# Pascal/MT+™

# DIGITAL RESEARCH INC.
# END USER PROGRAM LICENSE AGREEMENT

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THE DISKETTE PACKAGE. OPENING THE DISKETTE PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED AND YOUR MONEY WILL BE REFUNDED.

Title to the media on which the program is recorded and to the documentation in support thereof is transferred to you, but title to the program is retained by DRI. You assume responsibility for the selection of the program to achieve your intended results, and for the installation, use and results obtained from the program.

1. DEFINITIONS – In this License Agreement, the terms:

A   DRI means DIGITAL RESEARCH (CALIFORNIA) INC., P.O. BOX 579, Pacific Grove, California 93950, owner of the copyright in, or authorised licensor of, the program.

B.   Machine means the single microcomputer on which you use the program. Multiple CPU systems require additional licenses.

C   AUTHOR means any third party author and owner of the copyright in this program.

D.   Runtime Library means the set of copyrighted DRI language subroutines, provided with each language compiler, a portion of which must be linked to and become part of your program for that program to run on the computer.

2. LICENSE

You may:

a)   use the program on a single machine;

b)   copy the programme into any machine-readable or printed form for backup or modification purposes only in support of such use. (Certain programs, however, may include mechanisms to limit or inhibit copying. They are marked "copy protected") Copying of documentation and other printed materials is prohibited;

c)   modify the program and/or merge it into another program for your use on the single machine. (Any portion of this program merged into another program will continue to be subject to the terms and conditions of this Agreement); and,

d)   transfer the program and license to another party if you notify DRI of name and address of the other party and the other party agrees to a) accept the terms and conditions of this Agreement, b) pay the then current transfer fee. If you transfer the program, you must at the same time either transfer all copies, including the original, whether in printed or in machine-readable form to the same party or destroy any copies not transferred, this includes all modifications and portions of the program contained or merged into other programs.

You must reproduce and include the copyright notice on any copy, modification or portion merged into another program.

YOU MAY NOT USE, COPY, MODIFY, OR OTHERWISE TRANSFER THE PROGRAM, OR ANY COPY, MODIFICATION OR MERGED PORTION, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE AGREEMENT.

IF YOU TRANSFER POSSESSION OF ANY COPY, MODIFICATION OR MERGED PORTION OF THE PROGRAM TO ANOTHER PARTY, YOUR LICENSE IS AUTOMATICALLY TERMINATED.

3. TERM

The license is effective until terminated. You may terminate it at any other time by destroying the program together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the program together with all copies, modifications and merged portions in any form.

4. COMPOSITE PROGRAMS

As an exception to Paragraph 2, you are granted the right to include portions of the DRI Runtime Library in your developed programs, called Composite Programs, and to use, distribute and license such Composite Programs, to third parties without payment of any further license fee. You shall, however, include in such Composite Program, and on the exterior label of every diskette, a copyright notice in this form: "Portions of this program, 1984 Digital Research Inc." In cases where such Composite Program is contained in Read-Only-Memory (ROM) chips, a copyright notice in the form listed above, must be displayed on the exterior of the chips and internally in the chip (in ASCII literal form). As an express condition to use of the Runtime Library, you agree to indemnify and hold DRI harmless from all claims by you and third parties arising out of the use of Composite Programs.

5. LIMITED WARRANTY AND DISCLAIMER OF WARRANTY

DRI warrants the media on which the program is furnished to be free from defects in materials and workmanship under normal use for under ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

DRI warrants that each program which is designated by DRI as warranted in its program specifications, supplied with the program, will conform to such specifications provided that the program is properly used on a machine for which it was designed. If you believe that there is a defect in a warranted program such that it does not meet its specifications, you must notify DRI within the warranty period and in the manner set forth in the program specifications.

ALL OTHER PROGRAMS ARE PROVIDED "AS IS" AND NEITHER DRI, ITS DISTRIBUTORS OR DEALERS, NOR AUTHOR MAKE ANY WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT DRI OR AUTHOR) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

Neither DRI, its Distributors or Dealers, nor AUTHOR warrant that the functions contained in any program will meet your requirements or that the operation of the program will be uninterrupted or error free or that program defects will be corrected.

THE FOREGOING WARRANTIES ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND MAY ALSO HAVE OTHER RIGHTS WHICH VARY BY STATE OR JURISDICTION.

THE PROVISIONS OF THIS AGREEMENT DO NOT AFFECT YOUR STATUTORY RIGHTS.

## 6. LIMITATIONS OF REMEDIES

DRI's entire liability and your exclusive remedy shall be as follows:
1. With respect to defective media during the warranty period:
   a. DRI will replace media not meeting DRI's "Limited Warranty" if returned to DRI or a DRI authorized representative with a copy of your receipt.
   b. In the alternative, if DRI or such DRI authorized representative is unable to deliver replacement media free of defects in materials and workmanship, you may terminate this Agreement by returning the program and your money will be refunded.
2. With respect to warranted programs, in all situations involving performance or nonperformance during the warranty period, your remedy is (a) the correction or bypass by DRI of program defects, or (b) if, after repeated efforts, DRI is unable to make the program operate as warranted, you shall be entitled to a refund of the money paid or to recover actual damages to the limits set forth below.
For any other claim concerning performance or nonperformance by DRI pursuant to, or in any other way related to, the warranted

programs under this Agreement, you shall be entitled to recover actual damages to the limits set forth below.

IN NO EVENT WILL DRI, ITS DISTRIBUTORS OR DEALERS OR AUTHOR BE LIABLE FOR ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE ANY PROGRAM OR OTHERWISE, EVEN IF DRI, ITS DISTRIBUTORS OR DEALERS OR AUTHOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES AND JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

DRI, it's Distributors' or Dealers' liability to you for actual damages for any cause whatsoever, and regardless of the form of action, shall be limited to the greater of five thousand dollars ($5,000.00) or the money paid for the program that caused the damages or that is the subject matter of, or is directly related to, the cause of action.

## 7. SUPPORT SERVICE

Support Service from DRI, if any, will be described in program specifications or in the statement of service, supplied with the program, if there are no program specifications.

## 8. GENERAL

You may not sublicense, rent or lease this program. Any attempt to sublicense, rent or lease or, except as expressly provided in this Agreement, to transfer any of the rights, duties or obligations hereunder is void.
This Agreement will be construed under the Uniform Commercial Code of the State of California.

# DIGITAL RESEARCH™

Dear Pascal/MT+ Customer,

Thank you for buying this implementation of Digital Research's
Pascal/MT+ 80 compiler for the Amstrad range of CP/M Plus computers.
The enclosed disk contains everything you need to create and debug
Pascal applications.  Please take the time to read these notes which
should get you off to a flying start.

Pascal/MT+ is available for most CP/M, CP/M Plus, CP/M-86, MS-DOS and
PC DOS computers.  The manuals are therefore not specific to the
Amstrad machines and you may be sure your applications will run on many
different computers with, at most, a recompilation for the 8086.
These notes provide specific instructions for Amstrad users.

In order to get the entire language onto one diskette, we have changed
slightly the files on sides A and B compared to the description in the
manual.   We moved MTERRS.TXT, DBUGHELP.TXT and INDEXER.PAS to side A,
and we removed all files requiring the AMD 9511 chip from side B as this
chip is not available on your computer.  Apart from these changes,
side A corresponds to what the Programmer's Guide calls Disk 1 and side
B to Disk 2.

If you have a two-drive system, you should make work disks as described
in Chapter 1 of the Programmer's Guide.  The manual can be followed
exactly for two-drive systems.  We recommend that you always use CP/M
Plus which makes disk swapping very easy.

If you have a single drive system, you may of course make two work disks
as described in the manual, and rely on the operating system to prompt
you when to change disks.  However, you will probably find it more
convenient to put your source code on the same disk as the compiler and
then transfer the compiled object code to the linker disk before
linking it.  In this case your compiler work disk should include
PIP.COM and ED.COM or your favourite text editor, and the linker disk
should contain SID.COM if you wish to use this as well as or instead of
the Pascal debugger.

You may like to know that we also supply our CBasic Compiler for the Amstrad computers.  This language is easy to learn and use and has powerful commercial and graphic extensions, but is not perhaps as flexible as Pascal.

We hope you find Pascal/MT+ enjoyable and useful.  Many first-class professional products have been created with it.  Yours could be the next.

Yours sincerely,



DIGITAL RESEARCH (UK) LTD

# DIGITAL RESEARCH®

# Pascal/MT+™
### Language
### Reference Manual

# DIGITAL RESEARCH®

# Pascal/MT+™
## Language
## Reference Manual

# 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 on 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 <u>Pascal/MT+ Language Programmer's Guide</u> 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 to 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.

# Foreword

The Pascal/MT+ language is a full implementation of standard Pascal as set forth in the International Standards Organization (ISO) standard DPS 7/185. 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 microputers 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 on 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's 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 to 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.1.

# 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-struc'ure of Pascal/MT+.
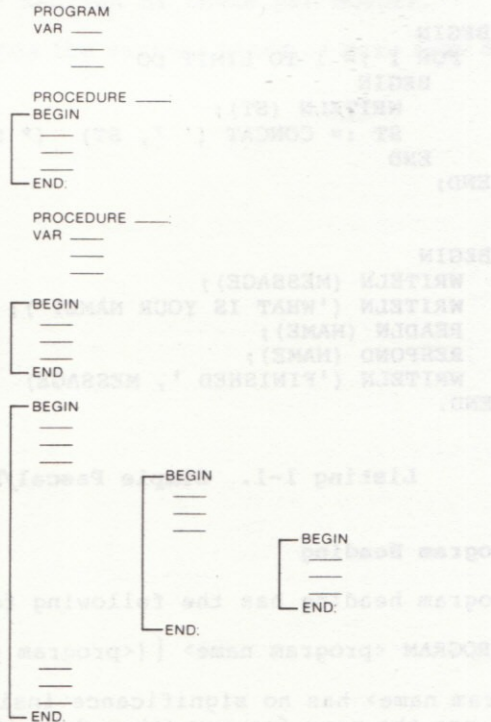


**Figure 1-1.  Block-structure in Pascal/MT+**

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 proceed 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       = $1B;
    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 [1..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(0) 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 [1..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 [1..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[I] IN LOWER THEN (* MAKE UPPERCASE *)
           LINE[I] := 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 <u>variant,</u> or <u>nonvariant.</u>  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 identifer. Outside the record, the field names can be used for different identifers. 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**

Pascal/MT+ provides ......... .... ....... 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 |

field has a variant part, it must be the last field in the list. The <variant field name> in different variants can 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 case label, use empty parentheses.

The <case selector> is either a <type name> or a <field 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 is 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 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[10];
           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 : EMP TYPE OF
               SELF : (YEARS : INTEGER);
               GOV, BUS : (TITLE : STRING[10];
                           HIRING : 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 : (E, F       : INTEGER);
    3 : (G          : LONGINT)
  END;
```

# 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,

$$4 - 3 + 1 = 2 \quad \text{whereas} \quad 4 - (3 + 1) = 0$$

**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 or set inclusion | scalar or string set | boolean boolean | lowest lowest |
| >= | greater or equal or set inclusion | scalar or string (see 4.4) | boolean boolean | lowest 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.

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]
[1..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]

          [1..20] + [3, 5, 11..34] = [1..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+1]) OF
LESS    : ; (* DO NOTHING *)
SAME    : DUPLICATES := DUPLICATES + 1;
GREATER :
    BEGIN
        SWITCHED := SWITCHED + 1;
        INTERCHANGE(N[I], N[I+1])
    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> <u>increments</u> 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 <u>before</u> 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 intitially 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 F (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 <u>formal</u> parameters.  The parameters in the procedure call are called <u>actual</u> 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 <u>address</u> 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 [1..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
  A1 : ARRAY [1..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(A1);      (* PASS A1 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  @CMD | | 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[1..10] OF CHAR;
          P : ^INTEGER;

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

## ARCTAN Function

<u>Syntax:</u>

        FUNCTION ARCTAN(X);

<u>Explanation:</u>

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

<u>Example:</u>

        ARCTAN(1) = 0.  (* THE ANGLE IS PI / 4 *)

## ASSIGN Function

<u>Syntax:</u>

        PROCEDURE ASSIGN( FILE, NAME );

<u>Explanation:</u>

        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.<br><br>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(S1,'/',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,1,POS('g',LONG_STR)-1);
          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 | F |
| 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)

<u>**EXIT Function**</u>

<u>Syntax:</u>

        PROCEDURE EXIT;

<u>Explanation:</u>

        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.

<u>Example:</u>

        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;

<u>Output:</u>

        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

<u>Syntax:</u>

        PROCEDURE FILLCHAR( DESTINATION, LENGTH, CHARACTER);

<u>Explanation:</u>

        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.

<u>Example:</u>

        PROCEDURE FILL_DEMO;

        VAR
            BUFFER : PACKED ARRAY[1..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)=1
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.

<u>**INSERT Function**</u>

<u>Syntax:</u>

        PROCEDURE INSERT(SOURCE, DESTINATION, INDEX);

<u>Explanation:</u>

     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.

<u>Example:</u>

        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;

<u>Output:</u>

        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

<u>Syntax:</u>

        FUNCTION ODD(INTEGER) : BOOLEAN;

<u>Explanation:</u>

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

<u>Example:</u>

```
        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

<u>Syntax:</u>

        FUNCTION ORD(SCALAR) : INTEGER;

<u>Explanation:</u>

    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.

<u>Example:</u>

        FUNCTION DIG2DEC (C : CHAR) : INTEGER;

        (* C MUST BE IN THE RANGE '0' ..'9' *)

        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

<u>Syntax:</u>

        PROCEDURE PUT(FILE VARIABLE);

<u>Explanation:</u>

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

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 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

<u>SEEKREAD, SEEKWRITE</u> Function
_____

<u>Syntax:</u>

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


<u>Explanation:</u>

        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)=1
     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, HIR_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

<u>Syntax:</u>

        FUNCTION SQR(X) : REAL or INTEGER

<u>Explanation:</u>

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

<u>Example:</u>

        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);
  CLRBIT(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 status 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 <u>Pascal/MT+ Language Programmer's Guide</u> 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/O

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 <u>one</u> 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
  F1, F2 : INTFILE;
  F3     : FILE OF REC;
  F4     : FILE OF ARRAY[1..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 <u>first</u> element in the file is automatically placed in the buffer.  Calling GET places the <u>next</u> 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.

```
Stmt  Nest     Source Statement

   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[16];
   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

Stmt  Nest     Source Statement

```
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[0] := $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([P],...). 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.

```
Stmt  Nest   Source Statement

  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/O

## 7.5 Sequential I/O

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[1..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.

This ∅ is ∅ a ∅ line ◯ △   This ∅ is ∅ the ∅ next ∅ line ◯ △   This ∅ is ∅ the ∅ last ∅ line ◯ △ ◡

**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 ∅ is ∅ a ∅ string   35 ⊙ △

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.

```
           Stmt   Nest   Source Statement

             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     1     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 IORE7SULT = 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.

| Stmt | Nest | Source Statement |
|------|------|------------------|
| 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/O

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<br>Monterey | Brown, Susan<br>Pacific Grove | ␣␣␣␣␣␣ . . . .<br>. . . . . . . . . . | Jones, Alan<br>Carmel |
|---|---|---|---|
| Record 0 | Record 1 | Record 2 | Record 3 |

