the operating system. SZ is the number of bytes to transfer. SZ is related to the size of BUF; it must be a multiple of 128.

If BUF is 128 bytes, SZ must be 128. If BUF is 4096 bytes, SZ can be as large as 4096. RB is the relative block number, which can be in the range -1 to 32767. When RB is -1, the run-time routines assume sequential block transfer. When RB is greater than -1, the routine calculates the correct file location and opens new extents as needed.

The data transfers either to or from your BUF variable for the specified number of bytes.

CHAIN Function

Syntax:

```
PROCEDURE CHAIN(FILE);
```

Explanation:

CHAIN allows you to chain from one program to another.

See Section 3.3 in the Pascal/MT+ Language Programmer's Guide for more information.

CHR Function

Syntax:

```
FUNCTION CHR(X) : CHAR;
```

Explanation:

CHR returns the character whose ASCII value is the integer X.

Examples:

```
WRITELN(CHR(7));      (* BEEP THE TERMINAL *)

IF C IN ['a'..'z'] THEN
   C := CHR(ORD(C) - 32); (* CONVERT TO UPPERCASE *)
```

CLOSE Function

Syntax:

```
PROCEDURE CLOSE    ( FILE, RESULT )
PROCEDURE CLOSEDEL ( FILE, RESULT )
```

Explanation:

The CLOSE procedure closes files. You must use it to guarantee that data written to a file is purged from the buffer to the disk.

CLOSEDEL closes and deletes temporary files after use. FILE is any filetype variable. RESULT is a VAR INTEGER parameter that has the same value as IORESULT upon return from CLOSE.

Files are implicitly closed when an open file is RESET. The number of files that can be open at a time is CPU-dependent. For CP/M systems, this number is limited only by the amount of memory available for File Control Blocks (FCBs).

CONCAT Function

Syntax:

```
FUNCTION CONCAT( SOURCE1, SOURCE2, ... , SOURCEn) : STRING;
```

Explanation:

CONCAT returns a string in which all strings in the parameter list are concatenated. The strings can be string variables, string literals, or characters. You can concatenate a string of zero length. The total length of all strings truncates at 256 bytes. See the COPY function for restrictions when using both CONCAT and COPY.

Example:

```
PROCEDURE CONCAT_DEMO;
VAR
  S1,S2 : STRING;
BEGIN
  S1 :=  'left link, right link';
  S2 :=  'root root root';
  WRITELN(Sl,'/',S2);
  S1 := CONCAT(S1,' ',S2,'!!!!!!');
  WRITELN(S1);
end;
```

Output:

```
left link, right link/root root root
left link, right link root root root!!!!!!
```

COPY Function

Syntax:

```
FUNCTION COPY( SOURCE, LOCATION, NUM BYTES) : STRING;
```

Explanation:

COPY returns a string with the number of characters specified in NUM BYTES from SOURCE, beginning at the index specified in LOCATION. SOURCE must be a string. LOCATION and NUM_BYTES are integer expressions.

The COPY routine does not check whether LOCATION is out of bounds or negative. Truncation occurs if NUM_BYTES is negative or NUM_BYTES plus LOCATION exceeds the length of the SOURCE.

Example:

```
PROCEDURE COPY_DEMO;
BEGIN
  LONG STR := 'Hi from Cardiff-by-the-sea';
  WRITELN(COPY(LONG_STR,9,LENGTH(LONG_STR)-9+1));
END;
```

Output:

```
Cardiff-by-the-sea
```

Note: COPY and CONCAT are string returning pseudo-functions and have only one statically allocated buffer for the return value. Therefore, if you use these functions more than once within the same expression, the value of each occurrence becomes the value of the last occurrence. For example,

```
CONCAT(A,STRING1) = CONCAT(A,STRING2)
```

is always true, because the concatenation of A and STRING2 replaces that of A and STRING1. As a further example,

```
WRITELN( COPY(STRING1,1,4), COPY(STRING1,5,4))
```

writes the second set of four characters in STRING1 twice.

COS Function

Syntax:

```
FUNCTION COS(X) : REAL;
```

Explanation:

COS returns the cosine of X. X ' the angle in radians, must be real or integer. The result is real.

Example:

```
IF COS(ANG) = SIN(ANG) THEN
  WRITELN(‘45 DEGREES');
```

DELETE Function

Syntax:

```
PROCEDURE DELETE( TARGET, INDEX, SIZE);
```

Explanation:

DELETE removes SIZE characters from TARGET beginning at the byte named in INDEX. TARGET is a string. INDEX and SIZE are integer expressions. No action occurs if SIZE is zero.

Note: serious errors result if SIZE is negative. The data and surrounding memory can be destroyed if the INDEX plus the SIZE is greater than the TARGET, or the TARGET is empty.

Example:

```
PROCEDURE DELETE_DEMO;
VAR
  LONG_STR : STRING;
BEGIN
  LONG_STR := ‘       get rid of the leading blanks';
  WRITELN (LONG_STR) ;
  DELETE(LONG_STR,l,POS('g',LONG_STR)-l);
  WRITELN(LONG_STR);
END;
```

Output:

```
      get rid of the leading blanks
get rid of the leading blanks
```

DISPOSE Function

Syntax:

```
PROCEDURE DISPOSE(VAR P : POINTER);
PROCEDURE DISPOSE(VAR P : POINTER, VARIANTS);
```

Explanation:

DISPOSE deallocates space that NEW allocates. When DISPOSE returns, the value of the pointer variable is undefined. If you are using the FULLHEAP memory manager, the space is available for reuse. Otherwise, the space is not available for reallocation.

See NEW for an example of using DISPOSE and more information about deallocating variant records.

EOLN, EOF Function

Syntax:

```
FUNCTION EOLN : BOOLEAN;
FUNCTION EOLN(VAR F : TEXT) : BOOLEAN;
FUNCTION EOF : BOOLEAN;
FUNCTION EOF(VAR F : FILE) : BOOLEAN;
```

Explanation:

EOLN returns TRUE when the window variable is over the end-of-line character in a file. EOF returns TRUE when the window variable is over an end-of-file character. If you do not specify a file, the default input file is assumed.

EOLN returns TRUE on disk TEXT files when a READ statement reads the last valid character on a line. The sequence of statements for a READ on nonconsole files is,

```
CH := F^;
GET(F);
```

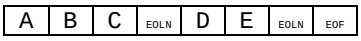
This positions the window variable over the end-of-file character. Thus, EOLN returns TRUE on nonconsole TEXT files when the last character is read, and a blank returns instead of the end-of-line character.

On console files, this sequence reverses; READ has an initial call to GET followed by an assignment from the window variable. For this reason, EOLN returns TRUE in console files after the carriage/return line-feed is read. EOLN returns TRUE in nonconsole files after the last character is read. A blank still returns in the character.

EOF, like EOLN, returns TRUE when the last character is read on nonconsole files. On console files, EOF is TRUE only when the end-of-file indicator is entered. The system does not support reading past the end-of-file on console or disk files; it can crash. The window variable returns a blank when EOF is TRUE.

EOF does not become TRUE at the end of the valid data in non-TEXT files if the data does not fill up the entire last sector of the file.

The following example illustrates these concepts. Suppose the input stream for a TEXT file consists of



If you repeatedly read characters from this stream, EOLN and EOF return the values summarized in Table 6-3.

Table 6-3. EOLN, EOF Values for a TEXT File

Console			Nonconsole		
Character returned	EOLN	EOF	Character returned	EOLN	EOF
A	F	F	A	F	F
B	F	F	B	F	F
C	F	F	C	T	F
space	T	F	space	F	F
D	F	F	D	F	F
E	F	F	E	T	F
space	T	F	space	T	T
space	T	T	space	T	T

For a non-TEXT file, suppose the input stream consists of

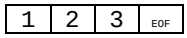


Table 6-4 shows the values of EOF when you repeatedly read integers from the input stream.

Table 6-4. EOF Values for a Non-TEXT File

Value returned	EOF
1	F
2	F
3	F
6682	F
.	.
.	.
.	.
6682	T

(Note that 6682 is the end of the sector)

EXIT Function

Syntax:

```
PROCEDURE EXIT;
```

Explanation:

EXIT leaves the current procedure or function, or the main program. If used in an INTERRUPT procedure, EXIT also loads the registers and reenables interrupts before exiting. EXIT is the equivalent of the RETURN statement in FORTRAN or BASIC. You usually execute it as a statement following a test.