**Figure 7-2.  Records in a File**

```
   Stmt  Nest   Source Statement

     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^.ADDRESS := 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)**

End of Section 7

# Appendix A
# Reserved Words and Predefined Indentifiers

## 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 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 |
| DIV | GOTO | NOT | SET | |
| DOWNTO | IF | OF | THEN | |
| ELSE | IN | OR | TO | |

## Pascal/MT+ Predefined Identifiers

| | | | | |
|---|---|---|---|---|
| $BDOS | CLRBIT | INSERT | PRED | SORT |
| $BDOSR | CONCAT | INTEGER | PUNCH | STRING |
| $BOND | COPY | IORESULT | PUT | SUCC |
| $BHER | COS | LENGTH | READ | SWAP |
| $BHER | CREATE | LN | READHEX | TEST |
| $PHIT | CSP | LONG | READER | TELL |
| $PMRK | CSPF | LWRITEHEX | REAL | TRUNC |
| $HLS | DELETE | MAXAVAIL | RESET | TSTBIT |
| ABS | DISPOSE | MAXINT | REWRITE | WAIT |
| ADDR | EOF | MEMAVAIL | RIM85 | WNB |
| ARCTAN | EOLN | MOVE | ROUND | WORD |
| ASSIGN | EXIT | MOVELEFT | SEEKREAD | WRD |
| BLOCKREAD | EXP | MOVERIGHT | SEEKWRITE | WRITE |
| BLOCKWRITE | FALSE | NEW | SETBIT | WRITEHEX |
| BOOLEAN | FILLCHAR | ODD | SHL | WRITELN |
| BYTE | GET | OPEN | SHORT | I/O |
| CHAIN | GNB | OPENX | SHR | XIOMS |
| CHAR | HI | ORD | SIN | |
| CHR | INLINE | OUT | SIZE | |
| 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.

<div align="center">End of Appendix B</div>

# 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 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).
* LASR 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. <u>A Primer on Pascal.</u> Cambridge, Massachusetts: Winthrop Publishers, 1976.

<u>Draft Proposal ISO/DP 7185; Programming Languages-Pascal.</u> 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. <u>PASCAL: An Introduction to Methodical Programming</u>. Potomac, Maryland: Computer Science Press, 1978.

Grogono, Peter. <u>Programming in Pascal.</u> Reading, Massachusetts: Addison-Wesley, 1978.

   A good introduction for self-teaching.

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

   First definition of Pascal.  Best used as a reference document.

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

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

   Advanced textbook.

End of Appendix D

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

Draft Proposal ISO/DP 1863 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

TSTBIT, 6-64
WNB, 6-66

**G**

garbage collection, 6-37
global,
  declaration, 1-5
  scope, 2-2
GOTO statement, 5-2, 5-5

**H**

hardware ports, 7-5, 7-6
heap, 6-37, 6-54
hexadecimal integer, 2-2

**I**

identifier, 1-3, 1-5, 2-1, 3-11
IF statement, 5-6
indexes for arrays, 3-4
inner block, 1-5
INP,
  predeclared array, 7-5
input/output routines, 6-9
INTEGER,
  data type
  literal, 2-2, 3-4
internal data representation,
  3-4, 3-5

**L**

label,
  on a GOTO statement, 5-5
  on CASE statements, 5-2
least-significant bit, 3-3
length of identifiers, 2-1
LENGTH,
  predefined function, 3-8
local declaration, 1-5
logical expressions, 4-5
logical operators, 4-2
  AND, OR, one's complement NOT,
    4-5
long integer, 2-2
long integer literal, 2-3
LONGINT,
  data type, 3-4
  literal

**M**

main program block, 1-1, 1-4
main variant, 3-12

members of a set, 4-6, 5-1
miscellaneous functions, 6-10
MOD operator, 4-3
MODEND,
  reserved word, 1-5
module, 1-4
MODULE,
  reserved word, 1-5
mutually recursive procedures,
  6-1

**N**

named constant, 2-3, 3-6, 6-38,
  6-65
named constants, 6-67
native machine word, 3-3
natural logarithm, 6-28, 6-36
nested block, 1-1, 1-5,
  procedure, 6-12
  variants, 3-11, 6-40
nested WITH statements, 5-9
nesting comments, 1-6
NIL,
  pointer value, 3-6
nonvariant record, 3-10
NOT,
  Boolean operator, 4-4
null pointer, 3-6
numeric literal, 2-2

**O**

ODD,
  pseudo-function, 3-3
one's complement NOT, 4-5
operator, 4-1
  arithmetic, 4-1
  assignment, 5-1
  Boolean, 4-2
  logical, 4-2
  precedence, 4-1
  relational, 4-2
  set, 4-3
OR,
  Boolean operator, 4-4
ORD,
  pseudo-function, 3-3
ordinal,
  data types, 3-2, 3-5
  type, 3-2, 3-6, 3-9, 5-2, 5-4,
  value, 3-6, 6-43, 6-47
ordinal value of FALSE, 3-3
ordinal value of TRUE, 3-3
OUT,

**R**

random access I/O, 6-56, 7-1, 7-13
random record number, 7-13
real number, 2-2, 3-5
real-number literal, 2-3
record,
  data type, 3-10
recursive procedures, 6-1
redirected I/O, 7-5, 7-6
relational operators, 4-2, 4-3
relational operators on sets
  IN, =, <>, <=, >=, 4-6
remainder, 4-3
REPEAT statement, 5-7
reserved word PACKED, 3-7
reserved word,
  DO, 5-3
  PACKED, 3-7
reserved words, 2-2
run-time entry points, 2-2

**S**

scalar data type, 3-1, 5-4
scalar type, 6-43, 6-62
scientific notation, 2-3
scope, 1-5, 2-2, 5-5, 6-12
  global, 1-5
  local, 1-5
  of a CASE statement, 5-2
  of a control variable, 5-4
semicolon
  as a valid statement, as a
    statement separator, 5-3
  statement separator, 1-4
sequential I/O, 7-1, 7-9
set constructor, 4-5
set expressions, 4-3, 4-5
set operations,
  union, intersection,
    difference, 3-9
set operators, 3-9, 4-3
  +, *, -.pp, 4-6
set type definition, 3-9
sets,
  data type, 3-9
short data type, 3-4
simple data type, 3-1, 5-4
simple type, 4-3, 7-1
sine of an angle, 6-58
square root of a number, 6-61
statements
  assignment, 5-1
statements, 1-4

CASE, 5-2
compound, 5-1
empty, 5-3
FOR, 5-3
FOR DOWNTO, 5-4
GOTO, 5-2, 5-5
IF, 5-6
Pascal, 5-1
REPEAT, 5-7
WHILE, 5-8
WITH, 5-8
string,
  handling routines, 6-10
  indexing, 3-8
  literal, 2-3, 3-9, 6-20
  static, 2-9, 3-9
  zero-length, 6-20
STRING,
  predefined data type, 3-8
strong type check, 3-2, 3-6
structured data types, 3-1
  arrays, records, sets, files,
    3-7
structured type, 6-67, 7-1
subrange, 3-5, 3-6
subrange data types, 3-2, 3-9,
    5-1, 6-62, 6-67
successor of a scalar, 6-62
syntax, 5-1

**T**

TEXT file, 6-14, 6-50, 6-67, 7-9
transfer functions, 6-10
TRUE,
  BOOLEAN value, 3-3
  FALSE, Boolean values, 5-6, 5-7,
    5-8
type checking, 3-2, 6-38, 6-50
type conversion, 3-2
type conversion functions,
  FUNCTION SHORT, 3-4
  FUNCTION LONG, 3-4
  FUNCTION XLONG, 3-4
type conversion operator, 3-2
type definition, 3-1
  nonvariant record, 3-11
  variant record, 3-12

# DIGITAL RESEARCH®

## Pascal/MT+™
### Language

## Programmer's Guide
### for the CP/M®
### Family of Operating Systems

# Foreword

The Pascal/MT+™ language is a full implementation of standard Pascal as set forth in the International Standards Organization (ISO) standard DPS/7185. Pascal/MT+ also has several additions to standard Pascal that increase its power to develop high quality, efficiently maintainable software for microprocessors. 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+ system, which runs under any of the CP/M® family of operating systems on an 8080, 8085, or Z80® -based microcomputer with at least 48K bytes of memory. The manual tells you how to use the compiler, linker, and the other Pascal/MT+ programming tools. Also included are topics related to the operating system for your particular implementation.

For information about the Pascal/MT+ language, refer to the Pascal/MT+ Language Reference Manual.

iii

# Table of Contents

# Table of Contents
## (continued)

# Appendixes

# Tables, Figures, and Listings

# Tables, Figures, and Listings
## (continued)

# Section 1
# Getting Started with Pascal/MT+

The Pascal/MT+ system includes a compiler, a linker, a large library of run-time subroutines, and other programming tools to help you build better programs faster.  The programming tools are

- DIS8080™, a disassembler
- LIBMT+™, a software library-building utility
- a dynamic debugger

The Pascal/MT+ system runs under any of the CP/M family of operating systems on an 8080, 8085, or Z80-based computer.  The compiler and linker need at least 48K bytes of memory to run.  To handle larger programs, they both need more memory.

The size of a program developed with Pascal/MT+ depends on the size of the source code, and on the number of run-time subroutines it uses.  Typically, the minimum size of a simple program is about 8K bytes.

Figure 1-1 illustrates the software development process using the Pascal/MT+ system.
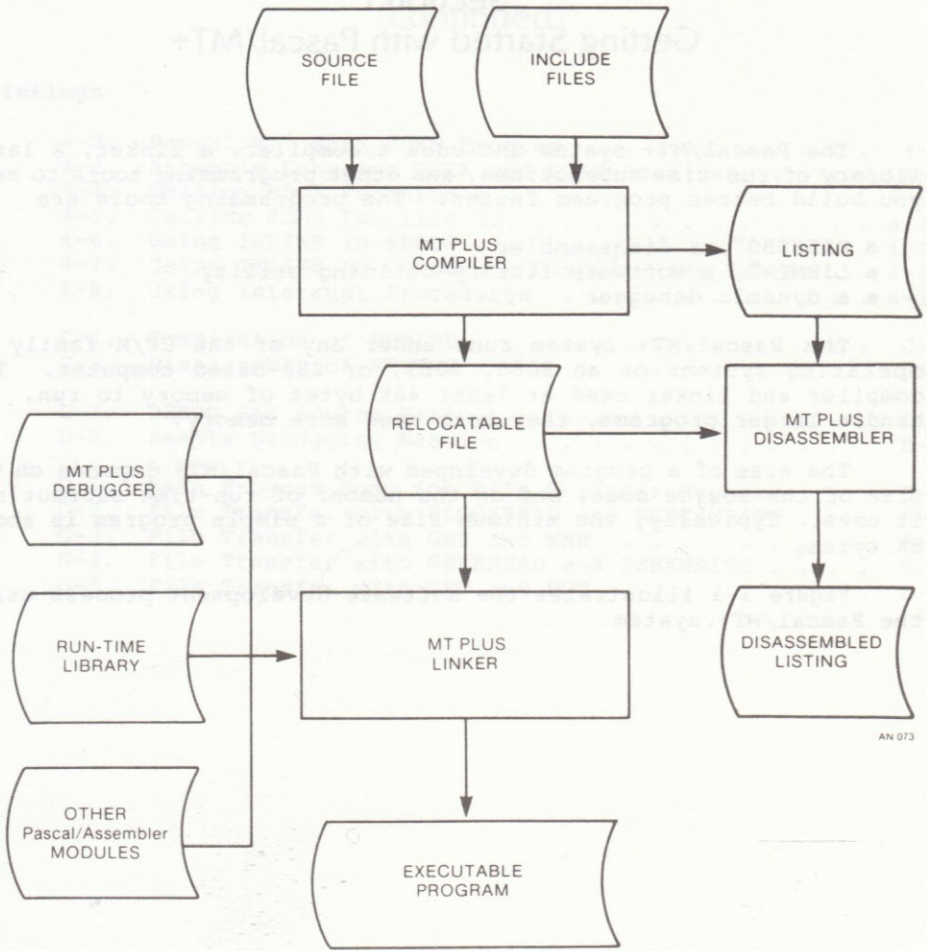
Figure 1-1.  Software Development Under Pascal/MT+

## 1.1  Pascal/MT+ Distribution Disks

The Pascal/MT+ system is supplied on three separate disks. These disks contain a number of files of different types.  Table 1-1 describes the filetypes used in the Pascal/MT+ system.  Table 1-2 briefly describes the contents of each distribution disk.

Table 1-1.  Pascal/MT+ System Filetypes

| Filetype | Contents |
|---|---|
| BLD | Build file; input file used by LIBMT+ |
| COM | Command file; directly executable under CP/M |
| CMD | Linker input command file |
| DOC | Document file; contains printable text in ASCII form |
| ERL | Relocatable object file; contains relocatable object code generated by the compiler |
| ERR | Error message file output by compiler |
| LIB | Library file; contains subroutines |
| MAC | Assembly language source file for RMAC |
| PAS | Pascal source file; contains source code in ASCII form (the compiler also accepts SRC as a source filetype) |
| PRN | Print file output by compiler |
| PSY | Intermediate symbol file used by linker |
| SRC | Pascal source file; contains source code in ASCII form (the compiler also accepts SRC as a source filetype) |
| SYP | Symbol file used by debugger |
| SYM | Symbol file used by SID |
| TXT | Text file; contains text of messages output by compiler |
| nnn | Hexadecimal n; used for numbering overlays |

**Table 1-2.  Pascal/MT+ Distribution Disks**

| Disk 1 | |
| --- | --- |
| File | Content or Use |
| LINKMT.COM | Pascal/MT+ Linker |
| MTPLUS.COM | Pascal/MT+ Compiler |
| MTPLUS.000 | Compiler Root Program |
| MTPLUS.001 | Compiler Overlay |
| MTPLUS.002 | Compiler Overlay |
| MTPLUS.003 | Compiler Overlay |
| MTPLUS.004 | Compiler Overlay |
| MTPLUS.005 | Compiler Overlay |
| MTPLUS.006 | Overlay used with Debugger |
| PASLIB.ERL | Pascal/MT+ Run-time System Module |
| ROVLMGR.ERL | Relocatable Overlay Manager |
| MOD1.SRC | Sample Program |
| MOD2.SRC | Sample Program |
| DEMOPROG.SRC | Sample Program |

| Disk 2 | |
| --- | --- |
| File | Content or Use |
| IOCHK.BLD | LIBMT+ input command file to produce IOERR.ERL |
| DIS8080.COM | Pascal/MT+ Disassembler |
| LIBMT+.COM | LIBMT+ Librarian Utility |
| XREF.COM | Pascal cross reference utility |
| AMD9511.CMD | LINK/MT+ input command file for linking AMDIO, FPRTNS, REALIO, and TRAN9511 |
| AMD9511X.CMD | LINK/MT+ input command file for linking just AMDIO and FPRTNS |

## Table 1-2.   (continued)

| Disk 2 (continued) | |
|---|---|
| **File** | **Content or Use** |
| STRIP.CMD | LINK/MT+ input command file to produce STRIP.COM |
| XREF.DOC | Document file containing instructions for XREF, cross reference utility |
| INDEXER.DOC | Document file containing instructions for INDEXER, source file index utility |
| BCDREALS.ERL | BCD arithmetic module (does not include square root or transcendentals) |
| DEBUGGER.ERL | Debugging module that can be linked to a program |
| FPREALS.ERL | Software floating-point math module (contains REALIO.ERL) |
| FPRTNS.ERL | Hardware floating-point transcendental math module for AMD9511 |
| FULLHEAP.ERL | Heap management and garbage collection module.  PASLIB.ERL contains only USCD™-style stack/heap routines. |
| RANDOMIO.ERL | Random I/O file processing module |
| REALIO.ERL | Real arithmetic I/O module used only with AMD9511 |
| TRAN9511.ERL | Transcendental math module for use with AMD9511 |
| TRANCEND.ERL | Transcendental math module (for software floating-point only) |
| UTILMOD.ERL | Module containing KEYPRESSED, RENAME, and EXTRACT utilities |
| FIBDEF.LIB | File Information Block definition |
| APUSUB.MAC | AMD9511 routines for TRAN9511 |
| CHN.MAC | Source for @CHN; chain routine can be altered to do bank switching in a non-CP/M environment |

**Table 1-2.   (continued)**

| Disk 2 (continued) | |
|---|---|
| File | Content or Use |
| CWT.MAC | Source for @CWT routine |
| DIVMOD.MAC | Source for DIV and MOD routines that include a direct CP/M call for divide by 0 error message |
| OVLMGR.MAC | Overlay Manager source containing user-selectable options; unmodified version already in PASLIB.ERL |
| RST.MAC | Source for @RST routine |
| AMDIO.SRC | Module containing routines to interface with the AMD9511; must be customized for specific hardware |
| ATWNB.SRC | Source for @WNB routine |
| CALC.SRC | Sample program for testing floating-point math useful for testing AMD9511 |
| CPMRD.SRC | Source for routine that uses @RST |
| GET.SRC | Source for low-level input routine |
| HLT.SRC | Source for a user-defined halt routine (current routine calls CP/M) |
| INDEXER.PAS | Source program for Pascal indexing program |
| IOERR.SRC | Source for a user-defined I/O error handling routine |
| PINI.SRC | Source for @INI initialization routine |
| PUT.SRC | Source for low-level output routine |
| RNB.SRC | Source for @RNB read next byte routine |
| RNC.SRC | Source for @RNC read next character routine |
| STRIP.SRC | Source file for utility program used with LINK/MT to eliminate unused entry points in an overlay |
| TRAN9511.SRC | Source for AMD9511 routines |

Table 1-2.  (continued)

| Disk 2 (continued) | |
|---|---|
| File | Content or Use |
| UTILMOD.SRC | Source for module containing KEYPRESSED, RENAME, and EXTRACT |
| WNC.SRC | Source for @WNC routine |
| XBDOS.SRC | Source for BDOS routine that calls IOERR |
| XREF.SRC | Source for XREF, cross reference utility |
| DBUGHELP.TXT | Help file for debugger module |
| MTERRS.TXT | Compiler Error Message Text File |

## 1.2  Installing Pascal/MT+

The first thing you should do when you receive your Pascal/MT+ system is make copies of both the distribution disks.

**Note:**  you have certain responsibilities when making copies of Digital Research products.  Be sure you read your licensing agreement.

Although you can use the compiler, linker, and other utilities directly from the distribution disks, it is more convenient if you copy specific files from the distribution disks to working system disks.  One way to set up your Pascal/MT+ system is to use one disk for compiling and another disk for linking.  You can use other disks for the programming tools, assorted source code, and examples.

This suggested configuration is just one way of setting up your disks.  The important thing is that all the compiler modules are on the same disk, and all the linker modules are on one disk.  For simplicity, it is a good idea to put all the related relocatable files on the same disk as the linker.

Note that the file MTPLUS.006 is only necessary when using the debugger, and that the compiler can run without the error message file MTERRS.TXT.  If your compiler disk is short of space, you can eliminate these two files.

The following steps describe how to make a compiler disk and a linker disk:

1) Install both CP/M and the PIP utility on each of two blank disks. Label one disk as the compiler, and the other as the linker.

2) Put a text editor on the compiler disk.

3) Copy the following files from the distribution disks to the compiler disk:

   - MTPLUS.COM
   - MTPLUS.000 through MTPLUS.006
   - MTERRS.TXT

4) Copy the following files to the linker disk:

   - LINKMT.COM
   - all the ERL files

## 1.3  Compiling and Linking a Simple Program

If you have never used Pascal/MT+ before, the following step-by-step example shows you how to compile, link, and run a simple program. This example assumes that you are using a CP/M system with two disk drives, and that you are familiar with CP/M.

1) Put the compiler disk in drive A and the linker disk in drive B.

2) Using the text editor, create a file called TEST1.PAS and enter the following program. Put the file on drive B using PIP.

```
PROGRAM SIMPLE_EXAMPLE;

VAR
   I : INTEGER;

BEGIN
   WRITELN ('THIS IS JUST A TEST');
   FOR I := 1 TO 10 DO
      WRITELN (I);
   WRITELN ('ALL DONE')
END.
```

3) Now, compile the program with the following command:

   A>**MTPLUS B:TEST1**

   If you examine your directory, you see a file named
   TEST1.ERL that contains the relocatable object code
   generated by the compiler.  If the compiler detects any
   errors, correct your source program and try again.

4) Now, log on to drive B, and link the program using the
   following command:

   B>**LINKMT TEST1,PASLIB/S**

   Your directory now contains a file named TEST1.COM that is
   directly executable under CP/M.

5) To run the program, enter the command:

   B>**TEST1**

   Although the test program shown in the preceding steps is very
simple, it demonstrates the essential steps in the development
process of any program, namely editing, compiling, and linking.

   If you want to write other simple programs, follow the same
steps, but use your new program's filename instead of TEST1.

End of Section 1

# Section 2
# Compiling and Linking

This section tells how to use the compiler with its various options. It also describes how to link programs using the Pascal/MT+ linker, as well as different linkers.

## 2.1 Compiler Organization

The Pascal/MT+ compiler processes source files in three steps called passes or phases.

- Phase 0 checks the syntax and generates the token file.
- Phase 1 generates the symbol table.
- Phase 2 generates the relocatable object file.

The compiler creates some temporary files on the disk containing the source file, and under normal conditions it deletes those files. Make sure there is enough space on the disk, or use the T option to specify a different disk for the temporary files. See Section 2.2.3.

The compiler is segmented into overlays as shown in the following figure.



REQUIRED OVERLAYS                                    OPTIONAL OVERLAYS

| MT PLUS .001 | MT PLUS .002 | MT PLUS .003 | MT PLUS .004 | MT PLUS .005 | MT PLUS .006 | MTERRS.TXT |

ROOT { MT PLUS .000
        MT PLUS .COM

AN 071
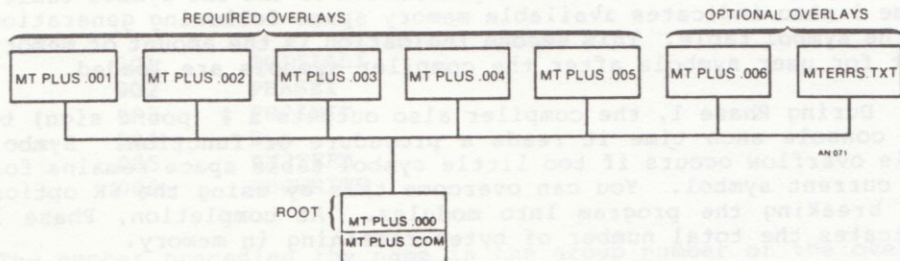
**Figure 2-1.  Pascal/MT+ Compiler Organization**

## 2.2 Invoking the Compiler

You invoke the Pascal/MT+ compiler with a command line of the following form:

        MTPLUS <filespec> {<options>}

where <filespec> is the source file to be compiled, and <options> is a list of optional parameters that you can use to control the compilation process.

2-1

The compiler can read the source file from any disk.  The
<filespec> must conform to the standard filespec format, and end
with a carriage return/line-feed, and CTRL-Z.  Refer to your
operating system manual for a description of a Digital Research
standard filespec.

If you do not specify a filetype, the compiler searches for the
file with no filetype.  If the compiler cannot find the file, it
assumes a SRC filetype, assumes a PAS filetype.  If the compiler
still cannot find the file, it displays an error message.

The compiler generates a relocatable object file with the same
filename as the input source program.  The relocatable file has the
ERL filetype.

### 2.2.1  Compilation Data

The Pascal/MT+ compiler periodically outputs information during
Phases 0 and 1 to assure you it is running properly.

During Phase 0, the compiler outputs a + (plus sign) to the
console for every 16 lines of source code it scans.

At the beginning of Phase 1, the compiler indicates the amount
of available memory space.  The space is shown as a decimal number
of memory bytes available before generation of the the symbol table.
Phase 1 also indicates available memory space following generation
of the symbol table.  This second indication is the amount of memory
left for user symbols after the compiler symbols are loaded.

During Phase 1, the compiler also outputs a # (pound sign) to
the console each time it reads a procedure or function.  Symbol
table overflow occurs if too little symbol table space remains for
the current symbol.  You can overcome this by using the $K option
and breaking the program into modules.  At completion, Phase 1
indicates the total number of bytes remaining in memory.

Phase 2 generates the relocatable object code.  During this
phase, the compiler displays the name of each procedure and function
as it is read.  The offset from the beginning of the module and the
size of the procedure (in decimal) follow the name.

When the processing is complete, the compiler displays the
following messages:

        Lines :    lines of source code compiled (in decimal)
        Errors:    number of errors detected
        Code  :    bytes of code generated (in decimal)
        Data  :    bytes of data reserved (in decimal)

## 2.2.2  Compiler Errors

When the compiler finds a syntax error, it displays the line containing the error.  If you are using the MTERRS.TXT file, the compiler also displays an error description.  If you are not using the MTERRS.TXT file, or you have a nonsyntax error, the compiler displays an error identification number.

When all processing is completed, the ERR file generated by the compiler summarizes all nonsyntactic errors.

**Note:** In Pascal/MT+, the compilation errors have the same sequence and meaning as in Jensen's and Wirth's Pascal User Manual and Report.  Appendix A contains a complete list of the error messages, explanations, and causes.

When the compiler encounters an error, it asks if you want to continue or stop, unless you use the command line option C.  (See Section 2.2.3.)

If the compiler cannot find an overlay or a procedure within an overlay, it displays messages of the following form:

        Unable to open    <filename>  <overlay # >
        Proc: "<procname>" not found ovl: <filename>   <overlay #>

The compiler displays the following procedure names if it cannot find an overlay name in the entry point table:

        001        INITIALI
        002        PHASE1
        003        PH2INIT
        004        BLK
        005        PH2TERM
        006        DBGWRITE

The number preceding the name is the group number of the overlay that contains the procedure.

Usually, you can find a missing overlay by ensuring that the name is correct, and that it is on the disk.  If you cannot find it, recopy the overlay from your distribution disk.  If you are sure the overlay is on the disk and you still get an error message, it means the file is corrupted.

## 2.2.3  Command Line Options

Compiler command line options control specific actions of the compiler such as where it writes the output files.  All command line options are single letters that start with a $ or a #.  Certain options require an additional parameter to specify where to send the output file or where an input file is located.  If you specify more than one option, do not put any blanks between the options.

Table 2-1 describes the command line options.  In this table, d stands for a parameter to specify a disk drive or output device. The parameters are as follows:

- X sends the output file to the console.
- P sends the output file to the printer.
- @ specifies the logged-in drive.
- Any letter from A to O specifies a specific drive.

**Table 2-1.  Default Values for Compiler Command Line Options**

| Option | Meaning | Default |
|--------|---------|---------|
| A | Automatically calls the linker at the end of compilation. This option requires a linker input command file with the same name as the input file. The linker must be named LINKMT.COM. | Compiler automatically chains. |
| B | Uses BCD rather than binary for real numbers. | Binary reals. |
| C | Continues on error; default is to pause and let user interact and asks on n each error, one at a time. | Compiler stops and asks on each error. |
| D | Generates debugger information in the object code and writes the PSY file to the drive specified by the R option. | No debugger information and no PSY file generated. |
| Ed | The MTERRS.TXT file is on disk d: where d=@,A..O. | MTERRS.TXT on default disk. |
| Od | MTPLUS.000, and MTPLUS.001 through MTPLUS.006 are on drive d: where d=@,A..O. | Overlays on default disk. |
| Pd | Puts the PRN (listing file) on disk d: where d=X,P,@, A..O. | No PRN file. |
| Q | Quiet; suppresses any unnecessary console messages. | Compiler outputs all messages. |
| Rd | Puts the ERL file on disk d: where d=@,A..O. | ERL file on default disk. |

Table 2-1.  (continued)

| Option | Meaning | Default |
|--------|---------|---------|
| Td | Puts the token file PASTEMP.TOK on disk d: where d=@,A..O. | PASTEMP.TOK on default disk. |
| V | Prints the name of each procedure and function when found in source code as an aid to determining error locations during Phase 0. | Procedure names not printed. |
| X | Generates an extended ERL file, including disassembler records. | ERL file cannot be disassembled. |
| @ | Makes the @ character equivalent to the ^ character. | @ not equivalent to ^. |

The following is an example of a Pascal/MT+ command line:

          A>MTPLUS A:TESTPROG $RBPXA

This command line tells the compiler to read the source from drive A, write the ERL file to drive B, display the PRN file on the console, and call the linker automatically.

## 2.2.4  Source Code Options

Source code compiler options are special instructions to the compiler that you put in the program source code.  A source code option is a single lower- or upper-case letter preceded by a dollar sign, embedded in a comment.  The option must be the first item in the comment.   Certain source code options require additional parameters.

You can put any number of options in a source program, but only one option per comment is allowed.  You cannot place blanks between the dollar sign and the option letter.  The compiler accepts blanks between the option letter and the parameter.

Pascal/MT+ supports twelve source code compiler options, as summarized in Table 2-2.

**Table 2-2.  Compiler Source Code Options**

| Option | Function | Default |
|--------|----------|---------|
| Cn | Use RST n instructions for REAL operation. | Use CALL instructions |
| E +/- | Controls entry point generation. | E+ |
| I<filespec> | Includes another source file into the input stream, for example, {$I XXX.LIB}. | |
| Kn | Removes built-in routines to save space in symbol table (n=0..15). | |
| L +/- | Controls the listing of source code. | L+ |
| P | Enter a form-feed in the PRN file. | |
| Qn | Use RST n instructions for loads and stores in recursive environments. | Use CALL instructions |
| R +/- | Controls range checking code. | R- |
| S +/- | Controls recursive/static variables. | S+ |
| T +/- | Controls strict type checking. | T- |
| W +/- | Generates warning messages. | W- |
| X +/- | Controls exception checking code. | X- |
| Z $nnnnH | Initialize hardware stack to nnnnH. | Contents of location 0006 at beginning of execution |

The following examples show proper source code compiler options:

```
{$E+}
(*$P*)
{$I D:USERFILE.LIB}
```

## Space Reduction: Real Arithmetic (Cn)

The Cn option reduces the amount of object code generated when using REAL arithemtic.  The Cn option tells the compiler to change all calls to @XOP (the REAL load and store routine) into a restart instruction.  This reduces all 3-byte CALL instructions to 1-byte CALL instructions.

You specify a restart instruction number in the range 0 to 7 and the compiler generates RST n instructions.  Be aware that in a CP/M environment, restart numbers 0 and 7 are not available.  If you have another operating system, you should consult your hardware documentation.

You must specify the Cn option in the main program so the compiler can generate code to load the restart vector and RST n instructions for any call to @XOP.  You must also specify the Cn option in any modules that use real numbers so the proper RST n instructions are generated.

## Entry Point Record Generation (E)

The E option generates entry point records in the relocatable file.  You enable the option using a + parameter, and disable it using a - parameter.  E+ is the default.

E+ makes global variables and all procedures and functions available as entry points.  For example, EXTERNAL declarations in separate modules can reference global variables and all procedures and functions if the E+ option is in effect.  E- suppresses the generation of entry point records, thus making all variables, procedures, and functions local.

## Include Files (I)

I<filespec> tells the compiler to include a specified file for compilation in the input stream of the original program.  The compiler supports only one level of file inclusion, so you cannot nest include files.

The filespec must contain the drive specification, filename, and filetype in standard format.  If you omit the filetype, the compiler looks for a file with the type of the main file.  The file must end with a carriage return/line-feed, and CTRL-Z.  If you omit the drive specification, the compiler looks on the default drive.

## Symbol Table Space Reduction (Kn)

Predefined identifiers normally take about 6K bytes of symbol table space.  The K option removes unreferenced built-in routine definitions from the symbol table to make more room for user symbols.

The K option uses an integer parameter ranging from 0 to 15. Each integer corresponds to different groups of routines as defined in Table 2-3.  Enter all K options before the words PROGRAM or MODULE in the source code.  Use as many K options as required, but place only one integer parameter after each letter K. Note that any reference in a program to the removed symbols generates an undefined identifier error message.

**Table 2-3.  $K Option Values**

| Group | Routines Removed |
|:-----:|------------------|
| 0 | ROUND, TRUNC, EXP, LN, ARCTAN, SQRT, COS, SIN |
| 1 | COPY, INSERT, POS, DELETE, LENGTH, CONCAT |
| 2 | GNB, WNB, CLOSEDEL, OPENX, BLOCKREAD, BLOCKWRITE |
| 3 | CLOSE, OPEN, PURGE, CHAIN, CREATE |
| 4 | WRD, HI, LO, SWAP, ADDR, SIZEOF, INLINE, EXIT, PACK, UNPACK |
| 5 | IORESULT, PAGE, NEW, DISPOSE |
| 6 | SUCC, PRED, EOF, EOLN |
| 7 | TSTBIT, CLRBIT, SETBIT, SHR, SHL |
| 8 | RESET, REWRITE, GET, PUT, ASSIGN, MOVELEFT, MOVERIGHT, FILLCHAR |
| 9 | READ, READLN |
| 10 | WRITE, WRITELN |
| 11 | unused |
| 12 | MEMAVAIL, MAXAVAIL |
| 13 | SEEKREAD, SEEKWRITE |
| 14 | RIM85, SIM85, WAIT |
| 15 | READHEX, WRITEHEX |

Listing Controls (L,P)

The L option controls the listing that the compiler generates during Phase 0.  You enable the L option with the + parameter and disable it with the - parameter.

The P option starts a new page by placing a form-feed character in the PRN file.

Space Reduction: Recursion (Qn)

The Qn option operates in a manner analagous to the Cn option. That is, you specify a restart instruction number in the range 0 to 7, and the compiler generates RST n instructions for every call to @DYN.

You must specify the Qn option in the main program so the compiler can generate code to load the restart vector and RST n instructions for any call to @DYN.  You must also specify the Cn option in any modules that use recursion so the proper RST n instructions are generated.

Run-time Range Checking (R)

The R option controls the generation of run-time code that performs range checking for array subscripts and storage into subrange variables.  You enable the R option with the + parameter and disable it with the - parameter.  Refer to Section 4.6.1 for information on range checking.

Recursion and Stack Frame Allocation (S)

The S option controls the stack frame allocation of procedure and function parameters and local variables.  The + parameter causes recursion.  The default parameter is -, and causes nonrecursion. Pascal/MT+ statically allocates global variables in programs and modules.  You must enable the S option before the reserved words PROGRAM and MODULE.  You cannot disable the S option within a separately compiled unit.  You can link modules that use the S+ option with those that do not.

Strict Type and Portability Checking (T,W)

The T option controls the strict type checking/nonportable warning facility.  The W option controls the display of warning messages pertaining to the T option.  You enable both options with the + parameter and disable them with the - parameter.  The default value for both options is -.

When the T option is enabled, the compiler performs only weak type checking.  If the T and W options are enabled, and the compiler detects a nonportable feature, the compiler displays error message 500.  String operations cause error 500 when the two options are enabled, because the STRING data type is not standard.

The T and W options check for compatibility with the ISO Pascal standard.   They  do  not  check  for  all  features  listed  in  the Pascal/MT+ Language Reference Manual, because certain features are implementation-dependent and others are software routines.

## Run-time Exception Checking (X)

In the current release of Pascal/MT+, the X option remains in effect.   Normally,  the  X  option  controls  exception  checking. Exception checking covers integer and real zero division, string overflow, real number overflow, and underflow. Refer to Section 4.6 for information on run-time error handling.

## Setting the Stack Pointer (Z)

The Z option initializes the stack pointer to nnnnH in non-CP/M environments.  In a CP/M environment, the compiler initializes the hardware stack by loading the stack pointer register with the contents of absolute location 0006H. Using the Z option suppresses this initialization.

You should enter the option as $Z+ only once before the PROGRAM line in the main program, and not on the individual modules.

## 2.3  Using the Linker

LINK/MT+ is the linkage editor that reads relocatable object modules with filetype ERL and generates an executable command file with filetype COM.  The linker can also generate overlay files.

You  invoke  LINK/MT+  with  a  command  line  of  the  following format:

        LINKMT <main module>{,<module>}{,<library>}

or

        LINKMT <new filespec>=<main module>{,<module>}{,<library>}

The linker writes the executable file to the same logical disk as the <main module>, unless you specify a new <filespec> using an equal sign.   The <main module> and each <module> can be on any logical drive.  You can specify the drive before each file in the command line.

The linker assumes a ERL filetype for the <main module> and all <modules> unless you specify a CMD filetype. See the discussion about the /F option for information about CMD files. LINK/MT+ can link a maximum of 32 files at one time.

The following examples show valid LINK/MT+ command lines:

    A>LINKMT CALC,TRANCEND,FPREALS,PASLIB/S

    A>LINKMT B:CALC=CALC,B:TRANCEND,FPREALS,PASLIB/S

    A>LINKMT D:NEWPROG=B:CALC,C:TRANCEND,C:FPREALS,C:PASLIB/S/M

## 2.3.1 Linker Options

Linker options are special instructions to LINK/MT+ that you specify in the command line. You specify options as a single lower- or upper-case letter. Each option must be preceded in the command line with a slash, /. Some options require an additional parameter. LINK/MT+ supports 13 options, as summarized in Table 2-4.

**Table 2-4.  Linker Options**

| Option | Function |
| --- | --- |
| C | Line continuation flag. Used only in CMD linker command files. |
| D:nnnnH | Relocate data area to nnnnH. |
| E | List entry points beginning with $, ?, or @ in addition to other entry points requiring /M or /W to operate. |
| F | Take preceding filename as a CMD linker command file containing input filenames, one per line. |
| Hnnnn | Write the output as a HEX file with nnnnH as the starting location of the hex format. This option is independent of the P option. Also, if you use this option, the compiler does not generate a COM file. |
| L | List modules as they are being linked. |
| M | List all entry points in tabular form. |
| P:nnnn | Relocate object code to nnnnH. |

Table 2-4.  (continued)

| Option | Function |
|---|---|
| S | Search preceding name as a library, extracting only the required routines. |
| W | Write a SID-compatible SYM file (written to the same disk as the COM file). |
| O:n | Number the overlay as n and use the previous filename as the root program symbol table.  By default, the range of n is 1 to 50, but you can extend it to 1 to 256 by altering the overlay manager. |
| Vn:mmmm | Overlay area starting address. |
| X:nnnn | Overlay static variable starting address when used with overlays, or amount of overlay data area when used with root modules. |

Continue Line  (/C)

     The C option indicates a continued line in a linker input command (CMD) file.  See the discussion of the F option below.

Data Location  (/D)

     The D:nnnn option tells the linker to start the data area at the hexadecimal address nnnn.  If you do not use the D option, the code and data are mixed in the object file.  By using the D option, you can solve some memory limitation problems.

     However, you should be aware that local file operations depend on the linker to zero the data area.  The linker does not zero the data area when you use the D switch, so these operations cannot be guaranteed.

Linker Input Command File  (/F)

     Normally in a CP/M environment, you must use the SUBMIT facility for typing repetitive sequences, such as linking multiple files together.  LINK/MT+ allows you to enter this data into a file and have the linker process the filenames from the file.  You must specify a file with a filetype of CMD and follow this filename with a /F, for example, CFILES/F.

The linker reads input from this file and processes the
filenames.  Filenames can be on one line, separated by commas, or
each name or group of names can be on a separate line.  At the end
of each line except the last, you must place a /C option.  The last
line must end with a carriage return or line-feed.

The input from the file is concatenated logically after the
data on the left of the filename.  In the command line, additional
options can follow the /F, but not additional object module names.

The following example demonstrates how to use a CMD file to
link the files CALC, TRANCEND, FPREALS, and PASLIB into a CMD file.
Use the following command to link the files:

     A>**LINKMT CALC/F/L**

The file CALC.CMD contains

     A:CALC,D:TRANCEND,FPREALS,B:PASLIB/S

The linker searches PASLIB for the necessary modules and generates a
link map.

## Hex Output (/H)

The H:nnnn option tells the linker to generate a HEX file
instead of a COM file, starting the program at the hexadecimal
address nnnn.  The specified address is independent of the default
relocation value of 100H.  This means you can relocate the program
to execute at 1D00H, for example, but have the HEX file addresses
start at 8000H, by using the parameters:

     /P:1D00/H:8000

## Load Maps (/L),(/E)

The L option tells the linker to display module code and data
locations as they are linked.

When used with the M or W options, the E option tells the
linker to display all routines as they are linked, including
routines that begin with ? or @, which are reserved for run-time
library routine names.  The E option does not enable the L, M, or W
option.  E does not display module code and data locations if used
alone.

## Memory Map (/M)

The M option generates a map and sends it to the map output
file.  Place the M option after the last file named in the parameter
list.

Program Relocation  (/P)

The P:nnnn option tells the linker to start the program at the hexadecimal address nnnn.  If you do not use the P option, the default address is 100H.

The linker does not generate space-filling code at the beginning of the program.  The first byte of the COM file is the byte of code that belongs in the specified starting location.

The syntax of the P option is

        /P:nnnn

where nnnn is a hexadecimal number in the range 0 to FFFF.

Run-time Library Search  (/S)

The S option tells the linker to search the file whose name the option follows as a library and to extract only the necessary modules.  The S option must follow the name of the run-time library in the linker command line.  The S option extracts modules from libraries only.  It does not extract procedures and functions from separately compiled modules.

The order of modules within a library is important.  Each searchable library must contain routines in the correct order and be followed by /S.   PASLIB and FPREALS are specially constructed for searchability.  Unless otherwise indicated, the other ERL files supplied with the Pascal/MT+ system are not searchable.  You cannot search user-created modules unless they are processed by LIBMT+, as described in Section 5.3.

Generate SYM File  (/W)

The W option tells the linker to generate a SID-compatible SYM file.  The file contains information about entry points in the program.  The linker uses the SYM file when it links overlays.  The V option also enables the W option.

Overlay Options

The linker uses three options to process an overlay or a root program in an overlay scheme.  The O option numbers the overlay and indicates that the previous filename is the root program symbol table. The Vm option sets the address of the overlay area.  The X option controls how the linker allocates data space for overlays. Section 3.2 explains these overlay options.

## 2.3.2  Required Relocatable Files

You must always link the run-time system PASLIB.ERL with your compiled program.  In addition, you need to link other ERL files with your program if it makes use of certain features of Pascal/MT+. The following are such files:

- RANDOMIO:  SEEKREAD and SEEKWRITE are resolved here.

- DEBUGGER:  @NLN, @EXT, @ENT generated when the debugger option is requested.  If @XOP and @WRL are undefined, see Section 5.2.

  The following files contain the real-number routines:

- BCDREALS:  BCD real numbers, @XOP, @RRL, and @WRL.

- FPREALS:  Binary real numbers @XOP, @RRL, and @WRL.

- TRANCEND:  Support for SIN, COS, ARCTAN, SQRT, LN, EXP, SQR. Use only with FPREALS.

  The following files contain real number routines used with the AMD9511:

- AMDIO:  Routines for interfacing with the AMD9511.  You must edit and recompile these to customize for specific hardware requirements.

- FPTRNS:  AMD9511 support routines.

- REALIO:  Read and Write real number routines necessary only when using the AMD9511.

- TRAN9511:  Transcendental routines for AMD9511 (replaces TRANCEND).

## 2.3.3  Linker Error Messages

Table 2-5 shows the linker error messages.

**Table 2-5.  Linker Error Messages**

| Message | Meaning |
|---|---|
| Unable to open input file: xxxxxxxx | The linker cannot find the specified input file. |
| Incompatible relocatable file format | The ERL file is corrupted, or it has a format that is incompatible with the format expected by LINK/MT+. |
| Duplicate symbol:  xxxxxxx | This usually means a run-time routine or variable has the same name as a user routine or variable. |
| SYSMEM not found in SYM file | This means the root program symbol file is corrupt. |
| External offset table overflow | This means you have exceeded the 200 externals plus offset addresses that the linker allows in its offset table. |
| Initialization of DSEG not allowed | The linker has encountered a DB or DW instruction in the Data segment. |

## 2.4 Using Other Linkers

When you compile your program using the X option, Pascal/MT+ generates an extended relocatable file containing disassembler records. If you do not use the X option, the ERL file might be Microsoft® compatible. However, Digital Research does not guarantee that an ERL file generated by Pascal/MT+ is compatible with other linkers such as L80.

However, using LIBMT+ to process the ERL files generated by the compiler can result in a Microsoft-compatible relocatable files (see Section 5.3).

<div align="center">End of Section 2</div>

## 2.4 Using Other Linkers

### 2.4.1 Linker Error Messages

When you compile your program using the X option, Pascal/MT+ generates an extended relocatable file containing disassembler records. If you do not use the X option, the REL file might be Microsoft* compatible. However, Digital Research does not guarantee that any REL file generated by Pascal/MT+ is compatible with other linkers such as LINK.

However, using LINK/MT+ to process the REL files generated by the compiler can result in a Microsoft-compatible relocatable files (see section 5.1).

| Message | Meaning |
|---|---|
| | The linker cannot find the specified input file. |
| Incompatible relocatable file format | The REL file is corrupted, or it has a format that is incompatible with the format expected by LINK/MT+. |
| Duplicate symbol: xxxxxx | This usually means a run-time routine or variable has the same name as a user routine or variable. |
| xxxxxx not found in SYM file | This means the root program symbol file is corrupt. |
| External offset table overflow | This means you have exceeded the 300 external plus offset addresses that the linker allows in its offset table. |
| Initialization of ROM not allowed | The linker has encountered a DB or DW instruction in the DATA segment. |

# Section 3
# Segmented Programs

One of the biggest advantages of Pascal/MT+ is the ability to write a large, complex program as a series of small, independent modules. You can code, test, debug, and maintain each module separately, and thereby greatly simplify the overall task of program design. The process of breaking a program into separate units is called segmenting.

Pascal/MT+ provides three methods for segmenting programs: modules, overlays, and chaining.

- Modules are separately compiled program sections. You can link modules together to build entire programs, libraries, or overlays.

- Overlays are sections of programs that only need to be in memory when a routine in that overlay is called. Otherwise, the overlay remains on the disk.

- Chaining allows one program to call another, leaving shared data for the new program in memory.

You can use these three features in any combination to produce modular programs that are easier to maintain and take up less memory than monolithic programs.

If you are not an experienced Pascal/MT+ programmer, you should start by writing programs without overlays.


## 3.1 Modules

The Pascal/MT+ system lets you do modular programming with little preplanning. You can develop programs until they become too large to compile and then split them into modules. The $E compiler option lets you make variables and procedures private.

Modules are similar in form to programs. The differences are the following:

- Use the word MODULE instead of the word PROGRAM.

- There is no main statement body in a module. Instead, after the definitions and declaration section, use the word MODEND, followed by a period.

The following is an example of a module:

```
      MODULE     LITTLEMOD;

      VAR

         MAINFILE  :  EXTERNAL TEXT;
      PROCEDURE ECHO (ST: STRING; TIMES: INTEGER);
      VAR
         I : INTEGER
      BEGIN
         FOR I := 1 TO TIMES DO
            WRITELN (MAINFILE, ST)
      END;

      MODEND.
```

Note that a module must contain at least one procedure or function.

Modules can have free access to procedures and variables in any other module.  If you want to keep procedures or variables private within a module, use the $E- compiler option.

Use the EXTERNAL directive to declare variables, procedures, and functions that are allocated in other modules or in the main program.  EXTERNAL tells the compiler not to allocate space in the module.  You can declare externals only at the global (outermost) level of a module or program.

For variables, put the word EXTERNAL between the colon and the type in a global declaration.  For example,

```
      VAR
         I,J,K : EXTERNAL INTEGER; (* in another module *)

         R:      EXTERNAL RECORD  (* in another module *)
                 x,y : integer;
                 st : string;
                 END;
```

Be sure the declarations match with the declarations in the module where the space is allocated.  The compiler and linker do not check declarations between modules.

For procedures and functions declared in other modules, put the word EXTERNAL before the word FUNCTION or PROCEDURE.  These external declarations must come before the first normal procedure or function declaration in the module or program.

Numbers and types of parameters must match in the Pascal/MT+ system. Returned types must match for functions; the compiler and linker do not type check across modules. External routines cannot have procedures and functions as parameters.

In Pascal/MT+, external names are significant to seven characters only. Internal names are significant to eight characters.

In Pascal/MT+, the code generated for main programs and for modules differs in the following ways:

● Main programs begin with sixteen bytes of header code. Modules do not.

● Main programs have a main body of code following the procedures and functions. Modules do not.

Listing 3-1 shows the outline of a main program and Listing 3-2 shows the outline of a module. The main program references variables and subprograms in the module; the module references variables and subprograms in the main program.