Example:

```
PROCEDURE EXITTEST;
{ EXIT THE CURRENT FUNCTION OR MAIN PROGRAM. }

  PROCEDURE EXITPROC(BOOL : BOOLEAN);

  BEGIN
    IF BOOL THEN
      BEGIN
        WRITELN('EXITING EXITPROC');
        EXIT;
      END;
    WRITELN('STILL IN EXITPROC, ABOUT TO LEAVE NORMALLY');
  END;

BEGIN
  WRITELN('EXITTEST........');
  EXITPROC (TRUE) ;
  WRITELN('IN EXITTEST AFTER 1ST CALL TO EXITPROC');
  EXITPROC (FALSE) ;
  WRITELN('IN EXITTEST AFTER 2ND CALL TO EXITPROC');
  EXIT;
  WRITELN('THIS LINE WILL NEVER BE PRINTED');
END;
```

Output:

```
EXITTEST.......
EXITING EXITPROC
IN EXITTEST AFTER 1ST CALL TO EXITPROC
STILL IN EXITPROC, ABOUT TO LEAVE NORMALLY
IN EXITTEST AFTER 2ND CALL TO EXITPROC
```

EXP Function

Syntax:

```
FUNCTION EXP(X) : REAL;
```

Explanation:

EXP returns the exponential of X. X must be real or integer. The result is real. The function returns a value that is the natural logarithm (base e) , raised to the power of X. Use this function with the natural logarithm function, LN.

Examples:

```
IF (EXP(LN(X) + LN(Y)) - (X * Y) <= TOLERANCE THEN
  WRITELN('LOGARITHM FUNCTIONS PASS TEST');

WRITELN(X, '**', Y, '=', EXP(Y * LN(X)));
```

FILLCHAR Function

Syntax:

```
PROCEDURE FILLCHAR( DESTINATION, LENGTH, CHARACTER);
```

Explanation:

FILLCHAR is a fast way to fill in large data structures with the same data. For example, FILLCHAR can blank out a buffer.

DESTINATION is a variable reference, but need not be a packed array of characters as in UCSD Pascal. It can be subscripted. LENGTH is an integer expression.

Note: if LENGTH is negative or greater than the length of DESTINATION, it overwrites adjacent code or data. CHARACTER is a literal or variable of type CHAR. Fill the DESTINATION with the number of characters specified by LENGTH.

Example:

```
PROCEDURE FILL_DEMO;

VAR
  BUFFER : PACKED ARRAY[l..256] OF CHAR;

BEGIN
  FILLCHAR(BUFFER,256,' ‘);        {BLANK THE BUFFER}
END;
```

GET Function

Syntax:

```
PROCEDURE GET(VAR F : FILE VARIABLE);
```

Explanation:

GET advances the window variable by one element and moves the contents of the indicated file into the window variable. EOF must be FALSE before GET executes. When there is no next element, EOF becomes TRUE and the value of the window variable becomes undefined. See Section 7 for more details on GET and TEXT files.

HI, LO, SWAP Function

Syntax:

```
FUNCTION     HI(BASIC_VAR) : INTEGER;
FUNCTION     LO(BASIC_VAR) : INTEGER;
FUNCTION   SWAP(BASIC_VAR) : INTEGER;
```

Explanation:

HI returns the upper 8 bits of BASIC_VAR (an 8- or 16-bit variable) in the lower 8 bits of the result.

LO returns the lower 8 bits, with the upper 8 bits forced to zero.

SWAP returns the upper 8 bits of BASIC_VAR in the lower 8 bits of the result and the lower 8 bits of BASIC_VAR in the upper 8 bits of the result.

Passing an 8-bit variable to HI results in 0. Passing 8 bits to LO does nothing.

The following example shows the results of these functions.

Example:

```
PROCEDURE HI_LO_SWAP;
VAR
  HL : INTEGER;
BEGIN
  WRITELN('HI_LO_SWAP........');
  HL := $104;
  WRITELN('HL=',HL);
  IF HI(HL) = 1 THEN
    WRITELN('HI(HL)=',HI(HL));
  IF LO(HL) = 4 THEN
    WRITELN('LO(HL)=',LO(HL));
  IF SWAP(HL) = $0401 THEN
    WRITELN('SWAP(HL)=',SWAP(HL));
END;
```

Output:

```
HI_LO SWAP.......
HL=260
HI(HL)=l
LO(HL)=4
SWAP(HL)=1025
```

INLINE Function

Syntax:

```
PROCEDURE INLINE(arg/arg/...);
```

Explanation:

INLINE is a built-in feature that allows you to insert data in the middle of a Pascal/MT+ procedure or function. You can insert small machine-code sequences and constant tables into a Pascal/MT+ program without using externally-assembled routines.

Section 4.3.2 of the Pascal/MT+ Language Programmer's Guide has examples of using INLINE.

INSERT Function

Syntax:

```
PROCEDURE INSERT(SOURCE, DESTINATION, INDEX);
```

Explanation:

INSERT puts SOURCE into DESTINATION at the location specified in INDEX. DESTINATION is a string. SOURCE is a character or string, literal or variable. INDEX is an integer expression. SOURCE can be empty.

Note: if INDEX is out of bounds or DESTINATION is empty, it destroys data. If inserting SOURCE into DESTINATION makes DESTINATION too long, it is truncated.

Example:

```
PROCEDURE INSERT_DEMO;
VAR
  LONG_STR : STRING;
  S1 : STRING[10];
BEGIN
  LONG_STR := 'Remember Luke';
  S1  :=   'the Force,';
  INSERT(S1,LONG_STR,10);
  WRITELN(LONG_STR);
  INSERT('to use ',LONG_STR,10);
  WRITELN(LONG_STR);
end;
```

Output:

```
Remember the Force, Luke
Remember to use the Force, Luke
```

IORESULT Function

Syntax:

```
FUNCTION IORESULT : INTEGER;
```

Explanation:

After each I/O operation, the run-time library routines set the value returned by the IORESULT function. In general, the value of IORESULT is system-dependent. Never attempt to WRITE the IORESULT because it resets to 0 before any I/O operation.

Refer to the Pascal/MT+ Language Programmer's Guide for more information about IORESULT.

Example:

```
ASSIGN(F,'C:HELLO');
RESET(F);

IF IORESULT = 255 THEN
   WRITELN('C:HELLO IS NOT PRESENT');
```

LENGTH Function

Syntax:

```
FUNCTION LENGTH( STRING) : INTEGER;
```

Explanation:

LENGTH returns the integer value of the length of the string.

Example:

```
PROCEDURE LENGTH_DEMO;
VAR
  S1 : STRING [40]
BEGIN
  S1 := 'This string is 33 characters long';
  WRITELN('LENGTH OF ',S1,'=',LENGTH(S1));
  WRITELN('LENGTH OF EMPTY STRING = ',LENGTH(‘’));
END;
```

Output:

```
LENGTH OF This string is 33 characters long=33
LENGTH OF EMPTY STRING = 0
```

LN Function

Syntax:

```
FUNCTION LN(X) : REAL;
```

Explanation:

LN returns the natural logarithm of X. X must be real or integer. The result is real.

MAXAVAIL, MEMAVAIL Function

Syntax:

```
FUNCTION MAXAVAIL : INTEGER;
FUNCTION MEMAVAIL : INTEGER;
```

Explanation:

The functions MAXAVAIL and MEMAVAIL work with NEW and DISPOSE to manage the heap memory area in Pascal/MT+.

MEMAVAIL returns the available memory at any given time, regardless of fragmentation. MAXAVAIL reports the largest block available.

If the result of these functions displays as a negative number, the amount of memory remaining is too large to express as a positive integer. You can display the return value with WRITEHEX.

See your Pascal/MT+ Language Programmer's Guide for more information on the use of dynamic memory.

MOVE, MOVERIGHT, MOVELEFT Function

Syntax:

```
PROCEDURE MOVE     (SOURCE, DESTINATION, NUM_BYTES)
PROCEDURE MOVELEFT (SOURCE, DESTINATION, NUM_BYTES)
PROCEDURE MOVERIGHT(SOURCE, DESTINATION, NUM_BYTES)
```

Explanation:

These procedures move the number of bytes contained in NUM BYTES from the SOURCE location to the DESTINATION location. MOVE and MOVELEFT are synonyms. They move from the left end of the source to the left end of the destination. MOVERIGHT moves from the right end of the source to the right end of the destination. The parameters passed to MOVERIGHT specify the left end of the source and destination.

The source and destination can be variables of any type, and they need not be of the same type. They can be pointers to variables, but not named or literal constants. The number of bytes is an integer expression between 0 and 64K.

MOVELEFT and MOVERIGHT transfer bytes from one data structure to another or move data within a data structure. These procedures move on a byte level, ignoring the data structure type. MOVERIGHT transfers bytes from the low end of an array to the high end. Without this procedure, you would need a FOR loop to pick up each character and put it down at a higher address. MOVERIGHT is much faster. You can use MOVERIGHT in an insert character routine to make room for characters in a buffer.

MOVELEFT can transfer bytes from one array to another, delete characters from a buffer, or move the values in one data structure to another.

When you use these procedures keep in mind the following:

- These procedures do not check whether the number of bytes is greater than the size of the destination. If the destination is not large enough, bytes spill into the adjacent data storage area.
- Moving 0 bytes moves nothing.
- There is no type checking.