```
PROGRAM EXTERNAL_DEMO;

<label, constant, type declarations>

VAR

    I,J : INTEGER;              (* AVAILABLE IN OTHER MODULES *)

    K,L : EXTERNAL INTEGER; (* LOCATED ELSEWHERE *)

EXTERNAL PROCEDURE SORT(VAR Q:LIST; LEN:INTEGER);

EXTERNAL FUNCTION  IOTEST:INTEGER;

PROCEDURE PROC1;
BEGIN
   IF IOTEST = 1 THEN
       (* CALL AN EXTERNAL FUNC NORMALLY *)
       ...
END;

BEGIN
   SORT(....)
   (* CALL AN EXTERNAL PROC NORMALLY *)
END.
```

**Listing 3-1.  Main Program Example**

```
        MODULE MODULE_DEMO;

        < label, const, type declarations>

        VAR

            I,J : EXTERNAL INTEGER; (* USE THOSE FROM MAIN PROGRAM *)

            K,L : INTEGER;          (* DEFINE THESE HERE *)

        EXTERNAL PROCEDURE PROC1; (* USE THE ONE FROM MAIN PROG *)

        PROCEDURE SORT(...);      (* DEFINE SORT HERE *)
            ...

        FUNCTION IOTEST:INTEGER;  (* DEFINE IOTEST HERE *)
            ...

        <maybe other procedures and functions here>
        MODEND.
```

**Listing 3-2.  Module Example**

## 3.2  Overlays

Using overlays, you can link programs so that parts of them
automatically load from the disk as they are needed.  Thus, a whole
program does not have to fit in memory simultaneously.  Store
infrequently used modules and module groups that need not be co-
resident in overlays.

The following terms are used in this section:

● **overlay:**  a set of modules, linked together as a unit, that
   loads into memory from disk when a procedure or function in one
   of the modules is referenced from somewhere else in the
   program.  Overlays have hexadecimal filetypes, for example,
   PROG.01F.

● **root program:**  the portion of the program that is always in
   memory.  Root programs have the COM filetype.  A root program
   consists of a main program, the run-time routines it requires,
   and optionally, the run-time routines the overlays require.

● **overlay area:**  an area of memory where the overlay manager
   loads overlays.  You must plan the location and size of the
   overlay areas and specify them at link-time.

● **overlay static variables:** global variables, or variables local to a run-time or assembly language routine in the overlay. When you link the overlay, the linker determines the amount of data space required for static variables.  Recursion reduces the amount of static data.  It does not necessarily eliminate it because run-time code linked with the overlay might contain static data.

### 3.2.1  Pascal/MT+ Overlay System

The major features of the Pascal/MT+ overlay system are the following:

● Supports up to 255 overlays.
● Supports up to 15 separate overlay areas.
● Overlays can call other overlays, even in the same overlay area.
● Overlays can access procedures and variables in the root.
● Overlays load from the disk only when necessary.
● Overlays can contain an arbitrary number of modules.
● Linkage to a procedure in an overlay is by name.
● You can specify drives containing individual overlays.

Overlays have an arbitrary number of entry points for the root program and other overlays to access.  They access the entry points by name.  The linker and relocatable formats limit overlay procedure and function names to 7 significant characters, as with all externals.

You assign overlay areas when you link the root module.  You assign overlay numbers when you link the overlay.  If you do not specify an overlay area when you link the root module, the default action is to place it in overlay area 1.

Most Pascal/MT+ programs use only one overlay area.  You can devise more extensive schemes using multiple overlay areas.  The overlay number determines the area where LINK/MT+ loads an overlay.

● Overlays 1 to 16 load into overlay area 1.
● Overlays 17 to 32 load into overlay area 2.
.
.
.
● Overlays 241 to 255 load into overlay area 15.

You must determine the size and address of overlay areas and make sure the overlays are smaller than the area into which they load.  If you do not specify the address for an overlay area, it defaults to the same address as overlay area 1.

3-5

The overlay manager loads overlays into memory in 128-byte segments, so consider the extra size when you save space for overlays.  You must specify area 1; the remaining areas are optional.

Overlays have one or more modules, written in Pascal or assembly language.  The overlay manager in PASLIB has space in its drive table for 50 overlays, numbered 1 to 50.  If you need more overlays, you can modify the overlay manager source, reassemble it, and link it before PASLIB.  The source code for the overlay manager is in the file OVLMGR.MAC on distribution disk #2.

You do not have to number overlays consecutively.  For example, if you want to use three overlays in three overlay areas, you can number them 1, 17, 33, or any combination that puts the overlays in different areas.

You can load more than 15 overlays into overlay area 1 by explicitly supplying the overlay area number when you link the root module.  Otherwise, the default number is 15.

### 3.2.2  Using Overlays

If a procedure or function is in an overlay, the compiler inserts a call to the overlay manager, @OVL, before the call to the procedure or function.  @OVL makes sure that the requested overlay is in memory, loading it from disk if necessary.  When the procedure or function returns, the overlay manager returns control to the calling procedure.

When part of a program calls an overlay-resident routine, the program accesses that routine through an entry point table at the beginning of the overlay.  Only procedures and functions declared without the $E- compiler option have their names in the entry point table.  Use the $E- option to make routines private to an overlay and to save space in the table.

Calling an Overlay Procedure

To tell the compiler that a procedure or function is in an overlay, put the overlay number in the declaration, as in the following examples:

        EXTERNAL [ 3 ] PROCEDURE CONV_SYM;
        EXTERNAL [ FIXUP ] FUNCTION NEW_TOK : INTEGER;

The overlay number must be an integer constant, either literal or named.

Overlays can access procedures, functions, variables, and run-time routines in the root by using regular external declarations.

If an overlay is not on the same disk as the root file, use the @OVS routine to specify the drive.  Declare the routine as shown in the following example:

```
EXTERNAL PROCEDURE @OVS
          ( OVERLAY_NUMBER : INTEGER; DRIVE : CHAR );
```

Call @OVS to define the drive before calling the overlay-resident procedure or function.  The drive must be upper-case, and can be the @ character or a letter from A through O.  The @ represents the logged-in disk.  You must ensure that the specified disk is on-line.

## Overlays Calling Other Overlays

The standard overlay manager does not reload a previous overlay when it returns from an overlay call.  If you want to return control to a previous overlay in the same overlay, you must use the reloading version of the overlay manager, which is in the file ROVLMGR.ERL on distribution disk #1.  If you need the reloading version, link it before PASLIB.

Overlays can call other overlays under the following conditions:

- You use /X to link overlays if there are static variables in the overlays.  This ensures that no procedure alters the data of another.

- You must use the reloading overlay manager if an overlay calls another overlay in the same overlay area.  If the overlays are in different overlay areas, both must be in memory at the same time.

## Assembly Language Modules

Pascal/MT+ overlays are always pure code, but other modules written in assembly language might not be.  The overlay does not reload if it is already in the overlay area.  Do not use DB in the Code segment for variables that are modified, because they are not initialized every time the overlay is called.

## 3.2.3  Linking Programs with Overlays

The linker separately links each part of a program containing overlays.  The linker first builds a SYM file containing the entry points for the root, and then uses that file when it links the overlays.

Before the entry points can be correct, you have to know how much code and data space the overlays need.  The first time that you link an overlay program, you have to link the entire program twice: once to determine the sizes, and once to produce the actual program files.  The following steps outline the linking process.

1) Link the root program without reserving space for the overlay areas and overlay data.  This step generates the first SYM file.

2) Use the SYM file from step 1 to link the overlays.  This step tells you how much space the overlays need.

3) Relink the root, specifying the overlay area addresses and static data size.  This step produces the SYM file with the correct entry points.

4) Relink the overlays, using the new SYM file.

There are three linker options that control overlay linking:

● The O option specifies overlay numbers.
● The V option specifies overlay area addresses.
● The X option specifies data area sizes.

#### Overlay Group and SYM Option /O:

/O:n tells the linker that the previous file is a SYM file and that n is the overlay number, in hexadecimal.  The linker uses the overlay number to make the filename.  This option is for overlays only.

If you make a change in an overlay, you need only to relink the overlay.  The exception is when the code size or data size changes beyond the constraints you gave when you linked the root.

#### Overlay Area Option /V:

/Vn:mmmm tells the linker where to locate the overlay area. mmmm is the hexadecimal address of the overlay area, and n is the overlay area number, in hexadecimal.

The V option automatically enables the E and W options, causing the linker to generate a SYM file.  This option is for root programs only.

You can use the /V option up to 16 times when you link the main program, once for each of the 16 overlay areas.  You must use it at least once to give the default address for overlay area 1.

To find the value for /V, link the root program with the necessary libraries. The root program's total code size plus 80H is the lowest address you can use for an overlay area.

## Overlay Local Storage Option /X:

X:nnnn controls how the linker allocates space for data. This option is for both roots and overlays. To determine the amount of data used by an overlay, link it and note the total data size put out by the linker.

**Note:** when you use this option, give yourself extra space so that you do not have to relink everything when the data areas change size.

When used to link roots, /X:nnnn tells the linker how much space to leave for overlay data. nnnn is the hexadecimal number of bytes.

When linking overlays, /X:nnnn tells the linker how far to offset a particular overlay's static data area.  nnnn is the hexadecimal number of bytes from the top of the root's data area. The default value for this option is /X:0000.

For example, suppose a program has two overlays with a combined total of 500 bytes of static data.  Overlay 1 has 350 bytes, and overlay 2 has 150 bytes.  Overlay 1 needs no offset, and overlay 2 needs to have its data area 350 bytes from the end of the root's data area. The minimum value for overlay 2 is /X:015E, which is 350 in hexadecimal.

## Linking a Root Program

Linking a root program is similar to linking a nonoverlaid program.  The difference is that you have to generate the SYM file, and you have to allow room for the overlay areas and for overlay static data.  The command line for linking a root program has the general form:

        LINKMT <modules and libraries> /Vn:mmmm/D:oooo/X:pppp

This command line shows the two required options Vn and D.  You can use any of the other options as needed.

- Use the V option for each separate overlay area.  You must at least specify the location of overlay area 1.  If you do not specify a location for any other overlay areas, the linker assigns them the same location as area 1.

- The D option specifies the location of the data area.  The value is the sum of the root's code size and the sizes of the overlays' code.  Leave room during development so that the overlay data areas can grow.

● Remember to use the X option if your program uses overlay static variables.

The overlay manager reads in 128 bytes of code at a time. Make sure you allow room at the end of your overlay areas so that the garbage bytes that pad out the last sector do not overwrite the next area. The minimum size for an overlay area should be the size of the largest overlay plus 80H, rounded to the next multiple of 128.

During development, you should leave some extra room in the overlay areas so that you do not have to relink the entire program if one overlay gets bigger.

If an overlay calls a library routine that the root does not call, the linker puts the routine in the overlay. To force a routine into the root, make a dummy reference to the routine in the root.

When you link a root program just to generate a SYM file, either use a dummy value for V or use the E and W options. Either way generates the symbol file.

Linking an Overlay

When linking an overlay, the linker uses the SYM file to tell which symbols are in the root. If an external symbol is not in the SYM file, the linker looks for it in the specified libraries. The command line for linking overlays takes the following form:

    LINKMT <prog>=<sym file>/O:n,<modules/libraries>/P:mmmm/X:ssss

The linker generates a file with the same name as the program, but with a filetype that is the overlay number in hexadecimal. If you do not specify the program name, the linker uses the name of the first module after the SYM file.

The command line above shows the options that are required for linking overlays. Note that the /X option is required only if the overlay uses static data.

● The O option tells the linker that the file is a SYM file and that the overlay number is n, in hexadecimal.

● For P, use the starting address of the overlay area. Use the same value that you use with the V option that sets up the overlay area.

● Use the X option to specify the offset from the end of the root modules's data to the beginning of the overlays's static data.

You must relink an overlay whenever you relink the root, because entry points change. Be sure to use the new SYM file.

### 3.2.4 Overlay Error Messages

The overlay manager can detect two errors:

● If the overlay manager cannot find the requested overlay, it displays a message of the form:

Unable to open    <filename>    <overlay #>

If the overlay is not on the default disk, call @OVS in the program to tell the overlay manager where to look.

● If the overlay manager cannot find a particular procedure or function in the specified overlay, it displays a message of the form:

Proc: "<procname>" not found ovl: <filename>    <overlay #>

The problem might be an incorrect EXTERNAL statement or a misnumbered overlay.

### 3.2.5 Example

The following example has a root program that asks for a character from the console keyboard. It calls one of two procedures, depending on the character entered. A large menu-driven business package could work in a similar way.

The main program and the two modules are shown in Listings 3-3, 3-4, and 3-5, respectively. These files are also on distribution disk #1. You should compile and link them to get a feel for using overlays. The files are the following:

● PROG.SRC
● MOD1.SRC
● MOD2.SRC

```
PROGRAM DEMO_PROG;

VAR
  I : INTEGER;  (* TO BE ACCESSED BY THE OVERLAYS *)
  CH: CHAR;

EXTERNAL [1] PROCEDURE OVL1; (* COULD HAVE HAD PARAMETERS *)

EXTERNAL [2] PROCEDURE OVL2; (* ALSO COULD HAVE HAD PARAMETERS *)

(* EITHER COULD ALSO HAVE BEEN A FUNCTION IF DESIRED *)

BEGIN
  REPEAT
    WRITE('Enter character, A/B/Q: ');
    READ(CH);
    CASE CH OF
      'A','a' : BEGIN
                  I := 1; (* TO DEMONSTRATE ACCESS OF GLOBALS *)
                  OVL1    (* FROM AN OVERLAY *)
                END;

      'B','b' : BEGIN
                  I := 2;
                  OVL2
                END
    ELSE
      IF NOT(CH IN ['Q','q']) THEN
        WRITELN('Enter only A or B')
    END (* CASE *)
  UNTIL CH IN ['Q','q'];
  WRITELN('End of program')
END.
```

**Listing 3-3.   PROG.SRC**

```
MODULE OVERLAY1;

VAR
  I : EXTERNAL INTEGER; (* LOCATED IN THE ROOT *)

PROCEDURE OVL1; (* ONE OF POSSIBLY MANY PROCEDURES IN THIS MODULE *)
  BEGIN
    WRITELN ('In overlay1, I=',I) END;

MODEND
```

**Listing 3-4.   MOD1.SRC**

```
MODULE OVERLAY2;

VAR
  I : EXTERNAL INTEGER; (* LOCATED IN THE ROOT *)

PROCEDURE OVL2; (*ONE OF POSSIBLY MANY PROCEDURES IN THIS MODULE *)
BEGIN
  WRITELN ('In overlay 2, I=',I) END;

MODEND.
```

**Listing 3-5.  MOD2.SRC**

After you compile the three modules, you must link them together.  Link the main program using the command:

**A>LINKMT PROG,PASLIB/S/D:1000/V1:4000/X:40**

This creates the files PROG.COM and PROG.SYM with the data located at 1000 (this is arbitrary).  The overlay areas, 1 to 16, are at 4000 (again arbitrary), and the overlay data size is estimated to be 64 (40H).

To link overlay 1, enter this command:

**A>LINKMT PROG=PROG/O:1,MOD1,PASLIB/S/P:4000/L**

This creates the overlay file PROG.001.  The /O:1 option tells the linker to read PROG.SYM, and this is overlay #1.  4000 is the address of the overlay area for this overlay.  The linker searches PASLIB to load only those modules required by this overlay, but not present in PROG.COM.

To link overlay 2, enter this command:

**A>LINKMT PROG=PROG/O:2,MOD2,PASLIB/S/P:4000/L**

The options are the same as above.  Note that /X is not needed when linking the overlays, because the overlays do not have any local data.

Now run the program.  Notice that if you enter the same letter more than once in succession, for example, A, A, A, the overlay does not reload.  However, when you enter the letters in alternate order, for example, A, B, A, ..., the overlays load for each call.

## 3.3  Chaining

Chaining allows one program to call another program into memory and transfer control to that program. Chaining is an implementation-dependent feature that might not be available on all implementations of Pascal/MT+.

When one program chains to another, the run-time routine loads the new program into the code area and starts execution. Programs pass information by leaving the information in the data area.

To chain programs, you must declare an untyped file (FILE;) and use the ASSIGN and RESET procedures to initialize the file to the name of the new program. You can then execute a call to the CHAIN procedure, passing the name of the file variable as a single parameter. The run-time library routine performs the appropriate functions to load in the file opened with the RESET statement.

There are two ways that chained programs can communicate: shared global variables, and absolute variables.

With the shared global variable method, you must guarantee that at least the first section of global variables is the communication area. You must declare the the shared variables identically so that they have the same location and size in all the chained programs. The remainder of the global variables do not need to be the same in each program. You must use the /D linker option to place the data areas at the same location in each program.

Using the absolute variable method, you typically define a record that is used as a communication area, and then define this record at an absolute location in each module.

To maintain the heap when chaining from one program to another, you must declare the variable SYSMEM as an EXTERNAL INTEGER. SYSMEM contains the address of the top of the heap. The variables:

```
@EFL : INTEGER
@FRL : ARRAY[1..4] OF BYTE
```

contain the information necessary when using FULLHEAP. You can save this information in the global data area and then restore it at the beginning of the program you chain to. You must also use the linker option to give the same address for the global data area to each of the programs that are chained together.

Listings 3-6a and 3-6b lists two example programs that communicate with each other using absolute variables. The first program chains to the second program, which prints the results of the first program's execution.

```
(* PROGRAM #1 IN CHAIN DEMONSTRATION *)

PROGRAM CHAIN1;
TYPE
  COMMAREA = RECORD
               I,J,K : INTEGER
             END;
VAR
  GLOBALS : ABSOLUTE [$8000] COMMAREA;
  (* this address is arbitrary and might not work *)
  (* on your system *)
  CHAINFIL: FILE;

BEGIN (* MAIN PROGRAM #1 *)
  WITH GLOBALS DO
    BEGIN
      I := 3;
      J := 3;
      K := I * J
    END;
  ASSIGN(CHAINFIL,'CHAIN2.COM');
  RESET(CHAINFIL);
  IF IORESULT = 255 THEN
    BEGIN
      WRITELN('UNABLE TO OPEN CHAIN2.COM');
      EXIT
    END;
  CHAIN(CHAINFIL)
END.       (* END CHAIN1 *)
```

**Listing 3-6a.  Chain Demonstration Program 1**

```
(* PROGRAM #2 IN CHAIN DEMONSTRATION *)

PROGRAM CHAIN2;
TYPE
  COMMAREA = RECORD
               I,J,K : INTEGER
             END;
VAR
  GLOBALS : ABSOLUTE [$8000] COMMAREA;

BEGIN (* PROGRAM #2 *)
  WITH GLOBALS DO
    WRITELN('RESULT OF ',I,' TIMES ',J,' IS =', K)

END. (* RETURNS TO OPERATING SYSTEM WHEN COMPLETE *)
```

**Listing 3-6b.  Chain Demonstration Program 2**

End of Section 3

# Section 4
# Run-time Interface

This section explains how to interface Pascal/MT+ programs with the run-time environment and the operating system.  It also explains how to write programs that run without an operating system.

## 4.1  Run-time Environment

Figures 4-1, 4-2, and 4-3 show different the memory maps for a Pascal/MT+ program that has been compiled, linked, and loaded under CP/M.
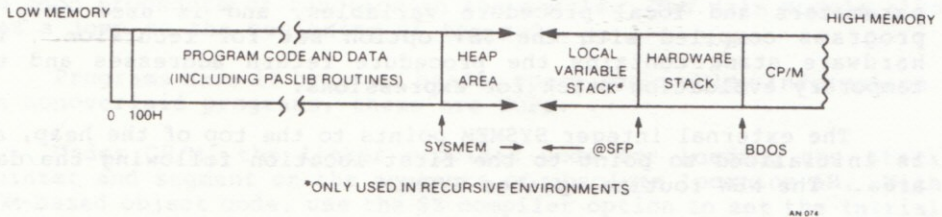


Figure 4-1.  Pascal/MT+ Memory Map at Run-time:
Program Linked without /D Option
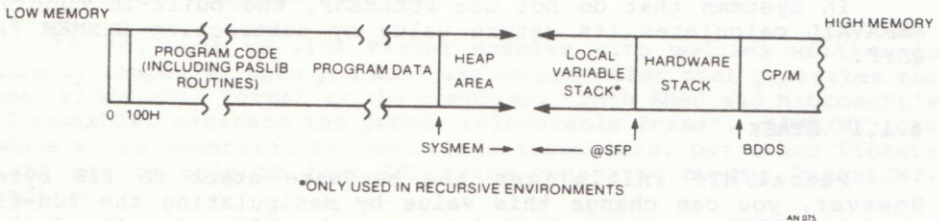


Figure 4-2.  Pascal/MT+ Memory Map at Run-time:
Program Linked with /D Option

LOW MEMORY                                                                    HIGH MEMORY

| ROOT PROGRAM (CODE ONLY) | OVERLAY AREA (S) | ROOT DATA AREA | OVERLAY DATA AREA | HEAP AREA | LOCAL VARIABLE STACK | HARDWARE STACK | CP/M |
|---|---|---|---|---|---|---|---|

0   100H

SPECIFIED BY /V OPTION          SPECIFIED BY /D OPTION          IMPLIED BY /X OPTION

AN 076
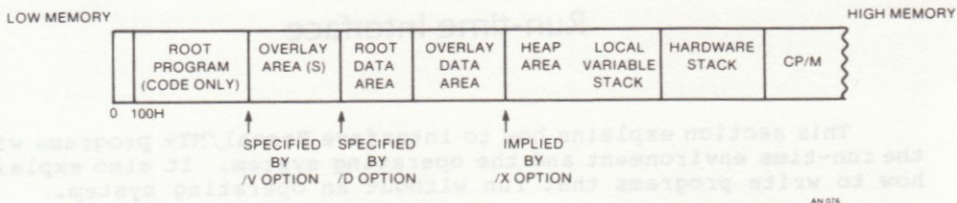
**Figure 4-3.   Pascal/MT+ Memory Map at Run-time:**
**Program with Overlays**

The heap grows toward high memory and the local variable stack grows toward low memory.  The local variable stack contains parameters and local procedure variables, and is used only in programs compiled with the $S+ option set for recursion.  The hardware stack contains the procedure return addresses and the temporary evaluation stack for expressions.

The external integer SYSMEM points to the top of the heap, and is initialized to point to the first location following the data area.  The NEW routine updates SYSMEM.

The external integer @SFP points to the top of the local variable stack, and is initialized to be the top of the hardware stack minus 128 bytes. The routines @LNK (allocate stack frame) and @ULK (deallocate stack frame) update @SFP.

In systems that do not use FULLHEAP, the built-in function MEMAVAIL calculates its return value by subtracting SYSMEM from @SFP.

### 4.1.1   STACK

Pascal/MT+ initializes the hardware stack to 128 bytes. However, you can change this value by manipulating the run-time variable @SFP as an external integer and subtracting the desired additional space, or adding space if you want to make it smaller. The following example illustrates how to do this:

```
        VAR @SFP:EXTERNAL INTEGER;
                        .
                        .
                        .
        (* in main program only!!! *)

        @SFP := @SFP - MORE_HW_STACK_SPACE_IN_BYTES;
```

For a program on an interrupt-driven system, it is often necessary to enlarge the hardware stack.

## 4.1.2  Program Structure

The Pascal/MT+ compiler generates program modules with simple structures.  A jump table at the beginning of each module has jumps to each procedure or function in the module.  The main module also has a jump to the beginning of the code.

Programs have 16 bytes of header space for overlay information. In nonoverlaid programs, these are NOPs.

Under CP/M, the linker provides code for loading the stack pointer and segment on the contents of absolute location 6H.  With ROM-based object code, use the $Z compiler option to set the initial stack pointer for your ROM requirements.  The compiler calls the @INI routine that initializes INPUT and OUTPUT text files.  If you use ROM, you can rewrite the @INI routine to suit your needs.

## 4.2  Assembly Language Routines

If you want to link Pascal modules with modules written in assembly language, then you must use an assembler that generates the same relocatable format as the compiler.  Both RMAC and Microsoft's M80 assembler generate the proper relocatable format.  LINK/MT+ can handle files generated by compatible assemblers, but other linkers might not be able to link ERL files generated by the Pascal/MT+ compiler.

The assemblers and the Pascal/MT+ compiler generate entry point and external reference records in the same relocatable file format. These records contain external symbol names.  The Pascal/MT+ relocatable format allows up to 7 characters in a name, but most assemblers generate 6-character names.  Therefore, you must limit names to 6 characters if you want a variable in a Pascal/MT+ program to be accessible by name to an assembly language routine.

The Pascal/MT+ compiler ignores the underscore character in names.  For example, A_B is the same as AB.  Symbols can begin with $ in M80 and with ? In RMAC.  Neither is a standard character in Pascal/MT+.  Also, M80 considers $ significant; RMAC does not. Thus, M80 places A$B in the relocatable file as A$B; in RMAC, the same symbol goes to the file as AB.  RMAC often uses $ to simulate the underscore, which makes it nontransportable to M80.

### 4.2.1  Accessing Variables and Routines

To access assembly language variables or routines from a Pascal program, you must perform the following steps:

- Declare them PUBLIC in the DATA segment of the assembly language module.

- Declare them EXTERNAL in the Pascal/MT+ program.

To access Pascal/MT+ global variables and routines from an assembly language routine, you must perform the following steps:

- Declare the name EXTRN in the DATA segment of an assembly language program.

- Declare the variable or routine at the global level in the Pascal program.

- Compile the program using the $E+ compiler option.

Listing 4-1 shows how an assembly language module references a variable that is declared in a Pascal/MT+ module.

```
; ASSEMBLY LANGUAGE PROGRAM FRAGMENT

    EXTRN    PQR

    LXI      H,PQR  ;GET ADDR OF PASCAL VARIABLE
              .
              .
              .
    END

(* PASCAL PROGRAM FRAGMENT *)

VAR (* IN GLOBALS *)
    PQR : INTEGER;   (* ACCESSIBLE BY ASM ROUTINE *)
```

**Listing 4-1.  Accessing External Variables**

### 4.2.2  Data Allocation

In the global data area, the compiler allocates variables in the order you declare them. The exception is variables that are in an identifier list before a type. These are allocated in reverse order. For example, given the declaration:

```
A,B,C : INTEGER
```

C is allocated first, then B, then A.

In memory, Pascal/MT+ stores variables together with no space left between one declaration and the next.  For example, given the declaration:

```
A     : INTEGER;
B     : CHAR;
I,J,K : BYTE;
L     : INTEGER;
```

the following storage layout appears:

| byte# | contents |
|-------|----------|
| 0 | A LSB (least significant byte) |
| 1 | A MSB (most  significant byte) |
| 2 | B |
| 3 | K |
| 4 | J |
| 5 | I |
| 6 | L LSB |
| 7 | L MSB |

Arrays are stored in row-major order.  For example, the declaration:

```
A: ARRAY [1..3, 1..3] OF CHAR
```

is stored in the following way:

| byte# | contents |
|-------|----------|
| 0 | A[1,1] |
| 1 | A[1,2] |
| 2 | A[1,3] |
| 3 | A[2,1] |
| 4 | A[2,2] |
| 5 | A[2,3] |
| 6 | A[3,1] |
| 7 | A[3,2] |
| 8 | A[3,3] |

Logically, this is a one-dimensional array of vectors.  In Pascal/MT+, all arrays are logically one-dimensional arrays of some type.

Records are stored like global variables.  Sets are stored as follows:

- Sets are stored as 32-byte items.
- Each element of the set uses one bit.
- Sets are byte oriented.
- The low-order bit of each byte is the first bit in that byte of the set.

The following figure shows the storage for the set A..Z:

Byte number

| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F | 10 | ... | 1F |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----|----|
| 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | FE | FF | FF | 07 | 00 | 00 | 00 | 00 | 00 | ... | 00 |

**Figure 4-4.  Storage for the Set A..Z**

The first bit, bit 65 ($41), is in byte 8, bit 1.  The last bit, bit 90, is in byte 11, bit 2.  Bit 0 is the least significant bit in the byte.

Table 4-1 below summarizes the size and range of Pascal/MT+ data types.

**Table 4-1.  Size and Range of Pascal/MT+ Data Types**

| Data Type | Size | Range |
|-----------|------|-------|
| CHAR | 1 8-bit-byte | 0..255 |
| BOOLEAN | 1 8-bit-byte | false..true |
| INTEGER | 1 8-bit-byte | 0..255 |
| INTEGER | 2 8-bit-bytes | -32768..32767 |
| BYTE | 1 8-bit-byte | 0..255 |
| WORD | 2 8-bit-bytes | 0..65535 |
| BCD REAL | 10 8-bit-bytes | 18 digits, 4 decimal |
| FLOATING REAL | 8 8-bit-bytes | $10^{-17}..10^{17}$ |
| STRING | 1...256 bytes | ----------- |
| SET | 32 8-bit-bytes | 0..255 |

### 4.2.3  Parameter Passing

When you call an assembly language routine from Pascal/MT+ or a Pascal/MT+ routine from assembly language, parameters pass on the stack.

On entry to the routine, the top of the stack is a single word containing the return address.  The parameters are below the return address, in reverse order from declaration.

Each parameter requires at least one 16-bit word of stack space.  A character or Boolean passes as a 16-bit word with a high-order byte of 00.

VAR parameters pass by address.  The address represents the byte of the actual variable with the lowest memory address.

Nonscalar parameters, except sets, always pass by address.  If the parameter is a value parameter, the compiler generates code to call @MVL to move the data.

The @SS2 routine handles set parameters.  If passed by value, the actual value of the set goes on the stack.  Sets are stored on the stack with the least significant byte on top and the most significant byte on bottom.

The following example shows a typical parameter list on entry to a procedure:

```
PROCEDURE DEMO(I,J : INTEGER; VAR Q:STRING; C,D:CHAR);

        STACK--->   0      RETURN ADDRESS
                   +1      RETURN ADDRESS
                   +2      D
                   +3      BYTE OF 00
                   +4      C
                   +5      BYTE OF 00
                   +6      ADDRESS OF ACTUAL STRING
                   +7      ADDRESS OF ACTUAL STRING
                   +8      J (LSB)
                   +9      J (MSB)
                  +10      I (LSB)
                  +11      I (MSB)
```

The assembly language program must remove all parameters from the stack before returning to the calling routine.  This is usually done with an RET n instruction, where n is the number of bytes of parameters.  In the example above, n is 12.

Function values return on the stack.  They are placed below the return address before the function returns.  When the program flow reenters the calling program, the returned value is on the top of the stack.

Assembly language functions can only return the simple types INTEGER, REAL, BOOLEAN, and CHAR.  Assembly language functions cannot return structured types.

## 4.2.4  Assembly Language Interface Example

Listings 4-2 and 4-3 illustrate the interface between a Pascal program and some assembly language routines.

The Pascal/MT+ program performs the PEEK and POKE functions found in BASIC.  The assembly language module simulates the PEEK and POKE.  PEEK returns the byte found at the address passed to it, and POKE puts the byte at the specified address.

```
PROGRAM PEEK_POKE;

TYPE
      BYTEPTR = ^BYTE;

VAR
      ADDRESS : INTEGER;
      CHOICE : INTEGER;
      BBB : BYTE;
      PPP : BYTEPTR;

EXTERNAL PROCEDURE POKE (B : BYTE; P : BYTEPTR);
EXTERNAL FUNCTION PEEK (P : BYTEPTR) : BYTE;

BEGIN
  REPEAT
    WRITE('Address? (use hex for large numbers)  ')'
    READLN(ADDRESS);
    PPP := ADDRESS; {ONLY ALLOWED IN PASCAL/MT+}
    WRITE('1) Peek  OR  2) Poke ');
    READLN(CHOICE);
    IF CHOICE = 1 THEN
      WRITELN(ADDRESS,' contains ',PEEK(PPP))
    ELSE
      IF CHOICE = 2 THEN
        BEGIN
          WRITE('Enter byte of data: ');
          READLN(BBB);
          POKE(BBB,PPP)
        END
  UNTIL FALSE
END.
```

Listing 4-2.  Pascal/MT+ PEEK_POKE Program

```
            PUBLIC PEEK
            PUBLIC POKE

;Peek returns the byte found in the address passed on the stack
;It is declared as an external in a Pascal program as:
;EXTERNAL FUNCTION PEEK(P : BYTEPTR) : BYTE


PEEK:
            POP B       ;RETURN ADDRESS INTO BC
            POP D       ;POINTER TO BYTE INTO HL
            POP E,M     ;MOVE CONTENTS OF MEMORY POINTED TO BY HL INTO E
            MVI D,0     ;PUT A 00 INTO D
            PUSH D      ;RETURN FUNCTION VALUE
            PUSH B      ;PUT RETURN ADDRESS ON STACK
            RET         ;RETURN TO CALLER (NO PARAMETERS LEFT ON STACK)

;Poke places a byte into memory
;It is declared as an external in a Pascal program as:
;EXTERNAL PROCEDURE POKE(B : BYTE; P : BYTEPTR);

POKE:
            POP B       ;GET RETURN ADDRESS INTO BC
            POP H       ;THE BYTE POINTER IS PUT INTO HL
            POP D       ;REGISTER E GETS THE BYTE, D GETS THE EXTRA BYTE OF 00

            MOV M,E     ;PUT E INTO MEMORY POINTED TO BY HL

            PUSH B      ;RETURN ADDRESS ON TOP OF STACK
            RET         ;RETURN TO CALLER (NO PARAMETERS LEFT ON STACK)

            END
```

**Listing 4-3.  Assembly Language PEEK and POKE Routines**


## 4.3  Pascal/MT+ Interface Features

Pascal/MT+ provides several features that let you control your program's environment.  The following features are explained in this section:

- direct access to the operating system
- machine code inserted into Pascal source
- variables with absolute addresses
- interrupt procedures
- heap management

### 4.3.1  Direct Operating System Access

You can make BDOS function calls to the operating system by using the @BDOS routine.  You declare it in a Pascal/MT+ program as follows:

    EXTERNAL FUNCTION @BDOS(FUNC:INTEGER; PARM:WORD):INTEGER;

The first parameter is the BDOS function number.  The use of the second parameter depends on the specific function number.  Refer to your particular operating system's documentation for the list of functions.

The following example shows KEYPRESSED, a function that uses the @BDOS function.  KEYPRESSED returns TRUE if a key is pressed, FALSE if not.

    FUNCTION KEYPRESSED : BOOLEAN;

    BEGIN
       KEYPRESSED := (@BDOS(11,0) <> 0)
    END;

Listings 4-4 and 4-5 illustrate calls to BDOS function 6 and 23, respectively.

```
(* DEMO OF USING BDOS FUNCTION CALL 6 FOR CONSOLE IO *)

PROGRAM BDOS6;
VAR

  CH : CHAR;

  I : INTEGER;

  EXTERNAL FUNCTION @BDOS(FUNC:INTEGER; PARM:WORD):INTEGER;

BEGIN (* ECHO ANY INPUT CHARACTER TO THE CONSOLE UNTIL A : IS READ *)
REPEAT
  CH:=CHR(@BDOS(6,WRD($FF))); (* READ CHARACTER *)
  IF CH <> ':' THEN
    BEGIN
    I:=@BDOS(6,WRD(CH));  (* WRITE CHARACTER *)
    END;
  UNTIL CH= ':';
END.
```

## Listing 4-4.  Calling BDOS Function 6

```
(* DEMO OF USING BDOS FUNCTION CALL 23 TO RENAME FILES *)

PROGRAM BDOS23;
TYPE
   FCBLK = PACKED ARRAY [0..36] OF CHAR;
   X = FILE;

VAR
   F1 : X;
   F2 : FCBLK;
   I  : INTEGER;
   OLDNAME,NEWNAME : STRING;

   (* EXTRACT IS A PROCEDURE TO FETCH THE FILE NAME INTO THE STRING *)
   (* IT IS A MODIFIED VERSION OF THE PROCEDURE IN UTILMOD. *)
   (* THIS VERSION RETURNS THE FILE NAME FORMATTED FOR CPM *)
   EXTERNAL PROCEDURE EXTRACT(VAR F:X; NAME:STRING);

   EXTERNAL FUNCTION @BDOS(FUNC:INTEGER; PARM:WORD):INTEGER;

BEGIN
   WRITE('ENTER OLD FILE NAME: ');   (* GET THE OLD FILE NAME *)
   READLN(OLDNAME);
   ASSIGN(F1,OLDNAME);               (* USE ASSIGN TO CONVERT THE STRING *)
                                     (* TO A VALID CPM FILE NAME *)
   EXTRACT(F1,OLDNAME);              (* USE THE UTILITY PROCEDURE EXTRACT *)
                                     (* TO RETRIVE THE FORMATED FILE NAME *)
6  MOVE(OLDNAME,F2,12);              (* MOVE IT TO THE FCB USED BY BDOS CALL 23 *)
   OLDNAME[0] := CHR(12);            (* EXTRACT DOES NOT RETURN THE LENGTH *)
   CLOSE(F1,I);                      (* SO WE CAN USE IT FOR NEWNAME *)

   WRITE('ENTER NEW FILE NAME: ');   (* GET THE NEW FILE NAME *)
   READLN(NEWNAME);
   ASSIGN(F1,NEWNAME);               (* CONVERT IT TO A CPM FORMATTED FILE NAME *)
   EXTRACT(F1,NEWNAME);
   MOVE(NEWNAME,F2[16],12);          (* MOVE IT TO THE FCB FOR BDOS CALL 23 *)
   NEWNAME[0] := CHR(12);            (* MOVE IN THE LENGTH *)

   (* CALL THE RENAME FUNCTION. PASS A POINTER TO THE FCB *)
   (* CONTAINING THE OLD AND NEW FILE NAMES *)
   IF @BDOS(23,WRD(ADDR(F2))) = 255 THEN
      WRITELN('RENAME FAILED. ',OLDNAME,' NOT FOUND.')
   ELSE
      WRITELN('FILE ',OLDNAME,' RENAMED TO ',NEWNAME);

END.
```

**Listing 4-5.  Calling BDOS Function 23**

## 4.3.2 INLINE

INLINE is a built-in feature that lets you 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.

INLINE syntax is similar to that of a procedure call:

● The word INLINE is followed by a left parenthesis.

● After the parenthesis come any number of arguments.

● Arguments must be constants, or variable references that evaluate to constants.

● Arguments can be of types CHAR, STRING, BOOLEAN, INTEGER, or REAL.

● Separate the arguments with slashes.

● The arguments end with a right parenthesis.

Note that a string in single apostrophes does not generate a length byte, but simply the data for the string.

The address of a variable evaluates to the absolute data address, unless the program is set up to run with recursion. Then the address is the offset into the appropriate stack frame.

Literal constants of type integer are allocated one byte if the value falls in the range 0 to 255. Named integer constants always get two bytes.

The Pascal/MT+ system features a built-in mini-assembler for 8080/8085 CPUs. The compiler translates a double quote followed by an assembly language mnemonic into a hexadecimal value. For example,

```
"MOV A,M
```

translates as $7E. Appendix E contains a complete list of the valid opcodes for the mini-assembler. The following example illustrates INLINE:

```
INLINE( "LHD  /      (*LHD OPCODE FOR 8080*)
         VAR1 /      (*REFERENCE VARIABLE*)
        "SHLD /      (*SHLD OPCODE FOR 8080*)
         VAR2 /      (*REFERENCE VARIABLE*)
```

To facilitate branching, the syntax *+n and *-n, (where n is an integer), is included as legal operand to INLINE.  For example,

```
INLINE("IN / $03/
       "ANI/ $02/
       "JNZ/ *-4 );
```

The location that the * references is the previous opcode, not the address of the * character.

The following listing uses INLINE in a procedure that calls CP/M and returns a value.  This routine is @BDOS in the run-time library PSALIB.

```
           FUNCTION @BDOS(FUNC:INTEGER; PARM:WORD):INTEGER;
     CONST
       CPMENTRYPOINT = 5;       (* SO IT ALLOCATES 2 BYTES *)
     VAR
       RESULT : INTEGER;        (* SO WE CAN STORE IT HERE *)

     BEGIN
       INLINE( $2A / FUNC /         (* LHLD FUNC    *)
               $4D /                (* MOV C,L      *)
               $2A / PARM /         (* LHLD PARM    *)
               $EB /                (* XCHG         *)
               $CD / CPMENTRYPOINT / (* CALL BDOS   *)
               $6F /                (* MOV L,A      *)
               $26 / $00 /          (* MVI H,0      *)
               $22 / RESULT );      (* SHLD RESULT *)

       @BDOS := RESULT;         (* SET FUNCTION VALUE *)
     END;
```

**Listing 4-6.  Using INLINE in @BDOS**

The following listing uses INLINE to construct a compile-time table.  The table is the entire body of a procedure.  By getting the address of the procedure, the program can access the table.  Notice that the dummy procedure is not intended to be an executable procedure, and that the table is treated as code.