Example:

```
PROCEDURE MOVE_DEMO;
CONST
  STRINGSZ = 80;
VAR
  BUFFER : STRING[STRINGSZ];
  LINE : STRING;

PROCEDURE INSRT(VAR DEST : STRING; INDEX : INTEGER; VAR SOURCE :
STRING);
BEGIN
  IF LENGTH(SOURCE) <= STRINGSZ - LENGTH(DEST) THEN
    BEGIN
      MOVERIGHT(DEST[ INDEX ], DEST[ INDEX+LENGTH(SOURCE)],
        LENGTH(DEST)-INDEX+1);
      MOVELEFT(SOURCE[1], DEST[INDEX], LENGTH(SOURCE));
      DEST[0] :=CHR(ORD(DEST[0]) + LENGTH(SOURCE))
    END;
END;

BEGIN
  WRITELN('MOVE_DEMO.......');
  BUFFER := 'Judy J. Smith/ 335 Drive/ Lovely, Ca. 95666';
  WRITELN(BUFFER);
  LINE := 'Roland '
  INSRT(BUFFER, POS('5',BUFFER)+2,LINE);
  WRITELN (BUFFER);
END;
```

Output:

```
MOVE_DEMO......
Judy J. Smith/ 355 Drive/ Lovely, Ca. 95666
Judy J. Smith/ 355 Roland Drive/ Lovely, Ca. 95666
```

NEW Function

Syntax:

```
PROCEDURE NEW (VAR P : POINTER);
PROCEDURE NEW (VAR P : POINTER; VARIANTS);
```

Explanation:

NEW dynamically allocates space for a record of the pointer's type, and sets the value of the pointer to the new record. For variant records, the procedure allocates enough space to hold the largest variant, unless you specify which variant you want.

Specify the variant by its tag value. If the record has nested variants, specify the variants in the order of nesting. When you deallocate a record with DISPOSE, use the same parameter list.

Example:

```
PROGRAM NEWDEMO;

TYPE
  COL = (RED, YELLOW, BLUE, GREEN, ORANGE, PURPLE);
  PTR = ^REC;
  REC = RECORD
    A : INTEGER;
    CASE LIGHT : COL OF
      RED    : ();
      YELLOW : (R : REAL);
      BLUE   : (
        CASE   TINT : COL OF
          GREEN : (W, X, Y, Z : INTEGER);
          PURPLE : (H, I, J, K : REAL)
               )
    END;

VAR
  GENERAL, SMALL, BIG  : PTR;

BEGIN
  WRITELN('THIS PROGRAM DOES NOTHING BUT TWEAK THE HEAP');

  NEW(GENERAL);           (* FOR ANY VARIANT *)
  NEW(SMALL, RED);        (* FOR SMALLEST VARIANT *)
  NEW(BIG, BLUE, PURPLE); (* FOR LARGER VARIANT *)

  DISPOSE(GENERAL);
  DISPOSE(SMALL, RED);
  DISPOSE(BIG, BLUE, PURPLE)
END.
```

ODD Function

Syntax:

```
FUNCTION ODD(INTEGER) : BOOLEAN;
```

Explanation:

ODD returns TRUE if the expression is odd and FALSE if it is not.

Example:

```
IF ODD(LENGTH(ANSWER)) THEN
  WRITELN('THAT''S ODD!')
ELSE
  WRITELN('EVEN I BELIEVE THAT')
```

OPEN Function

Syntax:

```
PROCEDURE OPEN ( FILE, FILENAME, RESULT );
```

Explanation:

The OPEN procedure opens an existing file for input. FILE is any file variable. Filename is a string that contains the CP/M filename. RESULT is an integer variable, which on return from OPEN, has the same value as IORESULT.

The OPEN procedure is the same as the sequence:

```
ASSIGN(FILE, FILENAME);
RESET (FILE) ;
RESULT := IORESULT;
```

Example:

```
OPEN ( INFILE, 'A:FNAME.DAT', RESULT);
```

ORD Function

Syntax:

```
FUNCTION ORD(SCALAR) : INTEGER;
```

Explanation:

ORD returns the ordinal value of a scalar or enumerated type expression. The result is an integer. For an enumerated type, the ordinal value is the same as the order of declaration, starting with 0.

Example:

```
FUNCTION DIG2DEC (C : CHAR) : INTEGER;

(* C MUST BE IN THE RANGE '0' ..'19' *)

BEGIN
  DIG2DEC := ORD(C) - ORD('0');
END;
```

PACK, UNPACK Function

Syntax:

```
PROCEDURE PACK(A : ARRAY[M ... N] OF T; Z : ARRAY[U ... V] OF T;
PROCEDURE UNPACK(A : ARRAY[M ... N] OF T; Z : ARRAY[U ... V) OF
T;
```

Explanation:

The Pascal/MT+ compiler accepts PACK and UNPACK but does not execute them. Because Pascal/MT+ is byte-oriented, these procedures are unnecessary.

PAGE Function

Syntax:

```
PROCEDURE PAGE(FILE VARIABLE);
```

Explanation:

PAGE skips to the top of a new page when a TEXT file is printing by inserting a begin-page character in the output file. If you do not specify the output file, it defaults to standard output.

POS Function

Syntax:

```
FUNCTION POS( PATTERN, SOURCE )  : INTEGER;
```

Explanation:

POS returns the integer value of the position of the first occurrence of PATTERN in SOURCE. If PATTERN is not in the string, the function returns 0. SOURCE is a string. PATTERN is a string, character, or literal.

Example:

```
PROCEDURE POS_DEMO;
VAR
  STR,PATTERN : STRING;
  CH : CHAR;
BEGIN
  STR := 'Ada Lovelace';
  PATTERN := 'Love';
  CH := 'v';
  WRITELN('position of ',PATTERN,' in ',STR,' is
  POS(PATTERN,STR));
  WRITELN('position of ',CH,' in ',STR,' is ',POS(CH,STR));
  WRITELN('pos of ''z'' in ',STR,' is ',POS('z',STR));
END;
```

Output:

```
position of Love in Ada Lovelace is 5
position of v in Ada Lovelace is 7
position of 'z' in Ada Lovelace is 0
```

PRED Function

Syntax:

```
FUNCTION PRED(SCALAR) : SCALAR;
```

Explanation:

PRED returns the value of the predecessor of a scalar expression. The ordinal value of the predecessor is 1 less than the ordinal value of the expression.

Example:

```
TYPE
  WEEKDAY = ( SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
              THURSDAY, FRIDAY, SATURDAY );

PRED(FRIDAY) = THURSDAY
PRED(2 * 2) = 3

PRED('D')     = 'C'
```

PURGE Function

Syntax:

```
PROCEDURE PURGE( FILE );
```

Explanation:

PURGE deletes the file associated with the file variable. The file is deleted from the disk directory.

Example:

```
ASSIGN(F,'BADFILE.BAD');

PURGE(F);        (* DELETE BADFILE.BAD *)
```

PUT Function

Syntax:

```
PROCEDURE PUT(FILE VARIABLE);
```

Explanation:

PUT transfers the contents of the window variable associated with F to the next available record in the file. You must assign to the window variable before executing a PUT. You can use this procedure only if EOF is TRUE. After execution, EOF remains TRUE and the window variable becomes undefined.

READ, READLN Function

Syntax:

```
PROCEDURE READ (FILE VARIABLE, variable, variable, ...);
PROCEDURE READLN(FILE VARIABLE, variable, variable, ...);
```

Explanation:

These procedures read from the file associated with the file variable into the variables listed. If you do not specify a file, the procedures default to the standard input.

READLN works with TEXT files only, but both routines, when reading from TEXT files, convert Booleans, reals, and integers from their ASCII representations. All numbers convert on input, but the formatting is lost. Therefore, you should separate numbers from each other and from other data types by a blank or a carriage return/line-feed.

READLN reads the data and then sets the file pointer at the beginning of the next line. READ does not skip over data. When reading strings, both procedures read from the current position to the end of the line. Use READLN to read strings.

When reading from non-console files, the sequence of operations for each data item is equivalent to:

```
<variable> := F^;
GET(F);
```

When reading from the console, the sequence is

```
GET(F);
<variable> := F^;
```

For non-TEXT files, the variables in the parameter list must be the same type as the data read from the file. The compiler does not typecheck, however. You must construct a parameter list compatible with your file's format.

READHEX, WRITEHEX, LWRITEHEX Function

Syntax:

```
PROCEDURE READHEX   (VAR F : TEXT; VAR W : ANYTYPE; SIZE : l..4);
PROCEDURE WRITEHEX  (VAR F : TEXT; EXPRESSION : ANYTYPE; SIZE:
1..4);
PROCEDURE LWRITEHEX (VAR F : TEXT; EXPRESSION : LONGINT; SIZE:
l..4);
```

Explanation:

These routines read and write text in hexadecimal representation. SIZE specifies the number of bytes to read or write.

READHEX reads two characters for each byte, then it skips to the next carriage return/line feed. You cannot read more than one hexadecimal number from a single line.

WRITEHEX writes two characters for each byte. It does not output any leading or trailing blanks or a carriage return/line feed.

LWRITEHEX is like WRITEHEX, except that it only works with long integers, and it can handle up to four bytes.

The 8-bit version of Pascal/MT+ does not have LWRITEHEX, and its maximum data size for READHEX is 2 bytes.