```
PROGRAM DEMO_INLINE;

TYPE
    IDFIELD = ARRAY [1..4] OF ARRAY [1..10] OF CHAR;

VAR
    TPTR : ^IDFIELD;

PROCEDURE TABLE;
BEGIN
    INLINE(    'DIGITAL   ' /
               'RESEARCH  ' /
               'SOFTWARE  ' /
               'TOOLS.....' );
END;


BEGIN (* MAIN PROGRAM *)
    TPTR := ADDR(TABLE);
    WRITELN(TPTR^[3]) (* SHOULD WRITE 'SOFTWARE  ' *)
END.
```

**Listing 4-7.  Using INLINE to Construct a Compile-time Table**


The address of the procedure is the address of the table only in a static environment.  If you compile the program with the $Q+ option for recursion, the compiler generates extra code at the beginning of the procedure for recursion management.  The compiler generates six extra bytes if the $Q option is set, and five extra bytes if the option is not set.

**Note:**  the table must be in the same module as the statement that calls ADDR.

### 4.3.3  Absolute Variables

You can declare ABSOLUTE variables if you know the address at compile-time.  The following examples show the special syntax for declaring absolute variables:

```
I     : ABSOLUTE [$8000] INTEGER;
SCREEN: ABSOLUTE [SCRN_AD] ARRAY[0..15, 0..63] OF CHAR;
```

Note that you must put the address of the variable in brackets [...].  The address must be a constant, either named or literal.

The compiler does not allocate space in the data area for ABSOLUTE variables.  Make sure no compiler-allocated variables conflict with the absolute variables.

String variables cannot be stored at all locations.  On the 8080, strings must be between 100H and FFFFH, so that the run-time routines can distinguish between a string address and a character on top of the stack.

### 4.3.4  Interrupt Procedures

Pascal/MT+ has a special procedure type to handle interrupts. When an interrupt occurs, the procedure associated with that particular interrupt is invoked; you do not call interrupt procedures from the program. When the interrupt procedure finishes, control returns to where it was interrupted.  You select the vector to be associated with each interrupt.

You declare an interrupt procedure as follows:

    PROCEDURE INTERRUPT [ <vec num> ] <procname> ;

Interrupt procedures can exist only in the main program, so that the interrupt vectors can load correctly.  At the beginning of the program, the compiler generates code to load the vector with the procedure address.

For 8080/Z80 systems, the vector number range is 0 to 7.  For Z80 mode 2 interrupts, allocate an interrupt table by declaring an ABSOLUTE variable, and use the ADDR function to fill in the table. Use INLINE in a Z80 environment to initialize the I register.

The compiler generates code to push the registers on entering an interrupt procedure, and to pop the registers and reenable interrupts on exiting the procedure.  Because many interrupt modes are possible on the Z80, the Z option does not generate the Z80 'RET1' instruction.

**Note:** you must initialize the interrupt vectors.  The compiler does not generate code to store in the absolute locations occupied by the interrupt vector table.

Interrupt procedures cannot have parameter lists, but can have local variables and can access global variables.

The Pascal/MT+ system does not generate reentrant code. Typically, interrupt procedures set global variables but do not perform other procedure calls or I/O.  For this reason, you should avoid sets, strings, procedure calls, and file I/O. You should also avoid calling CP/M and routines in the run-time packages that include data.  If you use CP/M, notice that I/O through the CP/M BDOS typically reenables interrupts.

To disable interrupts around sections of Pascal code, use INLINE and the mini-assembler to place EI (enable interrupt) and DI (disable interrupt) instructions around the code.

The following program illustrates interrupt procedures.  The program waits for one of four switches to interrupt and then toggles the state of a light attached to the switch.  The I/O ports for the lights are 0 to 3, and the switches use interrupt restarts 2, 3, 4, and 5.

```
PROGRAM INT_DEMO;
CONST
  LIGHT1 = 0;          (* DEFINE I/O PORT CONSTANTS *)
  LIGHT2 = 1;
  LIGHT3 = 2;
  LIGHT4 = 3;

  SWITCH1 = 2;         (* DEFINE INTERRUPT VECTORS *)
  SWITCH2 = 3;
  SWITCH3 = 4;
  SWITCH4 = 5;

VAR

  LIGHT_STATE : ARRAY [LIGHT1..LIGHT4] OF BOOLEAN;
  SWITCH_PUSH : ARRAY [LIGHT1..LIGHT4] OF BOOLEAN;

  I : LIGHT1 .. LIGHT4;

PROCEDURE INTERRUPT [ SWITCH1 ] INT1;
BEGIN
  SWITCH_PUSH[LIGHT1] := TRUE
END;

PROCEDURE INTERRUPT [ SWITCH2 ] INT2;
BEGIN
  SWITCH_PUSH[LIGHT2] := TRUE
END;

PROCEDURE INTERRUPT [ SWITCH3 ] INT3;
BEGIN
  SWITCH_PUSH[LIGHT3] := TRUE
END;

PROCEDURE INTERRUPT [ SWITCH4 ] INT4;
BEGIN
  SWITCH_PUSH[LIGHT4] := TRUE
END;
```

**Listing 4-8.  Using Interrupt Procedures**

```
BEGIN (* MAIN PROGRAM *)

  (* INITIALIZE BOTH ARRAYS *)

  FOR I := LIGHT1 TO LIGHT4 DO
    BEGIN
      LIGHT_STATE[I] := FALSE;  (* ALL LIGHTS OFF *)
      SWITCH_PUSH[I] := FALSE;  (* NO INTERRUPTS YET *)
    END;


  REPEAT

    REPEAT        (* UNTIL INTERRUPT *)
    UNTIL SWITCH_PUSH[LIGHT1] OR SWITCH_PUSH[LIGHT2] OR
          SWITCH_PUSH[LIGHT3] OF SWITCH_PUSH[LIGHT4];

    FOR I := LIGHT1 TO LIGHT4 DO (* SWITCH LIGHTS *)
      IF SWITCH_PUSH[I] THEN
        BEGIN
          SWITCH_PUSH[I] := FALSE;
          LIGHT_STATE[I] := NOT LIGHT_STATE[I]; (* TOGGLE IT *)
          OUT[I] := LIGHT_STATE[I]
        END

  UNTIL FALSE; (* FOREVER DO THIS LOOP *)

END. (* OF PROGRAM *)
```

**Listing 4-8.  (continued)**


### 4.3.5  Heap Management

You can manage the heap two ways:

1) Use the ISO standard routines as they are implemented in
   FULLHEAP.ERL.  When you use this method:

   ● the NEW routine uses a standard heap.

   ● dynamic data goes to the smallest space that can hold the
     requested item.

   ● the DISPOSE routine disposes the item passed to it.

   ● when necessary, MAXAVAIL, or NEW gathers free memory into
     a free list, combines adjacent blocks, and reports the
     largest available block of memory.

   ● MEMAVAIL returns the largest never-allocated memory
     space.

2) Use NEW, DISPOSE, and MEMAVAIL, which are part of the
   PASLIB.ERL run-time library.  When you use this method:

   ● the heap is treated as a stack.

   ● NEW puts the dynamic data on top of the stack.

   ● the stack grows from the end of the static data towards
     the hardware stack.

   ● DISPOSE performs no function, but is included for symbol
     table use.

   ● you can simulate the MARK and RELEASE routines of UCSD
     Pascal™ by using the system integer SYSMEM, which points
     to the top of the heap, as shown in the following
     example:

```
     MODULE UCSDHEAP,

     VAR
           SYSMEM : EXTERNAL INTEGER;

     PROCEDURE MARK(VAR P:INTEGER);
     BEGIN
           P := SYSMEM
     END;

     PROCEDURE RELEASE(P:INTEGER);
     BEGIN
           SYSMEM := P
     END;

     MODEND.
```

## 4.4  Recursion and Nonrecursion

Pascal/MT+ does not automatically produce recursive code,
because recursion increases overall code size and decreases
execution speed.  You can generate recursive code with the S
compiler source code option (see Section 2.2.4).

When using recursion, return addresses for all procedures are
stored on the hardware stack.  If recursion is deeply nested, the
default stack size of 128 bytes might be too small.  If so, the
program can overwrite local or global data as recursion continues.
You can solve this problem by modifying @SFP, as described in
Section 4.1.

## 4.5  Stand-alone Operation

If you want to run Pascal/MT+ programs in a ROM-based system, perform the following steps:

1) Use the $Z compiler option to tell the compiler where to initialize the hardware stack pointer.

2) If the program performs I/O you have three choices:

- Use redirected I/O for all READ and WRITE statements. This replaces the run-time character I/O routines with user-written I/O routines. Refer to the Pascal/MT+ Language Reference Manual.

- Rewrite GET and the run-time routines @RNC and @WNC. @RNC is the read-next-character routine; @WNC is the write-next-character routine. You must rewrite GET because the read-integer and read-real routines call it.

- Build a simulated CP/M BDOS in your PROM. If you are constructing your program to run in a totally stand-alone environment, such as an Intel SBC-80/10 board, you can write an assembly language module to link in front of your program.

  This routine can jump around the standard code that simulates the BDOS, and can simulate the CP/M BDOS for functions 1: Console Input, 2: Console Output, and 5: List Output.

  The function number is in the C register; the data for output is in E. For input (Function 1), return the data in the A register. All registers are free to use, and the stack contains nothing but the return address.

  Note: this is just a suggestion; Digital Research does not give detailed application support for this method.

3) You can shorten or eliminate the INPUT and OUTPUT FIB storage in the @INI module. You need this storage for TEXT file I/O compatibility, but you might not need it in a ROM-based environment.

  Make sure any changes to INPUT and OUTPUT are also handled in @RST (read a string from a file) and @CWT (read until EOLN is true on a file).

  The distribution disk includes three skeletons for the @INI, @RNC, GET, and @WNC routines that you can use in ROM environments.

If your program does any reads or writes and does not use the heap or overlays, you can rewrite the @INI procedure in your program as follows:

```
PROCEDURE @INI;
BEGIN
END;
```

4) In ROM environment, you cannot use the PROCEDURE INTERRUPT [vector] construct to handle interrupts. You must construct an assembly language module and link it as the main program (first file). This module must contain JMP instructions at the interrupt vector locations to jump to the Pascal/MT+ interrupt routines.

   **Note:** find the interrupt routines with the /M linker option.

5) The integer- and real-divide routines contain a direct call to CP/M for the divide by 0 error message. If there is a possibility of that error ocurring in your program, modify the routine in DIVMOD.MAC, which is on your distribution disk #2.

6) Link any changed run-time routines before linking the run-time library to resolve the references, making sure to use the /S option, as in the following example:

   **A>LINKMT USERPROG,MYWNC,MYRNC,GET,MYINI,PASLIB/S**

7) Strings cannot reside below 100H. If you have any constant strings, named or literal, at the beginning of your program, fill out the remaining space in the first PROM with a table, or with a DS to get the Pascal/MT+ program to exist at locations greater 100H. Remember, if you put tables or data first, you must jump around them to begin execution of the Pascal/MT+ program, starting with its first byte.

## 4.6 Error and Range Checking

The Pascal/MT+ system supports two types of run-time checking: range checking and exception checking. The default state of the compiler disables range checking and enables exception checking.

Error checks and routines set Boolean flags. These flags, along with an error code, load onto the stack and call the built-in routine @ERR, which tests the Boolean flag.

If no error occurs, the flag is FALSE, so @ERR exits to the compiled code and continues execution. If an error occurs, @ERR acts appropriately, as described in Table 4-2.

**Table 4-2.  @ERR Routine Errors**

| Value | Meaning |
|:---:|:---|
| 1 | Divide by 0 check |
| 2 | Heap overflow check (unused, see below) |
| 3 | String overflow check (unused, see below) |
| 4 | Array and subrange check |
| 5 | Floating point underflow |
| 6 | Floating point overflow |
| 7 | 9511 transcendental error |

### 4.6.1  Range Checking

Range checking monitors array subscripts and subrange assignments.  It does not check when you read into a subrange variable.

When range checking is enabled, the compiler generates calls to @CHK for each array subscript and subrange assignment.  The @CHK routine leaves a Boolean value on the stack and the error code number 4.  The compiler generates calls to @ERR after the @CHK call.  If an error occurs, @ERR asks you whether it should continue or abort.

When range checking is disabled, and an array subscript falls outside the valid range, you get unpredictable results.  For subrange assignments, the value truncates at the byte level.

### 4.6.2  Exception Checking

Exception checking is enabled by default.  In the current release, the $X- compiler option does not disable exception checking.  The conditions checked for are the following:

- integer and real numbers divided by 0
- real number underflow and overflow
- string overflow

The various exceptions produce the following results:

- Floating-point underflow:  @ERR does not print a message.  The result of the operation is 0.0.

- Floating-point overflow:   the result of the operation is a large number.

● Division by zero:  the result is the largest possible number.

● Heap overflow:  the error processor takes no action.

● String overflow:  the string is truncated.

### 4.6.3  User-supplied Handlers

You can write your own @ERR routine instead of using the system routine.  Declare the routine as follows:

        PROCEDURE @ERR(ERROR:BOOLEAN;  ERRNUM:INTEGER);

Your version of @ERR should check the ERROR variable and exit if it is FALSE.  If the value is TRUE, you can decide what action to take.

To use @ERR instead of the routine in PASLIB, link your routine ahead of PASLIB to resolve the references to @ERR.  The values of ERRNUM are in Table 4-2.

### 4.6.4  I/O Error Handling

The run-time routine, @BDOS, does not handle I/O errors. However, it returns the CP/M error code in IORESULT.  You can rewrite @BDOS, as described below, to check further for disk I/O errors.

XBDOS.SRC on distribution disk #2 contains an alternative @BDOS routine.  When XBDOS calls the BDOS with the CP/M I/O functions OPEN, RESET, CLOSE, WRITE, and REWRITE, it generates a call to IOERR, and passes the CP/M function call number.  You can then modify the IOERR routine, found in IOERR.SRC on distribution disk #2, to handle these I/O errors.

To use the I/O error handling code, compile both IOERR.SRC and XBDOS.SRC.  Then use the file named IOCHK.BLD on distribution disk #2 as input to LIBMT+.  IOCHK.BLD uses the relocatable files and creates a library called IOCHK.ERL.  You must link this library before PASLIB.  You cannot search IOCHK.ERL because all references to @BDOS are generated by PASLIB.

You do not have to declare @BDOS of IOERR external, because all the references to @BDOS come from PASLIB, and all the references to IOERR come from @BDOS.

<div align="center">End of Section 4</div>

# Section 5
# Pascal/MT+ Programming Tools

Pascal/MT+ provides three programming tools designed to increase programming productivity: a disassembler, a symbolic debugger, and a librarian.

● DIS8080 is a disassembler that combines a relocatable file with a corresponding PRN file to produce a file showing the assembly code for each Pascal/MT+ source line.

● The debugger is a relocatable file that you link into a program, enabling you to step through the program as it runs.

● LIBMT+ is a librarian utility that concatenates relocatable files into a searchable library file.

## 5.1 DIS8080, the Disassembler

The disassembler DIS8080 consists of one executable file, DIS8080.COM, which is on your Pascal/MT+ distribution disk #2.

DIS8080 generates a file showing the assembly language for each Pascal/MT+ source line. When you compile a program using the X option, the compiler generates an extended relocatable file with filetype ERL containing assembly language coding interspersed with Pascal/MT+ statements.

When you compile a program using the P option, the compiler generates print files with filetype PRN. Used together, these files enable the disassembler to investigate code the compiler produces. The files provide the information necessary to debug the program at the machine code level.

**Note:** because most of the compiler code is 8080 code, a disassembler for 8080 mnemonics comes only with CP/M releases.

Appendix C contains a listing of a sample disassembly. Figure 5-1 illustrates the operation of DIS8080.
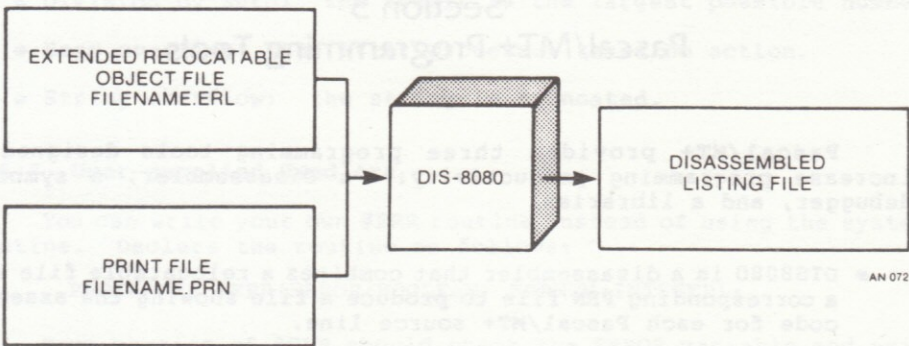
**Figure 5-1.  DIS8080 Operation**

You invoke the disassembler with a command line of the following form:

```
DIS8080<filename>[<destination name>][,L=nnn]
```

You do not have to specify a filetype.  DIS8080 searches for both the ERL and PRN file with the specified <filename>.  Both files must be on one logical disk drive.  The <destination name> can be a filename or a Pascal/MT+ logical device, CON: or LST:.  The default destination is CON:.  The L=nnn parameter enables you to specify the number of lines per page for the output device.  The nnn stands for an integer value.  The L=nnn parameter requires that you specify a destination name.

When the disassembler finds something unexpected in the ERL file, it generates an error message.  Continuing at this point produces more errors because the sequence is off.  An ERL file should have no errors.  To correct errors, recompile the program using the X compiler option, and be sure you are disassembling Pascal/MT+ code only.

## 5.2  The Debugger

The Pascal/MT+ debugger simplifies program maintainance.  The debugger consists of one relocatable object file, DEBUGGER.ERL, which is on distribution disk #2.

To use the debugger, you must link the DEBUGGER.ERL file into a source program along with the run-time support library, PASLIB.ERL.  The debugger then takes charge of the source program execution.

The debugger can perform the following tasks:

- display variables by name or address
- set symbolic breakpoints
- step through the program one statement at a time
- display symbol tables
- display entry and exit points for procedures and functions

The debugger displays line numbers in trace mode. However, in programs consisting of modules, line numbers repeat in each module. The debugger works only on programs without overlays.

You can use the debugger in a stand-alone environment. When the debugger requests the filename of the symbol table, press RETURN to disable the symbolic facilities. The display-by-address facilities remain in effect.

Appendix D shows a sample debugging session.

## 5.2.1   Debugging Programs

When you compile a program with the D option, the compiler generates a PSY file containing debugger information. You must compile all modules that you want to debug with the D option. The compiler writes the PSY file onto the disk containing the corresponding ERL file.

The PSY file contains records for each procedure, function, and variable in the program. The compiler generates code at the beginning and end of each item for debugger breakpoint logic. Address fields for each item are module relative.

The linker uses the ERL and PSY file to create a SYP file containing absolute addresses for each procedure, function, and variable. The debugger uses the SYP file to perform the various debugging tasks.

You must place the DEBUGGER.ERL file first in the list of files in the LINK/MT+ command line. The following example links the debugger, user program, and run-time library into an executable file named PROG1.COM.

        A>**LINKMT PROG1=DEBUGGER,PROG1,PASLIB/S**

The preceding example generates two undefined symbols, @XOP and @WRL. These are required only if PROG uses real numbers. If so, you must link the real number run-time library FPREALS.ERL with the other files in the command line.

To start the debugging session, run the program.  The debugger takes control, and requests the name of the symbol table file.  You must enter the user program SYP file.  You must enter both the filename and filetype.  Press RETURN if there is no symbol table. The debugger then prompts you for the BEgin or TRace command.  You can then proceed to debug the program using breakpoints and other debugger commands.

## 5.2.2  Debugger Commands

Debugger commands use the following rules and syntax elements:

- <name> refers to a variable name, a procedure or function name, or a prefixed variable name.  A prefixed variable name is a variable identifier prefixed with a procedure or function name. Names are from one to eight characters long and follow the syntax of the compiler.

- <num> refers to a decimal or hexadecimal number.  Hexadecimal numbers are prefixed with a $ and range from 0 to FFFF. Decimal numbers range from 0 to 32767.

- <parm> refers to a parameter.

- Specify an offset from the primary address with a + or -.  The debugger assumes + if not specified in the command.

- The ^ is an indirection character used with pointer variables. The ^ tells the debugger to display the data pointed to, not the contents of the pointer itself.

- The debugger ignores underscores, _.  Use underscores to make commands easier to read.

Several commands require an additional parameter.  Parameters have the following syntax:

        <parm> ::= [<name>|<num>|{^}  {[+|-] <num>}

Table 5-1 shows examples of parameters, given the following declarations:

```
TYPE
    PAOC = ARRAY [1..40] OF CHAR;

VAR
    ABC : INTEGER;
    PTR : ^PAOC;
```

**Table 5-1.  Examples of Parameters**

| Parameter | Meaning |
|-----------|---------|
| ABC | the value of variable ABC |
| PTR | the value of PTR |
| PTR^ | the array pointed to by PTR |
| ABC+10 | 10 bytes past ABC location |
| PTR^+10 | PTR^[11] |
| ABC-3 | 3 bytes before ABC |
| PTR^-3 | 3 bytes before the array, PAOC |
| $3FFD | Absolute location |
| $423B^ | 32 bytes pointed to by $423B |
| $3FFD+$5B | 32 bytes at $4058 |
| $423B^+49 | 32 bytes pointed to by contents of $423B + 49 |
| PROC1:I | local variable in PROC1 |
| PROC2:J^+9 | offset from local pointer |

The following displays a variable by <name>:

        DV <parm>{^}

If <name> is a pointer variable, DV displays the contents of the pointer.  If you use <name>^, DV displays the contents of the location addressed by the pointer.

Table 5-2 shows commands used when symbols are not available or when you want to display fields within record or array elements.  If symbols are available, you can use the the commands, but DV is easier to use.

**Table 5-2.  Debugger Display Commands**

| Command Syntax | Meaning |
|----------------|---------|
| DV <symbol> | Display Variable |
| DI <parm> | Display Integer |
| DC <parm> | Display Character |
| DL <parm> | Display Logical (Boolean) |
| DR <parm> | Display Real |
| DB <parm> | Display Byte |
| DW <parm> | Display Word |
| DS <parm> | Display String |
| DX <parm> {,num} | Display extended (structures).  This is always displayed in HEX/ASCII format.  Num is the size, in bytes, for memory dump.  The default value is 320 bytes. |

The following command alters the contents of a memory address:

    SE<parm>

The SE command displays the byte at the specified address in
decimal.  Enter a new value in either decimal or hexadecimal, then
press RETURN.  The new value replaces the displayed value, and the
debugger displays the next byte of memory.  If you enter a value
that does not fit in two bytes, the debugger uses the last two
digits.   To end the SE<parm> command, enter a period and press
RETURN.

Table 5-3 describes the other commands that enable control of
your program in a debugging session.

**Table 5-3.  Debugger Control Commands**

| Command Syntax | Meaning |
|---|---|
| BE | begins execution (start program from beginning). |
| DV <name> | displays the contents of the named variable. |
| E+ | enables display entry and exit of each procedure or function during execution (default on). |
| E- | disables entry / exit display. |
| GO | continues execution from a breakpoint. |
| PN | displays procedure names from SYP file. |
| RB <name> | removes breakpoint at procedure <name>. |
| SB <name> | sets breakpoint at beginning of procedure <name>. |
| SE <parm> | modifies contents of memory at <parm>. A period terminates this command. |
| TR or T | Trace – executes one line and returns. |
| T<num> | traces <num> lines and return. |
| VN <name> | displays variables associated with procedure <name>. |
| ?? | HELP! List of commands is found in DBUGHELP.TXT. |

## 5.3  LIBMT+, the Software Librarian

LIBMT+ is a software librarian program that performs two functions:

- It can logically concatenate ERL files together to construct a searchable library, such as PASLIB.

- It can also convert Pascal/MT+ ERL files that are compatible with Microsoft-compatible linkers, such as L80 and LINK-80™.

You invoke LIBMT+ with a command line of the form:

       LIBMT <filename>

where the <filename> contains only the name, not the type of the file. LIBMT+ accepts an input file of type BLD. A filetype of BLD contains an output filename followed by a list of input filenames, with each name on a separate line.

Pascal/MT+ modules, libraries, and appropriate assembly language modules are all valid as input files. You must specify the filetype but it need not be ERL. If the output file is to be processed by LINK/MT+, it must be of type ERL.

**Note:** LIBMT+ cannot process a Pascal/MT+ module compiled with the X (Extended Relocatable file) option. To process such a module, you must recompile it without the X option.

The following is an example of a BLD file for creating a LINK/MT+ compatible library:

       MYLIB.ERL
       MYMOD1.ERL
       MYMOD2.ERL
       MYMOD3.ERL

This file deletes any existing copy of MYLIB.ERL. It then concatenates the files MYMOD1.ERL, MYMOD2.ERL, and MYMOD3.ERL and places the output in MYLIB.ERL.

### 5.3.1  Searching a Library

The LINK/MT+ linker is a one-pass linker, so when you use the /S option to signify that a file is a library, the linker loads only those modules that have been referenced by previous modules. Therefore, the order of modules in your library is important. If the modules are concatenated as A, B, C, then modules B and C cannot contain references to module A unless they are guaranteed that module A is loaded. Module A, however, can contain references to B or C because this causes the linker to load them.

Remember that the linker can only extract entire modules from a library.  Single procedures from a modules cannot be extracted.  All entry points, both code and data, are used as a basis for searching when you use the /S option.  Only one entry point in a module need be referenced to force loading that entire module.

You cannot use LIBMT+ to alter PASLIB because of its special construction.  If you want to replace modules in PASLIB.ERL, link the replacement modules before linking PASLIB.  This resolves references to those routines before PASLIB is searched.  If the replacement routines are in a library, it is a good idea not to search the new library, because the references to the replacement routines sometimes are not made until PASLIB is searched.

### 5.3.2  LIBMT+ as a Converter to L80 Format

If the first line of the BLD file contains only L80 (or l80), the output file is L80-compatible; otherwise, it is compatible with LINK/MT+.

An L80-compatible file does not work with LINK/MT+.  The following is a sample BLD file for converting a library or module to L80 format:

```
L80
MYLIB.ERL
MYMOD.ERL
MYMOD2.ERL
MYMOD3.ERL
```

LIBMT+ creates a file called MYLIB.ERL, which contains the converted MYMOD1, MYMOD2, and MYMOD3.  The conversion process truncates all public names to six characters.  This can cause duplicate symbol errors when using L80 that did not occur when using LINK/MT+.  LINK/MT+ allows public names up to seven characters long.

The features gained by using this program and L80 are

- the ability to use multiple origins of code and data
- the ability to have initialized data in the DSEG
- the ability to use COMMON

The features of the Pascal/MT+ system lost when using this program and L80 are

- Overlays

- The ability to generate a HEX file

- The /D option of L80 reserves space in memory and writes uninitialized data to the disk, which can result in a very large COM file.

● Seven-character significance in public names.

● The disassembler does not work with REL files.

● The /F option (CMD files) cannot be used.

● Programs that link properly with LINK/MT+ might not link with L80 because they are too large to fit into memory at link-time.

● Unlike LINK/MT+, if you specify /P:4000 when using L80, the area from 100H through 3FFF is also saved in the COM file. LINK/MT+ saves the byte that is loaded at 4000H as the first byte in the COM file. This has both advantages and disadvantages.

● The Pascal feature, temporary files, does not operate with L80.

● Programs that work with LINK/MT+ might suddenly stop working with L80. If the /D option is not used, then all data is initialized to 00 by LINK/MT+. Therefore, you must watch out for uninitialized variables.


End of Section 5

# Appendix A
# Compiler Error Messages

**Table A-1. Compiler Error Messages**

| Message | Meaning |
|---|---|
| Recursion stack overflow | Evaluation stack collision with symbol table. Correct by reducing symbol table size, simplifying expressions. |
| Error # 1 Error in simple type | Self-explanatory. |
| Error # 2 Identifier expected | Self-explanatory. |
| Error # 3 'PROGRAM' expected | Self-explanatory. |
| Error # 4 ')' expected | Self-explanatory. |
| Error # 5 ':' expected | Possibly a = used in a VAR declaration. |
| Error # 6 Illegal symbol (possibly missing ';' on line above) | Symbol encountered is not allowed in the syntax at this point. |

**Table A-1.** (continued)

| Message | Meaning |
|---------|---------|
| Error # 7<br>Error in parameter list | Syntactic error in parameter list declaration. |
| Error # 8<br>'OF' expected | Self-explanatory. |
| Error # 9<br>'(' expected | Self-explanatory. |
| Error # 10<br>Error in type | Syntactic error in TYPE declaration. |
| Error # 11<br>'[' expected | Self-explanatory. |
| Error # 12<br>']' expected | Self-explanatory. |
| Error # 13<br>'END' expected | All procedures, functions, and blocks of statements must have an 'END'. Check for mismatched BEGIN/ENDs. |
| Error # 14<br>';' expected (possibly on line above) | Statement separator required here. |

Table A-1.   (continued)

| Message | Meaning |
|---------|---------|
| Error # 15<br>Integer expected | Self-explanatory. |
| Error # 16<br>'=' expected | Possibly a : used in a TYPE or CONST declaration. |
| Error # 17<br>'BEGIN' expected | Self-explanatory. |
| Error # 18<br>Error in declaration part | Typically an illegal backward reference to a type in a pointer declaration. |
| Error # 19<br>error in <field-list> | Syntactic error in a record declaration. |
| Error # 20<br>'.' expected | Self-explanatory. |
| Error # 21<br>'*' expected | Self-explanatory. |
| Error # 50<br>Error in constant | Syntactic error in a literal constant, also when using recursion and improperly using INP and OUT. |

**Table A-1.   (continued)**

| Message | Meaning |
|---|---|
| Error # 51<br>' =' expected | |
| | Self-explanatory. |
| Error # 52<br>'THEN' expected | |
| | Self-explanatory. |
| Error # 53<br>'UNTIL' expected | |
| | Can result from mismatched BEGIN/END sequences. |
| Error # 54<br>'DO' expected | |
| | Syntactic error. |
| Error # 55<br>'TO' or 'DOWNTO' expected in FOR statement | |
| | Self-explanatory. |
| Error # 56<br>'IF' expected | |
| | Self-explanatory. |
| Error # 57<br>'FILE' expected | |
| | Probably an error in a TYPE declaration. |
| Error # 58<br>Error in <factor> (bad expression) | |
| | Syntactic error in expression at factor level. |

**Table A-1.  (continued)**

| Message | Meaning |
|---|---|
| Error # 59<br>Error in variable | Syntactic error in expression at variable level. |
| Error # 99<br>MODEND expected | Each MODULE must end with MODEND. |
| Error # 101<br>Identifier declared twice | Name already in visible symbol table. |
| Error # 102<br>Low bound exceeds high bound | For subranges, the lower bound must be <= high bound. |
| Error # 103<br>Identifier is not of the appropriate class | A variable name used as a type, or a type used as a variable, can cause this error. |
| Error # 104<br>Undeclared identifier | The specified identifier is not in the visible symbol table. |
| Error # 105<br>Sign not allowed | Signs are not allowed on noninteger/nonreal constants. |

Table A-1.  (continued)

| Message | Meaning |
|---------|---------|
| Error # 106<br>Number expected | This error often occurs from making the compiler totally confused in an expression as it checks for numbers after all other possibilities have been exhausted. |
| Error # 107<br>Incompatible subrange types | For example, 'A'..'Z' is not compatible with 0..9. |
| Error # 108<br>File not allowed here | File comparison and assignment is not allowed. |
| Error # 109<br>Type must not be real | Self-explanatory. |
| Error # 110<br><tagfield> type must be scalar or subrange | Self-explanatory. |
| Error # 111<br>Incompatible with <tagfield> part | Selector in a CASE-variant record is not compatible with the <tagfield> type. |
| Error # 112<br>Index type must not be real | An array cannot be declared with real dimensions. |

## Table A-1.   (continued)

| Message | Meaning |
|---|---|
| **Error # 113**<br>Index type must be a scalar or a subrange | Self-explanatory. |
| **Error # 114**<br>Base type must not be real | Base type of a set can be scalar or subrange. |
| **Error # 115**<br>Base type must be a scalar or a subrange | Self-explanatory. |
| **Error # 116**<br>Error in type of standard procedure parameter | Self-explanatory. |
| **Error # 117**<br>Unsatisfied forward reference | A forwardly declared pointer was never defined. |
| **Error # 118**<br>Forward reference type identifier in variable declaration | You attempted to declare a variable as a pointer to a type that was not yet declared. |
| **Error # 119**<br>Respecified params not OK for a forward declared procedure | Self-explanatory. |

## Table A-1.  (continued)

| Message | Meaning |
|---------|---------|
| **Error # 120**<br>Function result type must be scalar, subrange or pointer | A function was declared with a string or other nonscalar type as its value.  This is not allowed. |
| **Error # 121**<br>File value parameter not allowed | Files must be passed as VAR parameters. |
| **Error # 122**<br>A forward declared function's result type cannot be respecified | Self-explanatory. |
| **Error # 123**<br>Missing result type in function declaration | Self-explanatory. |
| **Error # 125**<br>Error in type of standard procedure parameter | This is often caused by not having the parameters in the proper order for built-in procedures or by attempting to read/write pointers, enumerated types, and so on. |
| **Error # 126**<br>Number of parameters does not agree with declaration | Self-explanatory. |
| **Error # 127**<br>Illegal parameter substitution | Type of parameter does not exactly match the corresponding formal parameter. |

Table A-1.    (continued)

| Message | Meaning |
|---------|---------|
| Error # 128<br>Result type does not agree with declaration | When assigning to a function result, the types must be compatible. |
| Error # 129<br>Type conflict of operands | Self-explanatory. |
| Error # 130<br>Expression is not of set type | Self-explanatory. |
| Error # 131<br>Tests on equality allowed only | Occurs when comparing sets for other than equality. |
| Error # 133<br>File comparison not allowed | File control blocks cannot be compared because they contain multiple fields that are not available to the user. |
| Error # 134<br>Illegal type of operand(s) | The operands do not match those required for this operator. |
| Error # 135<br>Type of operand must be boolean | The operands to AND, OR, and NOT must be BOOLEAN. |

**Table A-1.** (continued)

| Message | Meaning |
| --- | --- |
| **Error # 136**<br>Set element type must be scalar or subrange | Self-explanatory. |
| **Error # 137**<br>Set element types must be compatible | Self-explanatory. |
| **Error # 138**<br>Type of variable is not array | A subscript was specified on a nonarray variable. |
| **Error # 139**<br>Index type is not compatible with the declaration | Occurs when indexing into an array with the wrong type of indexing expression. |
| **Error # 140**<br>Type of variable is not record | Attempting to access a nonrecord data structure with the dot form or the with statement. |
| **Error # 141**<br>Type of variable must be file or pointer | Occurs when an up arrow follows a variable that is not of type pointer or file. |
| **Error # 142**<br>Illegal parameter solution | Self-explanatory. |

Table A-1.   (continued)

| Message | Meaning |
|---------|---------|
| Error # 143<br>Illegal type of loop control variable | Loop control variables can be only local nonreal scalars. |
| Error # 144<br>Illegal type of expression | The expression used as a selecting expression in a CASE statement must be a nonreal scalar. |
| Error # 145<br>Type conflict | Case selector is not the same type as the selecting expression. |
| Error # 146<br>Assignment of files not allowed | Self-explanatory. |
| Error # 147<br>Label type incompatible with selecting expression | Case selector is not the same type as the selecting expression. |
| Error # 148<br>Subrange bounds must be scalar | Self-explanatory. |
| Error # 149<br>Index type must be integer | Self-explanatory. |
| Error # 150<br>Assignment to standard function is not allowed | Self-explanatory. |

Table A-1.  (continued)

| Message | Meaning |
|---|---|
| **Error # 151**<br>Assignment to formal function is not allowed | Self-explanatory. |
| **Error # 152**<br>No such field in this record | Self-explanatory. |
| **Error # 153**<br>Type error in read | Self-explanatory. |
| **Error # 154**<br>Actual parameter must be a variable | Occurs when attempting to pass an expression as a VAR parameter. |
| **Error # 155**<br>Control variable cannot be formal or nonlocal | The control variable in a FOR loop must be LOCAL. |
| **Error # 156**<br>Multidefined case label | Self-explanatory. |
| **Error # 157**<br>Too many cases in case statement | Occurs when jump table generated for case overflows its bounds. |
| **Error # 158**<br>No such variant in this record | Self-explanatory. |

Table A-1.   (continued)

| Message | Meaning |
|---|---|
| Error # 159<br>Real or string tagfields not allowed | Self-explanatory. |
| Error # 160<br>Previous declaration was not forward | |
| Error # 162<br>Parameter size must be constant | |
| Error # 163<br>Missing variant in declaration | Occurs when using NEW/DISPOSE and a variant does not exist. |
| Error # 165<br>Multidefined label | Label more than one statement with same label. |
| Error # 166<br>Multideclared label | Declare same label more than once. |
| Error # 167<br>Undeclared label | Label on statement was not declared. |
| Error # 168<br>Undefined label | A declared label was not used to label a statement. |
| Error # 169<br>Error in base set | |

**Table A-1.   (continued)**

| Message          Meaning |
|---|
| Error # 170<br>Value parameter expected |
| Error # 174<br>Pascal function or procedure expected<br><br>                    Self-explanatory. |
| Error # 183<br>External declaration not allowed at this nesting level<br><br>                    Self-explanatory. |
| Error # 201<br>Error in real number - digit expected<br><br>                    Self-explanatory. |
| Error # 202<br>String constant must not exceed source line |
| Error # 203<br>Integer constant exceeds range<br><br>                    Range on the  integer constants are<br>                    −32768..32767 |
| Error # 250<br>Too many scopes of nested identifiers<br><br>                    There is a limit of 15 nesting levels at<br>                    compile time.  This includes WITH and<br>                    procedure nesting. |
| Error # 251<br>Too many nested procedures or functions<br><br>                    There is a limit of 15 nesting levels at<br>                    execution time.  Also occurs when more<br>                    than 200 routines are in one compiled<br>                    module. |

Table A-1.   (continued)

| Message | Meaning |
|---|---|
| Error # 253<br>Procedure (or program body) too long | A procedure generated code that overflowed the internal procedure buffer. Reduce the size of the procedure and try again. The limit is 4096 bytes. |
| Error # 259<br>Expression too complicated | Your expression is too complicated (that is, too many recursive calls are needed to compile it). You should reduce the complication using temporary variable. |
| Error # 397<br>Too many FOR or WITH statements in a procedure | Only 16 FOR or WITH statements are allowed in a single procedure. |
| Error # 398<br>Implementation restriction | Normally used for arrays and sets that are too big to be manipulated or allocated. |
| Error # 407<br>Symbol Table Overflow | |
| Error # 496<br>Invalid operand to INLINE | Usually due to reference that requires address calculation at run-time. |
| Error # 497<br>Error in closing code file. | An error occurred when the ERL file was closed. Make more room on the destination disk and try again. |

Table A-1.  (continued)

| Message | Meaning |
| --- | --- |
| Error # 500<br>Non-ISO Standard feature. Not fatal. | |
| Error # 999<br>Compiler confused due to previous errors. | Make some corrections and try again.  It is also possible that while your program is syntactically correct, it can confuse the compiler if semantic errors exist. The compiler aborts early with this error number.  Look carefully at the line on which the compilation halts. |

End of Appendix A

# Appendix B
## Library Routines

The Pascal/MT+ compiler generates native machine code.  Each processor requires a library of run-time routines to support files and any other features that are not supported by the native hardware, but that are required to implement the entire Pascal language.  The following information is specific to the 8080/Z80 CP/M implementations of Pascal/MT+.

In Pascal/MT+, all I/O is performed and set variables are manipulated with library routines.  Only the run-time routines needed for a particular program are actually loaded when you link the program with LINK/MT+ and use the /S option.

Note that console I/O is assumed by the initialization routine, @INI.  This causes the input/output routines to be loaded even when you are not using them.  If you want to avoid this, you can write a replacement @INI routine and link it before linking the run-time library to resolve the @INI reference.

The table below lists the names of the run-time library routines and their purposes.  This table clarifies what these routines do, so that when you disassemble a program you have some information about what is happening in your program.  They are not here so that you can call these routines from your program. Digital Research does not guarantee parameter list compatibility between releases.

**Table B-1.  Run-time Library Routines**

| Routine | Purpose |
|---------|---------|
| @CHN | Program chaining routine |
| @MUL | Long Integer multiply |
| @EQD | String comparison routine for = |
| @NED | String comparison routine for <> |
| @GTD | String comparison routine for > |
| @LTD | String comparison routine for < |
| @GED | String comparison routine for >= |
| @LED | String comparison routine for <= |
| @EQS | Set equality |
| @NES | Set inequality |
| @GES | Set superset |
| @LES | Set subset |

**Table B-1.  (continued)**

| Routine | Purpose |
|---------|---------|
| @HLT | End of program halt routine; return to operating system |
| @SAD | Set union |
| @SSB | Set difference |
| @SML | Set intersection |
| @SIN | Set membership |
| @BST | Build singleton set |
| @BSR | Build subrange set |
| @EQA | Array comparison routine for = |
| @NEA | Array comparison routine for <> |
| @GTA | Array comparison routine for > |
| @LTA | Array comparison routine for < |
| @GEA | Array comparison routine for >= |
| @LEA | Array comparison routine for <= |
| @XJP | Table case jump routine |
| @LBA | Load concat string buffer address |
| @ISB | Initialize string buffer |
| @CNC | Concatenate a string to the buffer |
| @CCH | Concatenate a character to the buffer |
| @RCH | Read a character from a file |
| @CRL | Write a newline (CR) to a file |
| @CWT | Read until EOLN is TRUE on a file |
| @WIN | Write an integer to a file |
| @RST | Read a string from a file |
| TSTBIT | Test for a bit on |
| SETBIT | Turn a bit on |
| CLRBIT | Turn a bit off |
| SHL | Shift a word left |
| SHR | Shift a word right |
| @SFB | Set global FIB address |
| @DWD | Set default width and decimal places |
| @SIA | Reset input vector |
| @SOA | Reset output vector |
| @DIO | Set I/O vectors to default addresses |
| @INI | Run-time initialization |
| @STR | String store |
| @WCH | Write a string to a file |
| @DVL | 32-bit DIV software routine |

## Table B-1.   (continued)

| Routine | Purpose |
|---------|---------|
| @MDL | 32-bit MOD software routine |
| MOVELE | Block move left end to left end |
| MOVERI | Block move right end to right end |
| @CHW | Write a character to a file |
| @EQR | Real comparison for  = |
| @NER | Real comparison for  <> |
| @GTR | Real comparison for  > |
| @LTR | Real comparison for  < |
| @GER | Real comparison for  >= |
| @LER | Real compasison for  <= |
| @RRL | Read a real from a file |
| @WRL | Write a real to a file |
| @RAD | Real add |
| @RSB | Real subtract |
| @RML | Real multiply |
| @RDV | Real divide |
| @RNG | Real negate |
| @RAB | Real absolute value |
| @RDL | Read a long integer from a file |
| @RTL | Write a long integer to a file |
| SQRT | Real square root |
| TRUNC | Pascal built-in truncate function |
| ROUND | Pascal built-in round function |
| CHAIN | Pascal interface for @CHN |
| OPEN | File handling routine |
| BLOCKR | File handling routine |
| BLOCKW | File handling routine |
| CREATE | File handling routine |
| CLOSE | File handling routine |
| CLOSED | File handling routine |
| GNB | File handling routine |
| WNB | File handling routine |
| PAGE | File handling routine |
| EOLN | File handling routine |
| EOF | File handling routine |
| RESET | File handling routine |
| REWRIT | File handling routine |
| GET | File handling routine |

Table B-1.  (continued)

| Routine | Purpose |
|---------|---------|
| PUT | File handling routine |
| ASSIGN | File handling routine |
| PURGE | File handling routine |
| IORESU | File handling routine |
| COPY | File handling routine |
| INSERT | File handling routine |
| DELETE | File handling routine |
| POS | Run-time support for strings |
| @WNC | Write next character to a file |
| @RNC | Read next character from a file |
| @RIN | Read integer from a file |
| @RNB | Read n bytes from a file |
| @WNB | Write n bytes to a file |
| @BDOS86 | Call operating system directly |
| @NEW | Allocate memory for NEW procedure |
| @DSP | Deallocate memory for DISPOSE procedure |
| MEMAVA | MEMAVAIL function |
| MAXAVA | MAXAVAIL function |

End of Appendix B

# Appendix C
# Sample Disassembly

   This appendix contains the Pascal/MT+ program, PPRIME, which is
compiled with /X and /P options and then disassembled, producing the
following output.

   References to program locations are followed by a single
apostrophe (1000'), and references to data locations are followed by
a quotation mark (0000").

   The operand of instructions that reference external variables
points to the previous reference and the final reference contains
absolute 0000. The list of external chains follows the disassembly
of the program.

**Note:** the object code generated in this example does not
necessarily indicate the level of optimization present in the
current release of the Pascal/MT+ compiler. To determine the level
of optimization, compile programs yourself and use the disassembler
to examine the output.

Pascal/MT+     Release 5.5  Copyright (c) 1982 Digital Research
Page # 1
Compilation of: PPRIME