RESET Function

Syntax:

```
PROCEDURE RESET(FILE VARIABLE);
```

Explanation:

RESET moves the window pointer to the beginning of a file so that you can read it. The window variable is set to the first element of F. If you try to reset a file that does not exist, IORESULT returns a value of 255. Any other value means success. RESET calls CLOSE RESET calls CLOSE if the file is already open.

The file is open to reading and writing for random access. With nonconsole typed files, the procedure RESET does an initial GET. This process moves the first element of the file into the window variable.

The initial GET does not perform on console or untyped files because GET waits for a character, and you would have to type a character before your program could execute.

REWRITE Function

Syntax:

```
PROCEDURE REWRITE(FILE VARIABLE);
```

Explanation:

REWRITE creates a file on disk using the name associated with the file variable, deleting any existing file by that name. If the variable has no associated filename, specified with ASSIGN, REWRITE creates a temporary file.

Temporary files are useful for scratch pad memory and data that you no longer need after executing the program. The last two digits in the name make every temporary file unique, so you can have up to 100 temporary files.

The EOF and EOLN functions return TRUE because the file is an output file. The file is open for sequential writing only and is ready to receive data into its first element.

RIM85, SIM85 Function

Syntax:

```
FUNCTION RIM85 : BYTE;
PROCEDURE SIM85(VAL : BYTE);
```

Explanation:

These routines use the special 8085 instructions RIM and SIM. They call the procedure that contains the instruction. Under CP/M, the heap grows from the end of the data area, and the stack frame (for recursion) grows from the top of memory down. CP/M preloads the hardware stack register with the contents of absolute location 0006, unless the $Z option overrides it. The stack frame grows starting at 512 bytes below the initialized hardware value.

Note: these routines are only supported in the 8-bit version of Pascal/MT+.

ROUND Function

Syntax:

```
FUNCTION ROUND(REAL)    INTEGER;
```

Explanation:

ROUND converts a real to an integer by rounding it up or down to the nearest integer value.

Examples:

```
ROUND(2.67) =  3
ROUND(45.49) =  45
```

SEEKREAD, SEEKWRITE Function

Syntax:

```
PROCEDURE SEEKREAD (F : ANYFILE; RECORD_NUM : 0..MAXINT);
PROCEDURE SEEKWRITE(F : ANYFILE; RECORD_NUM : 0..MAXINT);
```

Explanation:

These procedures support random access I/O. SEEKREAD reads from the specified record into the window variable. SEEKWRITE writes from the window variable to the specified record. You must assign to the window variable prior to a SEEKWRITE or assign from the window variable after a SEEKREAD. The records are numbered sequentially, starting with record 0.

Files written using SEEKWRITE are contiguous, regardless of the record size. A file can be accessed sequentially or randomly, but not without executing a CLOSE before changing access modes.

To use SEEKREAD and SEEKWRITE, link in the library RANDOMIO, which supports random access.

Section 7 has examples of these procedures and more information about random-access I/O.

SHL, SHR Function

Syntax:

```
FUNCTION SHL(BASIC_VAR, NUM) : INTEGER;
FUNCTION SHR(BASIC_VAR, NUM) : INTEGER;
```

Explanation:

SHR shifts BASIC_VAR by NUM bits to the right, inserting 0 bits. SHL shifts the BASIC_VAR by NUM bits to the left, inserting 0 bits. BASIC_VAR is an 8- or 16-bit variable. NUM is an integer expression.

Suppose you obtain a 10-bit value from two separate input ports. Use SHL to read them in:

```
X := SHL(INP[8] & $1F, 3) ! (INP[9] & $1F);
```

The example reads from port number 8, masks out the three high bits returned from the INP array, and shifts the result left. Next, this result logically OR's with the input from port number 9, which has also been masked.

Example:

```
PROCEDURE SHIFT_DEMO;
VAR I : INTEGER;
BEGIN
  WRITELN(‘SHIFT_DEMO.........’);
  I := 4;
  WRITELN('I=',I);
  WRITELN('SHR(I,2)=',SHR(I,2));
  WRITELN(‘SHL(I,4)=',SHL(I,4));
END;
```

Output:

```
SHIFT DEMO........
I=4
SHR(I,2)=l
SHL(I,4)=64
```

SIN Function

Syntax:

```
FUNCTION SIN(ANGLE) : REAL;
```

Explanation:

SIN returns the sine of the angle. Express the angle in radians, as an integer or real expression.

SIZEOF Function

Syntax:

```
FUNCTION SIZEOF (VARIABLE OR TYPE NAME) : INTEGER;
```

Explanation:

SIZEOF is a compile-time function that returns the size of the parameter in bytes. Use it in MOVE statements for the number of bytes to be moved. With SIZEOF you do not need to keep changing constants as the program evolves. The parameter can be any variable or user-defined ordinal type.

SIZEOF is a compile-time function. Only the size of items that do not generate code to calculate their address can be a parameter to SIZEOF. The compiler must know the size of the item.

Example:

```
PROCEDURE SIZE_DEMO;

CONST
  NAMELN = 10;
  ADDRLN = 30;

VAR
  A : RECORD
    NAME : STRING[NAMELN];
    ADDR : STRING[ADDRLN)
  END;
  B : RECORD
    NAME : STRING[NAMELN];
    ADDR : STRING[ADDRLN];
    HIRE DATE : INTEGER;
    EMP_NUM : INTEGER
  END;

BEGIN
  READLN(A.NAME);
  READLN(A.ADDR);
  B.HIRE DATE := 0;
  B.EMP_NUM := 0;

  MOVE(A, B, SIZEOF(A));      (* MOVES THE NAME AND ADDR
                                 INTO B *)
  WITH B DO
    WRITELN (NAME, ADDR, HIRE_DATE, EMP_NUM)
END;
```

In this example, if you change the value for NAMELN or ADDRLN, you do not have to change the parameters to MOVE, because the SIZEOF function always returns the current size of record A.

SQR Function

Syntax:

```
FUNCTION SQR(X) : REAL or INTEGER
```

Explanation:

SQR returns the square of X. X must be real or integer. The result has the same type as X.

Example:

```
SQR(5)   = 25
SQR(4.0) = 16.0
```

SQRT Function

Syntax:

```
FUNCTION SQRT(X) : REAL;
```

Explanation:

SQRT returns the square root of X. X must be real or integer. The result is real.

SUCC Function

Syntax:

```
FUNCTION SUCC(X) : SCALAR;
```

Explanation:

X is a scalar or subrange expression. SUCC returns the value of X's successor.

Examples:

```
SUCC('A') = ‘B’
SUCC(FALSE) = TRUE
SUCC(23) = 24
```

TRUNC Function

Syntax:

```
FUNCTION TRUNC(REAL) : INTEGER;
```

Explanation:

TRUNC converts a real number to an integer by dropping the digits to the right of the decimal point.

Examples:

```
TRUNC(4.99)         = 4
TRUNC(36.2 + 1.11) = 37
```

TSTBIT, SETBIT, CLRBIT Function

Syntax:

```
FUNCTION TSTBIT(     BASIC_VAR, BIT_NUM) : BOOLEAN;
PROCEDURE SETBIT(VAR BASIC_VAR, BIT_NUM);
PROCEDURE CLRBIT(VAR BASIC_VAR, BIT_NUM);
```

Explanation:

TSTBIT returns TRUE if the designated bit is on, and returns FALSE if the bit is off.

SETBIT sets the designated bit in the parameter.

CLRBIT clears the designated bit in the parameter.

BASIC_VAR is any 8-or 16-bit variable. BIT_NUM is 0..15 with bit 0 on the right.

If BIT_NUM is out of range, results are unpredictable but the program continues. For example, trying to set or clear bit 10 of an 8-bit variable causes unpredictable results, but no error message.

Example:

```
PROCEDURE TST_SET_CLR_BITS;

VAR
  I : INTEGER;
BEGIN
  WRITELN('TST_SET_CLR_BITS........');
  I := 0;
  SETBIT(I,5);
  IF I = 32 THEN
    IF TSTBIT(I,5) THEN
      WRITELN('I=',I);
  CLRB IT (I, 5) ;
    IF I = 0 THEN
      IF NOT (TSTBIT(I,5)) THEN
        WRITELN('I=',I);
END;
```

Output:

```
TST_SET_CLR_BITS.......
I=32
I=0
```

WAIT Function

Syntax:

```
PROCEDURE WAIT(PORTNUM , MASK, POLARITY);
```

Explanation:

The WAIT procedure is only available in the 8-bit version of Pascal/MT+. PORTNUM and MASK are literal or named constants. POLARITY is a Boolean constant. WAIT generates a tight status wait loop:

```
IN portnum
ANI mask
J?? $-4
```

The WAIT procedure does not generate in-line code for the status loop. A states loop is constructed in the DATA area and called by the WAIT run-time subroutine. Thus, the loop is fast, but the call and return from the loop add a small amount of execution time. Use INLINE if time is critical.

Example:

```
PROCEDURE WAIT_DEMO;

CONST
  CONSPORT = $F7;      (* for EXPO NOBUS-Z COMPUTER *)
  CONSMASK = $01;

BEGIN
  WRITELN('WAIT_DEMO........');
  WRITELN('WAITING FOR A CHARACTER');
  WAIT(CONSPORT,CONSMAXK,TRUE);
  WRITELN('THANKS');
END;
```

WNB, GNB Function

Syntax:

```
FUNCTION GNB(FILEVAR: FILE OF PAOC):CHAR;
FUNCTION WNB(FILEVAR: FILE OF CHAR; CH:CHAR) : BOOLEAN;
```

Explanation:

These functions give you byte-level, high-speed access to a file. PAOC is any type that is a Packed Array Of Char. The optimum size of the packed array is in the range 128..4095.

GNB lets you read a file one byte at a time. GNB returns a value of type CHAR. The EOF function is valid when the physical end-of-file is reached but not based upon any data in the file. Attempts to read past the end of the file return $FF.

WNB lets you write a file one byte at a time. WNB requires a file and a character to write. The function returns a Boolean value that is TRUE if there was an error while writing that byte to the file. Written bytes are not interpreted.

GNB and WNB are faster than using F^, GET/PUT combinations, because of their larger buffer.

WRITE, WRITELN Function

Syntax:

```
PROCEDURE WRITE  (FILE VARIABLE, EXPR, EXPR, ... );
PROCEDURE WRITELN(FILE VARIABLE, EXPR, EXPR, ... );
```

Explanation:

These procedures write data to the file associated with F. If the file is a TEXT file, they convert numbers to ASCII and write the Boolean values as the strings TRUE and FALSE.

```
WRITE(F, DATA);
```

is equivalent to

```
F^ := DATA;
PUT(F);
```

WRITELN works only with TEXT files, ending an old line and starting a new one. The procedure is like WRITE, except it puts a carriage return and line feed after the data. A WRITELN with no expressions outputs only a carriage return/line-feed.

Data can be literal and named constants, integers, reals, subranges, enumerated, Booleans, strings, and packed arrays of characters, but cannot be structured types, such as records.

If you do not specify a file, the procedures default to the standard output file.

WRITE and WRITELN treat strings as arrays of characters. They do not write the length byte to the file.

You can specify the field format for any data type. The field format is

```
<real or non-real variable> : <field width>
```

or

```
<real variable> : <field width> : <fraction length>
```

The minimum <field width>, which is optional, is a natural number that specifies the smallest number of characters to write. The optional <fraction length> specifies the number of digits to follow the decimal point in a real number. For non-real numbers, specify only the field width. The data is right-justified in the field. A number is always expressed in exponential notation if a number is larger or smaller than the significant digits can represent.

If you do not specify a <field width>, real numbers are output in exponential format, and other types are output without any extra leading or trailing blanks.

Example:

```
PROGRAM DO_WRITE;

CONST
  STR = 'COLORLESS GREEN IDEAS';
  BUL = TRUE;
  INT = 9876;
  REL = 2345.678;

VAR
  F : TEXT;
  I : INTEGER;

BEGIN
  ASSIGN(F, ‘SAMPLE.TXT');
  REWRITE(F);
  WRITE(F, '*', 1, 2, 3);
  WRITE(F, 4, 5, 6);
  WRITELN(F, '*');

  WRITELN(F, ‘2: *’, STR, ‘*’);
  WRITELN(F, ‘3: *’, STR:40, ‘*’);
  WRITELN(F, ‘4: *’, BUL, ‘*’, INT, ‘*’, REL, ‘*’);
  WRITELN(F, ‘5: *', BUL:10,'*',INT:10,'*',REL:10,'*');
  WRITELN(F, ‘6: *', REL:10:3, '*', REL:8:1, ‘*’);
  CLOSE(F, I)
END.
```

Output:

```
 *123456*
 2: *COLORLESS GREEN IDEAS*
 3: *                    COLORLESS GREEN IDEAS*
 4: *TRUE*9876* 2.34567E+03*
 5: *       TRUE*       9876* 2.3456E+03*
 6: * 2345.678*  2345.7*
```

@BDOS Function

Syntax:

```
FUNCTION @BDOS;
```

Explanation:

@BDOS enables direct access to the CP/M operating system.

See the Pascal/MT+ Language Programmer's Guide for more information.

@BDOS86 Function

Syntax:

```
FUNCTION @BDOS86;
```

Explanation:

@BDOS86 enables direct access to the CP/M-86® operating system. See the Pascal/MT+ Language Programmer's Guide for more information.

@CMD Function

Syntax:

```
FUNCTION @CMD : ^STRING;
```

Explanation:

@CMD lets you access the command tail of a command line. The function retrieves the information from the command tail, moves it to a string, and returns a pointer to this string. The command tail starts with a blank. You can call @CMD only once, at the beginning of the program before you open any files.

Example:

```
PROGRAM @CMD_DEMO;
TYPE
  PSTRG = ^STRING;

VAR S : STRING[16]
    PTR : PSTRG;
    F : FILE OF INTEGER;

EXTERNAL FUNCTION @CMD : PSTRG;

BEGIN
  PTR := @CMD;
  S := PTR^;
  ASSIGN(F,S);
  RESET(F)
END.
```

@ERR Function

Syntax:

```
PROCEDURE @ERR;
```

Explanation:

@ERR is the default error handling routine in PASLIB. You can replace @ERR with your own error handling routines. See Section 4.6.3 of the Pascal/MT+ Language Programmer's Guide for more information.

@HLT Function

Syntax:

```
PROCEDURE @HLT;
```

Explanation:

@HLT unconditionally halts your program, and returns control to the operating system. Section 7.6 contains an example of using @HLT.

@HERR Function

Syntax:

```
FUNCTION @HERR;
```

Explanation:

@HERR is a predefined BOOLEAN variable that the NEW procedure uses to return the result of an allocation request. @HERR returns FALSE if space is available, or TRUE when there is no space.

You should always use @HERR in conjunction with NEW, because the heap management system in PASLIB does not signal an error if there is no space available when you make an allocation request.

@MRK Function

Syntax:

```
FUNCTION @MRK : INTEGER;
```

Explanation:

@MRK returns the address of the top of the heap. You must save the address if you want to use @RLS to restore the heap to its previous state.

You can use @MRK to mark more than one address, and then use @RLS to return to any of them.

See Section 4.3.5 of the Pascal/MT+ Language Programmer's Guide for more information.

@RLS Function

Syntax:

```
FUNCTION @RLS (INTEGER);
```

Explanation:

@RLS resets the top of the heap to the address returned by @MRK.

See Section 4.3.5 of the Pascal/MT+ Language Programmer's Guide for more information.

End of Section 6

Section 7
Input and Output

This section describes the Pascal/MT+ I/O (input/output) system. The I/O system is hardware-independent, and allows a program to transfer data between memory and external devices such as a console, printer, or disk. Pascal/MT+ provides both sequential and random access I/O.

7.1 Fundamentals of Pascal/MT+ I/0

A file is like an open-ended array that can contain elements of any simple or structured type. The size of a file is limited by your operating system or by the capacity of your disk.

In Pascal/MT+, a file variable has two parts: a File Information Block (FIB), and a buffer.

- The File Information Block contains information about the file such as the file's name and type, whether the file is open for reading or writing, and the end-of-file and end-of-line flags. The file named FIBDEF.LIB on your distribution disk contains a complete description of the FIB.

- The buffer holds one item of the file's base type. The I/O routines read data into or write data from the buffer, and it is the only part of the file variable that you can directly access. This buffer is sometimes called the 'window variable' because you can visualize it as a window into the file.

You declare a file variable like any other variable, as in the following example:

```
TYPE
  INTFILE = FILE OF INTEGER;
  REC     = RECORD

    X, Y, Z : REAL;
    I, J, K : INTEGER
  END;

VAR
  Fl, F2  : INTFILE;
  F3      : FILE OF REC;
  F4      : FILE OF ARRAY[l..10] OF CHAR;
  F5      : FILE;     (* UNTYPED FILE FOR BLOCK I/O *)
```

When you declare a file variable, the I/O system does not associate a physical disk file with that variable. You have to use the ASSIGN or OPEN procedure to associate an actual filename with the variable. After that, all input and output to the file is through the file variable.

In general, you use the file variable's name to refer to the file. If you want to reference the buffer, follow the name with the pointer character. For example,

```
ASSIGN (F3, 'TEST.DAT' );
```

associates the name TEST.DAT with the file variable F3, and

```
F2^ := 45;
```

puts the integer value 45 in the buffer of the file variable F2.

Each file must have an explicit end-of-file indicator. Most operating systems use a control character to indicate the end-of-file. When the I/O system encounters this character, the predefined function EOF returns TRUE.

Under some conditions, however, the valid data ends before the operating system signals an end-of-file condition. This can happen, for example, when the data does not fill the last sector in the file. In this case, EOF does not detect the actual end of the Data file. Therefore, you must use a dummy record as the last record, or save the number of records in a separate file.

7.2 Regular I/O

The two basic routines for reading and writing data are GET and PUT. GET reads the next file element into the buffer. PUT writes the contents of the buffer to the next position in the file.

To write data to a file using PUT, you have to assign the data to the buffer and then call PUT as in the following sequence:

```
F^ := ITEM;
PUT(F);
```

The newly written item is the last element in the file.

To read data with GET, you take the data from the buffer and then call GET, as in the following sequence:

```
ITEM := F^;
GET(F);
```

The reason for this sequence is not intuitive. Note however, that when you call RESET to open the file for reading, the first element in the file is automatically placed in the buffer. Calling GET places the next item in the buffer.

If you are reading from the console, you have to call GET before you access the buffer, because initially there is nothing in the buffer, and the program would wait indefinitely for the first character.

The program shown in Listing 7-1 demonstrates the GET and PUT routines. The program creates a file, writes some data to it, and then reads the data back from the file. Notice that you have to explicitly move data in and out of the buffer.

You usually do not have to use GET and PUT. The procedures READ and WRITE allow you to read and write data without worrying about the buffer. Both routines can handle any filetype. You do not have to treat the console and other devices differently when you use READ and WRITE.

```
 Stm    Nest  Source Statement
    t

    1      0  PROGRAM WRITE_READ_FILE_DEMO;
    2      0
    3      0  TYPE
    4      1    CHFILE = FILE OF CHAR;
    5      1  VAR
    6      1    OUTFILE : CHFILE;
    7      1    RESULT  : INTEGER;
    8      1    FILENAME: STRING[161;
    9      1
   10      1  PROCEDURE WRITEFILE( VAR F : CHFILE);
   11      1  VAR CH : CHAR;
   12      2  BEGIN
   13      2    FOR CH  =   '0' TO '9' DO
   14      2      BEGIN
   15      3        F^ := CH;
   16      3        PUT(F)
   17      3      END;
   18      2  END;
   19      1
   20      1  PROCEDURE READFILE( VAR F : CHFILE);
   21      1  VAR I  : INTEGER;
   22      2      CH : CHAR;
   23      2  BEGIN
   24      2    FOR I := 0 TO 9 DO
   25      2      BEGIN
   26      3        CH := F^;
   27      3        GET(F);
   28      3        WRITELN(CH);
   29      3      END;
   30      2  END;
```

Listing 7-1. File Input and Output

```
 Stm    Nest  Source Statement
    t
   31      1
   32      1  BEGIN
   33      1    FILENAME := 'TEST.DAT';
   34      1    ASSIGN(OUTFILE,FILENAME);
   35      1    REWRITE(OUTFILE);
   36      1    IF IORESULT = 255 THEN
   37      1      WRITELN('Error creating ',FILENAME)
   38      1    ELSE
   39      1      BEGIN
   40      2        WRITEFILE(OUTFILE);
   41      2        CLOSE(OUTFILE,RESULT);
   42      2        IF RESULT = 255 THEN
   43      2          WRITELN('Error closing ',FILENAME)
   44      2        ELSE
   45      2          BEGIN
   46      3            WRITELN('Successful close of ',FILENAME);
   47      3            RESET (OUTFILE) ;
   48      3            IF IORESULT = 255 THEN
   49      3              WRITELN('Cannot open ',FILENAME)
   50      3            ELSE
   51      3              READFILE(OUTFILE)
   52      3          END;
   53      2      END;
   54      1  END.
```

Listing 7-1. (continued)

7.3 INP and OUT Arrays

Pascal/MT+ allows direct manipulation of input and output hardware ports through two features.

1) Two predeclared arrays, INP and OUT, of type BYTE, can be subscripted with port number constants and expressions.

   The INP array can be used only in expressions. The OUT array can be used only on the LEFT side of an assignment statement. The most significant byte of INP contains 00 if the values from INP are assigned to variables of type INTEGER.

   You can subscript these arrays with integer expressions in the range 0 to 255. Two types of syntax are used with this feature. The code is always generated in-line for INP and OUT, but always uses variable port I/O instructions.

Examples:

```
OUT[(PORTNUM + I)] := $88;
OUT[O] := $88;
J := INP[(PORTNUM)];
```

2) A function INPORT_W, and a procedure OUTPRT_W manipulate I/O ports. Although they are present in the standard library, you must declare them as:

```
EXTERNAL FUNCTION  INPORT_W (PORTNUM:INTEGER):WORD;
EXTERNAL PROCEDURE OUTPRT_W(PORTNUM:INTEGER; DATA:WORD);
```

Examples:

```
INCHAR := INPORT_W(PORTNUM);
OUTPRT_W(PORTNUM,OUTCHAR);
OUTPRT_W($004F,OUTCHAR);
```

7.4 Redirected I/O

Redirected I/O is an alternative to the GET-character and PUT-character routines in the run-time package. Redirected I/O is useful when you do not want the regular I/O from your operating system. Also, this feature works well for converting numbers into strings and strings into numbers. The sample program shown in Listing 7-2 demonstrates this application.

Pascal/MT+ has a mechanism you can use to write your own character-level I/O drivers. This facility lets a ROM-based program be system-independent. It also works with user-written character input and output routines that get their data from, or write it to, strings or I/O ports. It lets them use the conversion routines built into the system Read-Write code.

Example:

```
READ( [ ADDR(getch) ], ... );
WRITELN( [ ADDR(putch) ], ... );
```

You can write the "getch" and "putch" routines in Pascal/MT+ or in assembly language. The parameter requirements for these routines are

```
FUNCTION getch : CHAR;
PROCEDURE putch( outputch: CHAR);
```

When you use this mechanism, keep in mind the following points:

- You must show the declaration of these routines.

- The names need not be getch/putch, but the GET character routine must not have parameters, and the PUT character routine must have one parameter of type CHAR.

- You can assign the address of the procedure to a pointer using the ADDR function and then specify this pointer. For example, READ([PI , . . .) . This saves typing time, but not execution time.

Note that READLN and EOF/EOLN cannot be used with redirected I/O because EOLN and EOF both operate on files. Note also that you cannot read into STRING variables requiring the use of READLN, because READLN uses EOLN.

The reason is that the @RST (read string) routine tries to read directly from the console device when no file is specified. You can rewrite the @RST routine to perform any input and editing functions you want for the target-system console device. This does not affect programs that do not use redirected I/O.

Referring to the program in Listing 7-2, note that WIR, the PUT character routine, (line 8) writes to a global string, named CONV, and GETCH, the GET character function, (line 28) gets its character input from this global string.

The test program code begins on line 39. The first statements initialize the variables required by WIR and GETCH. CONVERTING is a Boolean value that is TRUE when WIR is writing a number to CONV. CONV is initialized to the empty string, so its length byte is 0.

On line 42, the test variable I is assigned the value 2438. Then, on line 43 the regular WRITELN statement writes it to the console.

Line 44 demonstrates the concept of redirected I/O in this program.

```
WRITELN([ADDR(WIR)],I);
```

Here, WIR's address is passed to the WRITELN routine so that WIR is used instead of the PUT character routine in the run-time package. The run-time routines convert the number I into characters that are passed to WIR for output to the string, CONV. In this way, the contents of I are converted to a string. Note that WIR must always be called with a WRITELN because it uses the carriage return to signal that the number is complete.

```
 Stm    Nest  Source Statement
    t
    1      0  PROGRAM CONV_DEMO;
    2      0
    3      0  VAR
    4      1    I : INTEGER;
    5      1    CONV : STRING;
    6      1    CONVERTING : BOOLEAN;
    7      1
    8      1  PROCEDURE WIR(CH : CHAR);
    9      1  BEGIN
   10      2    IF CH = CHR($0A) THEN (* DONE,IGNORE LINEFEED *)
   11      2      EXIT;
   12      2    IF CONVERTING THEN
   13      2      IF CH <> CHR($0D) THEN (* NOT AT END OF STRING *)
   14      2        CONV := CONCAT(CONV,CH)
   15      2      ELSE
   16      2        CONVERTING := FALSE (* REACHED END-DONE *)
   17      2    ELSE
   18      2      BEGIN
   19      3        CONV
   20      3        IF CH <> CHR($0D) THEN
   21      3          BEGIN
   22      4            CONV := CONCAT(CONV,CH);
   23      4            CONVERTING := TRUE
   24      4          END
   25      4        END;
   26      2      END;
   27      1
   28      1  FUNCTION GETCH : CHAR;
   29      1  BEGIN
   30      2    IF LENGTH(CONV) > 0 THEN (* SOMETHING LEFT TO CONVERT
               *)
   31      2      BEGIN
   32      3        GETCH := CONV[1];
   33      3        DELETE(CONV,1,1);
   34      3      END
   35      3    ELSE
   36      2      GETCH := ' ';   (* RETURN BLANK-NO MORE CHARACTERS *)
   37      2  END;
   38      1
   39      1  BEGIN            (* MAIN PROGRAM *)
   40      1    CONVERTING := FALSE;
   41      1    CONV := '';
   42      1    I := 2438;
   43      1    WRITELN('I=',I);
   44      1    WRITELN([ADDR(WIR)],I); (* FIELD WIDTH MAY BE GIVEN *)
   45      1    I := 0;
   46      1    WRITELN('I=',I);
   47      1    WRITELN('CONV=',CONV);
   48      1    READ([ADDR(GETCH)],I);  (* READLN MAY NOT BE USED *)
   49      1    WRITELN('I=',I);
   50      1  END.
```

Listing 7-2. Redirected I/0 7-7

7.5 Sequential I/0

Sequential I/O means that the I/O system accesses the data items in a file in a serial fashion. Thus, you can read the data items one after the other, and you can add items only at the end of the file.

7.5.1 TEXT Files

A TEXT file is a file of ASCII characters subdivided into lines. The predefined type TEXT is used for ASCII files. A line is a sequence of characters terminated by a nonprintable end-of-line indicator, usually a carriage return and a line-feed.

A TEXT file is similar to a file of CHAR except that numbers are automatically converted when they are read from and written to the file. Numbers written to TEXT files convert to ASCII, and can be formatted. Numbers read from TEXT files convert to binary.

TEXT files differ from files of type CHAR in the following ways:

- TEXT files are subdivided into lines.
- TEXT files accept both ARRAY[l..N] OF CHAR, and PACKED ARRAY[1..N] OF CHAR as data.
- TEXT files accept STRINGS as data.
- Boolean values convert to the ASCII sequence TRUE or FALSE on write, but TRUE or FALSE do not convert to Boolean values.
- You can access a TEXT file with GET and PUT for character I/O (which do not do conversions), READ and WRITE, and READLN and WRITELN.

The format of a TEXT file in memory is a FIB and a 1-byte window variable. Figure 7-1 illustrates the way a TEXT file appears on disk.

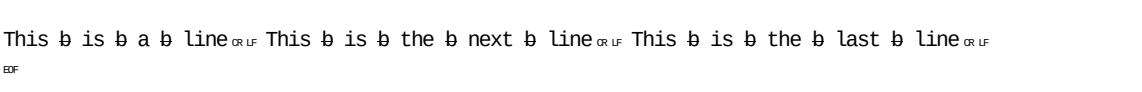


Figure 7-1. Lines in a TEXT File

The program in Listing 7-3 writes data to a TEXT file and reads it back for display on the output device. The procedure WRITE_DATA writes to the TEXT file and READ-DATA retrieves the information stored in the file.

The field format can be specified for any data type. For non-real numbers only the field width is specified, not the number of places after the decimal point. The data is right-justified in the field. The output is always expressed in exponential notation if a number is larger than the significant digits can represent. It is also written in exponential notation if the field width is too small to express the number.

The body of the WRITE_DATA procedure can be written in the following manner with the same results:

```
WRITELN (F, S);
WRITELN(F,I:4, 45.6789 : 9 : 4);
```

Referring to Listing 7-3, note that if a READLN were used on line 31, the integer value 35 would be read properly because the first blank terminates the number. However, the window variable would advance past the real number to the end of the file. Then, if you try to read the real number, you would only get the EOF.

STRINGS must always be read with a READLN because they are terminated with end-of-line characters. If the data in the file was

```
This b is b a b string 35 CR LF
```

the value returned for S would be the entire line, including the ASCII 35.

Within the READ_DATA procedure, lines 20 and 21 write the data to the console in the same format as in the file.

The main program stops after processing the call to READ_DATA on line 43. A CLOSE is not necessary because the data in TEXT.TST is not altered from the last CLOSE on that file.

```
 Stm    Nest  Source Statement
   t
 1      0     PROGRAM TEXT_IO_DEMO;
 2      0
 3      0     VAR F : TEXT;
 4      1         I : INTEGER;
 5      1         S : STRING;
 6      1
 7      1       PROCEDURE WRITE_DATA;
 8      1       BEGIN
 9      2         WRITELN(F,S);
 10     2         WRITE(F,I:4);
 11     2         WRITELN(F,45.6789:9:4);
 12     2       END;
 13     1
 14     1       PROCEDURE READ_DATA;
 15     1       VAR R : REAL;
 16     2       BEGIN
 17     2         READLN(F,S);
 18     2         READ(F,I);
 19     2         READ(F,R);
 20     2         WRITELN(S);
 21     2         WRITELN(I:4,' ',R:9:4);
 22     2       END;
 23     1
 24     1     BEGIN
 25     1       ASSIGN(F,'TEXT.TST');
 26     1       REWRITE(F);
 27     1       IF IORESULT = 255 THEN
 28     1         WRITELN('Error creating')
 29     1       ELSE
 30     1         BEGIN
 31     2           I := 35;
 32     2           S := 'THIS IS A STRING';
 33     2           WRITE_DATA;
 34     2           CLOSE(F,I);
 35     2           IF IORESULT = 255 THEN
 36     2             WRITELN('Error closing')
 37     2           ELSE
 38     2             BEGIN
 39     3               RESET(F);
 40     3               IF IORESULT = 255 THEN
 41     3                 WRITELN('Error opening')
 42     3               ELSE
 43     3                 READ_DATA;
 44     3             END;
 45     2         END;
 46     1     END.
```

Listing 7-3. TEXT File Processing

7.5.2 Writing to the printer

Listing 7-4 shows a typical way to write to the printer. The program declares a file variable of type TEXT on line 5, and then on line 11 assigns this file variable to the printer. The filename 'LST:' passed to ASSIGN means that F is associated with the list device. All data written to F routes to the printer.

Next, REWRITE is called to open the list device for writing. Lines 23 and 25 use standard Pascal formatting directives. Thus, on line 23, R is written in a field seven characters long with three digits to the right of the decimal place.

Once again, note that a CLOSE is not necessary because the data was already written and the buffer does not need to be flushed.

```
 Stm    Nest  Source Statement
    t
 1     0      PROGRAM PRINTER;
 2     0      (* WRITE DATA AND TEXT TO THE PRINTER *)
 3     0
 4     0      VAR
 5     1        F : TEXT;
 6     1        I : INTEGER;
 7     1        S : STRING;
 8     1        R : REAL;
 9     1
 10    1      BEGIN
 11    1        ASSIGN(F,'LST:');
 12    1        REWRITE(F);
 13    1        IF IORESULT = 255 THEN
 14    1          WRITELN('Error rewriting file')
 15    1        ELSE
 16    1          BEGIN
 17    2            S := 'THIS LINE IS A STRING';
 18    2            I := 55;
 19    2            R := 3.141563;
 20    2            WRITE(F,S);
 21    2            WRITE(F,I);
 22    2            WRITELN(F);
 23    2            WRITELN(F,R:7:3);
 24    2            WRITE (F,I,R) ;
 25    2            WRITE(F,I:4,R:7:3);
 26    2            WRITELN(F);
 27    2            WRITELN(F,'THIS IS THE END.')
 28    2          END
 29    2      END.
```

Listing 7-4. Writing to a Printer and Number Formatting

7.6 Random Access I/0

A random file is a typed Pascal file accessed with the random access procedures SEEKREAD and SEEKWRITE. You can randomly access any file by specifying the relative record number you want. This differs from sequential access in which you must access record 0 before record 1, and so on. In Pascal/MT+, you can randomly access up to 65,536 records.

With random files, a file that has been RESET can either be read with SEEKREAD or written to with SEEKWRITE. Sequential files, on the other hand, can be read only after a RESET. SEEKREAD can access a new file created with REWRITE after you have written data to the file.

Sequential records within a file written with SEEKWRITE are stored contiguously on the disk, regardless of the number of sectors occupied by a record. Because of this, you can access a file created using SEEKWRITE after a CLOSE and RESET using sequential access methods.

After SEEKREAD or SEEKWRITE has accessed a file, you must CLOSE the file and reopen it to access it with the sequential methods GET, READ, PUT, and WRITE.

The sample program in Listing 7-5 called RANDOM_DEMO, demonstrates random file access. This program creates or uses a record file of type PERSON. Each record in the file contains two strings: the name and the address of a person. The loop between lines 79 and 90 allows you to read any existing record with the procedure READRECS, or to write to any record with the procedure WRITEREC.

The main program begins on line 69 by asking if you want to create a file or open one. After you respond, line 78 resets the file. The repetitive loop allows reading and writing to continue until you stop it with a Q input.

In this program, note that the procedure ERRCHK checks IORESULT for errors encountered in the operating system.

The procedure READRECS asks for a record number, reads the record from the file, and writes it directly from window variable to the screen. Line 47 calls SEEKREAD and gives it the filename and record number. Line 51 writes the information.

Note that if record 0 and 2 contain data, you can attempt to read record 1, even though it contains no data. Thus, you must be careful when the system is unable to see errors in accessing unwritten records.

Note also that the window buffer works just as if it were declared like a pointer to a record type. To save the data elsewhere, you must make an assignment to a data structure of the same type as the file, in this case type PERSON. For example,

```
VAR TEMP : PERSON; .......
...TEMP := BF^;
```

The procedure WRITERECS asks you for the data it needs to fill a record of type PERSON (lines 56 through 61), and for the record number it should write (lines 62 and 63). Then on line 64, WRITERECS calls SEEKWRITE to write the data to the disk.

Figure 7-2 shows how the file looks after writing data to records 0, 1, and 3.

Smith, John Monterey Record 0	Brown, Susan Pacific Grove Record 1	~~bbbbbb~~ ..... Record 2	Jones, Alan Carmel Record 3

Figure 7-2. Records in a File

```
 Stm    Nest  Source Statement
    t
 1     0
 2     0      PROGRAM RANDOM_DEMO;
 3     0
 4     0      TYPE
 5     1      PERSON = RECORD
 6     1                 NAME : STRING;
 7     1                 ADDRESS : STRING;
 8     1               END;
 9     1
 10    1      VAR
 11    1        BF : FILE OF PERSON;
 12    1        S : STRING;
 13    1        I : INTEGER;
 14    1        ERROR : BOOLEAN;
 15    1        CH : CHAR;
 16    1
 17    1      EXTERNAL PROCEDURE @HLT;
 18    1
 19    1      PROCEDURE HALT;
 20    1      BEGIN
 21    2        CLOSE(BF,I);
 22    2        @HLT
 23    2      END;
 24    1
 25    1      PROCEDURE ERRCHK;
 26    1      BEGIN
 27    2        ERROR := TRUE;         (* DEFAULT *)
 28    2        CASE IORESULT OF
 29    2          0 : BEGIN
 30    4                WRITELN (‘SUCCESSFUL');
 31    4                ERROR := FALSE;
 32    4              END;
 33    3          1 : WRITELN('READING UNWRITTEN DATA');
 34    3          2 : WRITELN('CP/M ERROR');
 35    3          3 : WRITELN('SEEKING TO UNWRITTEN EXTENT');
 36    3          4 : WRITELN('CP/M ERROR');
 37    3          5 : WRITELN('SEEK PAST PHYSICAL END OF DISK');
 38    3              ELSE
 39    3                WRITELN('UNRECOGNIZABLE ERROR CODE
              :',IORESULT);
 40    3        END;
 41    2      END;
 42    1
```

Listing 7-5. Random File I/O

```
 43    1      PROCEDURE READRECS;
 44    1      BEGIN
 45    2        WRITE('RECORD NUMBER ?’);
 46    2        READLN(I);
 47    2        SEEKREAD(BF,I);
 48    2        ERRCHK;
 49    2        IF ERROR THEN
 50    2          EXIT;
 51    2        WRITELN(BF^.NAME,'/',BF^.ADDRESS);
 52    2      END;
 53    1
 54    1      PROCEDURE WRITERECS;
 55    1      BEGIN
 56    2        WRITE(‘NAME?');
 57    2        READLN(S);
 58    2        BF^.NAME := S;
 59    2        WRITE('ADDRESS?');
 60    2        READLN(S);
 61    2        BF^.ADDPESS := S;
 62    2        WRITE('RECORD NUMBER?');
 63    2        READLN(I);
 64    2        SEEKWRITE(BF,I);
 65    2        ERRCHK;
 66    2      END;
 67    1
 68    1      BEGIN
 69    1        WRITE('CREATE FILE?');
 70    1        READLN(S);
 71    1        IF S[1] IN [‘Y','y'] THEN
 72    1          BEGIN
 73    2            ASSIGN(BF,'BIG.FIL');
 74    2            REWRITE(BF);
 75    2            CLOSE(BF,I);
 76    2          END;
 77    1        ASSIGN(BF,'BIG.FIL');
 78    1        RESET(BF);
 79    1        REPEAT
 80    2          WRITE('R)EAD,W)RITE OR Q)UIT? ‘);
 81    2          READ(CH);
 82    2          WRITELN;
 83    2          CASE CH OF
 84    2            ‘R',’r’ : READRECS;
 85    3            ‘W’,’w’ : WRITERECS;
 86    3            ‘Q’,’q’ : HALT
 87    3                      ELSE
 88    3                        WRITELN(‘ENTER R, W OR Q ONLY')
 89    3          END
 90    2        UNTIL FALSE;
 91    1      END.
```

Listing 7-5. (continued)

Appendix A
Reserved Words and Predefined Identifiers

Pascal/MT+ Reserved Words

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

Pascal/MT+ Predefined Identifiers

@BDOS	CLRBIT	INSERT	PRED	SQRT
@BDOS86	CONCAT	INTEGER	PURGE	STRING
@CMD	COPY	IORESULT	PUT	SUCC
@ERR	COS	LENGTH	READ	SWAP
@HERR	CREATE	LO	READHEX	TEXT
@HLT	CSP	LONG	READLN	TRUE
@MRK	CSPF	LWRITEHEX	REAL	TRUNC
@RLS	DELETE	MAXAVAIL	RESET	TSTBIT
ABS	DISPOSE	MAXINT	REWRITE	WAIT
ADDR	EOF	MEMAVAIL	RIM85	WNB
ARCTAN	EOLN	MOVE	ROUND	WORD
ASSIGN	EXIT	MOVELEFT	SEEKREAD	WRD
BLOCKREA	EXP	MOVERIGHT	SEEKWRITE	WRITE
BLOCKWRI	FALSE	NEW	SETBIT	WRITEHEX
BOOLEAN	FILLCHAR	ODD	SHL	WRITELN
BYTE	GET	OPEN	SHORT	XIO
CHAIN	GNB	OPENX	SHR	XLONG
CHAR	HI	ORD	SIM85	
CHR	INLINE	OUT	SIN	
CLOSE	INP	PAGE	SIZEOF	
CLOSEDEL	INPUT	POS	SQR	

End of Appendix A

Appendix B
Pascal/MT+ Syntax

Backus-Naur Form (BNF) notation uses the following conventions:

- ::= The expression on the right of this symbol defines the item on the left. You can pronounce the symbol 'is rewritten as" or 'is defined as.'
- | A vertical bar indicates a choice between the items it separates. Pronounce it 'or.'
- { } Items within braces are optional. They can be repeated 0 or more times.
- < > Items within angle brackets are self explanatory or further defined in the syntax specifications.
- Items not in angle brackets are literal; enter them as they appear.

For example,

```
<identifier> ::= <letter> {<letter> | <digit> | _ }
```

states that an identifier is a letter followed by 0 or more letters, digits, or underscores.

End of Appendix B

Appendix C
Differences From ISO Standard

The following list summarizes the additions to ISO standard Pascal that are implemented in Pascal/MT+.

- Additional predefined scalars: BYTE, WORD, LONGINT, STRING.
- Expressions can contain input from I/O ports.
- Assignments can be made to I/O ports.
- Operators on integers & (and), !,| (or), and ~,\,? (not).
- CASE drops through on no match.
- ELSE on CASE statement.
- Interrupt, External, and Assembly Language procedures.
- BCD fixed point and binary floating-point reals.
- Long and short INTEGER data types.
- Modular compilation facilities.
- Redirectable I/O facilities (user written character I/O).
- Additional built-in procedures and functions:

  - bit and byte manipulation,
  - fast file I/O,
  - random file access,
  - move and fill procedures,
  - address and size of functions,
  - string manipulation,
  - heap management facilities.

The following list summarizes the differences between ISO standard Pascal and Pascal/MT+.

- Identifiers are significant in only the first 8 characters.
- Variables are not packed at the bit level.
- The order of declarations can vary.
- The null string is allowed.
- CHAR is not implemented as ISO string because variable-length strings are supported (PACKED ARRAY [1..n] OF CHAR).

End of Appendix C

Appendix D
Bibliography

Conway, Richard, David Gries, and E. Carl Zimmerman. A Primer on Pascal. Cambridge, Massachusetts: Winthrop Publishers, 1976.

Draft Proposal ISO/DP 7185; Programming Languages-Pascal. Can be obtained from American National Standards Institute, International Sales Department, 1430 Broadway, New York, New York, 10018.

Not designed for the novice. A precise language definition.

Findlay, William, and David A. Watt. PASCAL: An Introduction to Methodical Programming. Potomac, Maryland: Computer Science Press, 1978.

Grogono, Peter. Programming in Pascal. Reading, Massachusetts: Addison-Wesley, 1978.

A good introduction for self-teaching.

Jensen, Kathleen, and Niklaus Wirth. Pascal User Manual and Report. New York: Springer-Verlag, 1974.

First definition of Pascal. Best used as a reference document

Miller, Alan R. Pascal Programs for Scientists and Engineers. Berkeley, Ca.: Sybex, Inc. 1981.

Wilson, I.R. and A.M. Addyman. A Practical Introduction to Pascal. New York: Springer-Verlag, 1979.

Advanced textbook.

End of Appendix D

Index

S

T

U

V

W

Reader Comment Card

We welcome your comments and suggestions. They help us provide you with better product documentation.

Date _____

1. What sections of this manual are especially, helpful?

2. What suggestions do you have for improving this, manual? What information is missing or incomplete? Where are examples needed?

3. Did you find errors in this manual? (Specify section and page number.)

Pascal MT +™ Language Reference Manual
First Edition: February 1983
3024-2033