```
 Stmt  Nest    Source Statement
   1    0
   2    0      PROGRAM PPRIME;
   3    0      (* USES SIEVE OF ERATOSTHENES *)
   4    0      CONST
   5    1        SIZE=8190;
   6    1      VAR
   7    1        FLAGS:                    ARRAY[0..SIZE] OF BOOLEAN;
   8    1        I,PRIME,K,ITER:           INTEGER;
   9    1        COUNT:                    INTEGER;
  10    1
  11    1      BEGIN
  12    1        COUNT := 0;
  13    1        writeln('10 iterations');
  14    1        FOR ITER := 1 TO 10 DO
  15    1          BEGIN
  16    2            COUNT:=0;
  17    2
  18    2            FILLCHAR(FLAGS,SIZEOF(FLAGS),CHR(TRUE));
  19    2
  20    2            FOR I:=0 TO SIZE DO
  21    2              IF FLAGS[I] THEN
  22    2                BEGIN
  23    3                  PRIME:=I+I+3;
  24    3                  K:=I+PRIME;
  25    3                  WHILE K<=SIZE DO
  26    3                    BEGIN
  27    4                      FLAGS[K]:=FALSE;
  28    4                      K:=K+PRIME;
  29    4                    END;
  30    3                  COUNT:=COUNT + 1;
  31    3                END
  32    3        END;
  33    1        writeln(count,' primes');
  34    1      END.
  34    0      ----------------------------
  34    0      Normal End of Input Reached
```


**Listing C-1.   Compilation of PPRIME**

Output from disassembler:

Pascal/MT+ Release 5.5 Copyright (c) 1981 by MT MicroSYSTEMS  Page #  1
Disassembly of: PPRIME


Stmt  Nest    Source Statement / Symbolic Object Code

        FLAGS    EQU      0000
        ITER     EQU      2000
        K        EQU      2002
        PRIME    EQU      2004
        I        EQU      2006
        COUNT    EQU      2008
   1      0     PROGRAM PPRIME;

0000             DB       00,00,00,00,00,00,00,00
0008             DB       00,00,00,00,00,00,00,00
0010             JMP      0000

   2      0     (* USES SIEVE OF ERATOSTHENES *)
   3      0     CONST
   4      1       SIZE=8190;
   5      1     VAR
   6      1       FLAGS:            ARRAY[0..SIZE] OF BOOLEAN;
   7      1       I,PRIME,K,ITER:   INTEGER;
   8      1       COUNT:            INTEGER;
   9      1
  10      1     BEGIN

0013             LHLD     0006
0016             SPHL
0017             CALL     0000

  11      1       COUNT := 0;

001A             LXI      H,0000
001D             SHLD     2008"

  12      1       writeln('10 iterations');

0020             LXI      H,0000
0023             PUSH     H
0024             CALL     0000
0027             CALL     0038'
002A             DB       0D,31,30,20,69,74,65,72
0032             DB       61,74,69,6F,6E,73
0038             CALL     0000
003B             CALL     0000
003E             CALL     0000

  13      1       FOR ITER := 1 TO 10 DO

0041             LXI      H,0001
0044             PUSH     H
0045             LXI      H,000A
0048             PUSH     H
0049             POP      D
004A             POP      H
004B             DCX      H
004C             SHLD     2000"


Listing C-2.  Disassembly of PPRIME

Pascal/MT+  Release 5.5  Copyright (c) 1981 by MT MicroSYSTEMS  Page #   2
Disassembly of: PPRIME

| Stmt | Nest | Source Statement / Symbolic Object Code |
|------|------|------------------------------------------|
| 004F | | INX   H |
| 0050 | | PUSH  H |
| 0051 | | PUSH  D |
| 0052 | | CALL  0000 |
| 0055 | | SHLD  200A" |
| 0058 | | LHLD  2000" |
| 005B | | INX   H |
| 005C | | SHLD  2000" |
| 005F | | LHLD  200A" |
| 0062 | | DCX   H |
| 0063 | | SHLD  200A" |
| 0066 | | MOV   A,H |
| 0067 | | ORA   L |
| 0068 | | JZ    011D' |
| | | |
| 14 | 1 | BEGIN |
| 15 | 2 | COUNT:=0; |
| | | |
| 006B | | LXI   H,0000 |
| 006E | | SHLD  2008" |
| | | |
| 16 | 2 | |
| 17 | 2 | FILLCHAR(FLAGS,SIZEOF(FLAGS),CHR(TRUE)); |
| | | |
| 0071 | | LXI   H,0000" |
| 0074 | | PUSH  H |
| 0075 | | LXI   H,1FFF |
| 0078 | | PUSH  H |
| 0079 | | LXI   H,0001 |
| 007C | | PUSH  H |
| 007D | | CALL  0000 |
| | | |
| 18 | 2 | |
| 19 | 2 | FOR I:=0 TO SIZE DO |
| | | |
| 0080 | | LXI   H,0000 |
| 0083 | | PUSH  H |
| 0084 | | LXI   H,1FFE |
| 0087 | | PUSH  H |
| 0088 | | POP   D |
| 0089 | | POP   H |
| 008A | | DCX   H |
| 008B | | SHLD  2006" |
| 008E | | INX   H |
| 008F | | PUSH  H |
| 0090 | | PUSH  D |
| 0091 | | CALL  0053' |
| 0094 | | SHLD  200C" |
| 0097 | | LHLD  2006" |
| 009A | | INX   H |
| 009B | | SHLD  2006" |
| 009E | | LHLD  200C" |
| 00A1 | | DCX   H |

**Listing C-2.   (continued)**

| Stmt | Nest | Source Statement / Symbolic Object Code |
|------|------|------|
| 00A2 | | SHLD  200C" |
| 00A5 | | MOV   A,H |
| 00A6 | | ORA   L |
| 00A7 | | JZ    011A' |
| 20   | 2 | IF FLAGS[I] THEN |
| 00AA | | LXI   H,0000" |
| 00AD | | XCHG |
| 00AE | | LHLD  2006" |
| 00B1 | | DAD   D |
| 00B2 | | MOV   A,M |
| 00B3 | | RAR |
| 00B4 | | JNC   0117' |
| 21   | 2 | BEGIN |
| 22   | 3 | PRIME:=I+I+3; |
| 00B7 | | LHLD  2006" |
| 00BA | | XCHG |
| 00BB | | LHLD  2006" |
| 00BE | | DAD   D |
| 00BF | | INX   H |
| 00C0 | | INX   H |
| 00C1 | | INX   H |
| 00C2 | | SHLD  2004" |
| 23   | 3 | K:=I+PRIME; |
| 00C5 | | LHLD  2006" |
| 00C8 | | XCHG |
| 00C9 | | LHLD  2004" |
| 00CC | | DAD   D |
| 00CD | | SHLD  2002" |
| 24   | 3 | WRITELN(PRIME); |
| 00D0 | | LHLD  2004" |
| 00D3 | | PUSH  H |
| 00D4 | | LXI   H,0021' |
| 00D7 | | PUSH  H |
| 00D8 | | CALL  0025' |
| 00DB | | CALL  0039' |
| 00DE | | CALL  0000 |
| 00E1 | | CALL  003F' |
| 25   | 3 | WHILE K<=SIZE DO |
| 00E4 | | LHLD  2002" |
| 00E7 | | PUSH  H |
| 00E8 | | LXI   H,1FFE |
| 00EB | | PUSH  H |
| 00EC | | CALL  0000 |

Listing C-2.   (continued)

Pascal/MT+  Release 5.5  Copyright (c) 1981 by MT MicroSYSTEMS  Page #  4
Disassembly of: PPRIME

```
Stmt  Nest     Source Statement / Symbolic Object Code

00EF             POP     PSW
00F0             JNC     0110'

  26    3                 BEGIN
  27    4                   FLAGS[K]:=FALSE;

00F3             LXI     H,0000"
00F6             XCHG
00F7             LHLD    2002"
00FA             DAD     D
00FB             PUSH    H
00FC             LXI     H,0000
00FF             XCHG
0100             POP     H
0101             MOV     M,E

  28    4                   K:=K+PRIME;

0102             LHLD    2002"
0105             XCHG
0106             LHLD    2004"
0109             DAD     D
010A             SHLD    2002"

  29    4                 END;

010D             JMP     00E4'

  30    3                 COUNT:=COUNT + 1;

0110             LHLD    2008"
0113             INX     H
0114             SHLD    2008"

  31    3               END
  32    3             END;

0117             JMP     0097'
011A             JMP     0058'

  33    1       writeln(count,' primes');

011D             LHLD    2008"
0120             PUSH    H
0121             LXI     H,00D5'
0124             PUSH    H
0125             CALL    00D9'
0128             CALL    00DC'

012B             CALL    00DF'
012E             CALL    0139'
0131             DB      07,20,70,72,69,6D,65,73
0139             CALL    0129'
013C             CALL    003C'
```

**Listing C-2.   (continued)**

Stmt  Nest    Source Statement / Symbolic Object Code

013F          CALL   00E2'

  34    1    END.

0142          CALL   0000

External reference chain @WIN    --> 012C
External reference chain @CRL    --> 0140
External reference chain @LEI    --> 00ED
External reference chain @FIN    --> 0092
External reference chain @SFB    --> 0126
External reference chain @DWD    --> 013A
External reference chain @INI    --> 0018
External reference chain @WRS    --> 013D
External reference chain @HLT    --> 0143
External reference chain OUTPUT  --> 0122
External reference chain FILLCH  --> 007E

## Listing C-2.   (continued)

## End of Appendix C

Listing C-2. (continued)

End of Appendix C

Listing C-2. (continued)

# Appendix D
## Sample Debugging Session

This appendix supplies a sample debugging session that uses the
source file DEBUG.PAS, shown below.

```
Stmt   Nest    Source Statement

  1     0
  2     0      (* EXAMPLE TO ILLUSTRATE DEBUGGER *)
  3     0
  4     0      PROGRAM DEBUG;
  5     0      VAR
  6     1        HEXARR : STRING[16];
  7     1        CH : CHAR;
  8     1        I : INTEGER;
  9     1
 10     1      (* DUMMY PROC TO ALLOW SETTING BREAKPOINT *)
 11     1
 12     1      PROCEDURE BREAK;
 13     1      BEGIN
 14     2      END;
 15     1
 16     1      (* FUNCTION TO CONVERT FROM INTEGER TO HEX CHARACTER *)
 17     1
 18     1      FUNCTION CONVERT( I : INTEGER) : CHAR;
 19     1      BEGIN
 20     2        CONVERT := HEXARR[1];
 21     2      END;
 22     1
 23     1      BEGIN
 24     1        HEXARR:= '0123456789ABCDEF';
 25     1
 26     1        REPEAT
 27     2          BEGIN
 28     3          WRITELN('ENTER INTEGER TO CONVERT: '); READ(I);
 29     3          CH:=CONVERT(I);
 30     3          BREAK; (* BREAK ON RETURN FROM CONVERT *)
 31     3          WRITELN('HEX DIGIT IS: ',CH);
 32     3          END
 33     3        UNTIL FALSE;
 34     1
 35     1      END.
```

**Listing D-1.  DEBUG.PAS Source File**

In the following sample session, you interactively debug a simple program.  Your input is shown in boldface print; the column on the right provides an explanation of each step.

| | |
|---|---|
| `A>MTPLUS B:DEBUG $D` | Compile the program with the Debug option. |
| Pascal MT+      Release 5.5<br>(c) 1981 MT MicroSYSTEMS, Inc. | System displays banner. |
| `A>LINKMT B:DEBUG=DEBUGGER,B:DEBUG,PASLIB/S` | Link the object file with the debugger. |
| Link/MT+ Release 5.5 | System displays banner.<br>**Note:** the linker might display @WRL as an undefined symbol.  If your program does not use real numbers, you can ignore it. |
| `A>B:DEBUG` | Run program. |
| Pascal/MT+ Symbolic Debugger, Release 5.5 | System displays banner. |
| Symbol table filename (<return> only for none)? **B:DEBUG.SYP** | Load the symbols. |
| Use BEgin or TRace to start a program<br>`+>SB BREAK`<br>`+>BE` | Set breakpoint, then start the program. |
| ENTER INTEGER TO CONVERT:<br>**5**<br>Breakpoint reached | Enter data. |
| `+>DV I`<br>Address: 0272 Contains: 5 | Examine I.  It is correct. |
| `+>DV CH`<br>Address: 0270 Contains: 0 == 30 | Examine CH.  It is wrong.  Why?  Because convert is not returning the correct value.  Reviewing the source shows that a 1 was typed when an I was intended on line 16.  Before recompiling check for other errors. |
| `+>DC HEXARR+5`<br>Address: 0263 Contains: 4 == 34 | Examine HEXARR[5].  It is not 5. |
| `+>DX HEXARR`<br>Address: 025E Contains:<br>025E= 10 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45   .0123456789ABCDE<br>026E= 46 00 30 00 05 00 00 00 00 00 00 00 00 00 00 00   F.0............ | Examine all of HEXARR.  All the digits are off by 1.  Note that HEXARR is a string and therefore HEXARR[0] is the length field.  The code for convert does not allow for this. |
| `+>^C` | Now that you have determined the problem, exit DEBUGGER, and go back to the source and fix it. |

# Appendix E
# Interprocessor Portability

This appendix describes the features of Pascal/MT+ that are not portable to versions for other microprocessors and operating systems. A program without the following features should compile with another Pascal/MT+ compiler with little or no changes to the source code.

This does not mean that all of the features listed below are not implemented on any other target processors. It only indicates that they are hardware-dependent, and if implemented, are implemented differently in different versions of the compiler. If you use any of these hardware-dependent features, isolate them so that they are easy to modify when you port the program.

While every effort is made to support compatibility, Digital Research does not guarantee complete portability to all implementations. The guidelines that follow are subject to change without notice. There is no additional information concerning portability to other Pascal/MT+ compilers.

If you want to write portable programs, avoid the following features:

- Avoid INLINE.

- Avoid I/O ports (hardware-dependent).

- Avoid redirected I/O (hardware-dependent).

- Avoid device names such as CON:, RDR:, etc.

- Avoid scattering calls to IORESULT throughout the program. Isolate the calls. IORESULT values depend on the operating system.

- Avoid ABSOLUTE addressing (hardware-dependent).

- Avoid INTERRUPT procedures (hardware-dependent).

- Avoid the use of variant records that circumvent type checking.

- Avoid chaining. Chaining is implementation-dependent.

- Avoid having overlays call other overlays. This is not possible on other operating systems.

- Avoid dependence upon EOF for non-TEXT files because it is implementation dependent. Some operating systems keep track of how much information is in the file to the exact byte, while others only keep track to the sector/block level, and the last sector/block contains garbage information.

- Avoid using temporary files.

- Avoid BLOCKREAD/BLOCKWRITE because these might not be implemented on all operating systems. Use SEEKREAD/SEEKWRITE instead.

### End of Appendix E

# Appendix F
# Mini-assembler Mnemonics

The following table lists the valid 8080 mini-assembler mnemonics for the INLINE construct of the Pascal/MT+ compiler. Spaces and commas are ignored when mnemonics appear in an INLINE construct. For example, "MOV A,M/ is the same as "MOVAM/.

### Table F-1. 8080 Mini-assembler Mnemonics

| Mnemonic | Value | Mnemonic | Value |
|----------|-------|----------|-------|
| NOP | 000H | DADH | 029H |
| LXIB | 001H | LHLD | 02AH |
| STAXB | 002H | DCXH | 02BH |
| INXB | 003H | INRL | 02CH |
| INRB | 004H | DCRL | 02DH |
| DCRB | 005H | MVIL | 02EH |
| MVIB | 006H | CMA | 02FH |
| RLC | 007H | SIM | 030H |
|  |  | LXISP | 031H |
| DADB | 009H | STA | 032H |
| LDAXB | 00AH | INXSP | 033H |
| DCXB | 00BH | INRM | 034H |
| INRC | 00CH | DCRM | 035H |
| DCRC | 00DH | MVIM | 036H |
| MVIC | 00EH | STC | 037H |
| RRC | 00FH |  |  |
|  |  | DADSP | 039 |
| LXID | 011H | LDA | 03AH |
| STAXD | 012H | DCXSP | 03BH |
| INXD | 013H | INRA | 03CH |
| INRD | 014H | DCRA | 03DH |
| DCRD | 015H | MVIA | 03EH |
| MVID | 016H | CMC | 03FH |
| RAL | 017H | MOVBB | 040H |
|  |  | MOVBC | 041H |
| DADD | 019H | MOVBD | 042H |
| LDAXD | 01AH | MOVBE | 043H |
| DCXD | 01BH | MOVBH | 044H |
| INRE | 01CH | MOVBL | 045H |
| DCRE | 01DH | MOVBM | 046H |
| MVIE | 01EH | MOVBA | 047H |
| RAR | 01FH | MOVCB | 048H |
| RIM | 020H | MOVCC | 049H |
| LXIH | 021H | MOVCD | 04AH |
| SHLD | 022H | MOVCE | 04BH |
| INXH | 023H | MOVCH | 04CH |
| INRH | 024H | MOVCL | 04DH |
| DCRH | 025H | MOVCM | 04EH |

**Table F-1.  (continued)**

| Mnemonic | Value | Mnemonic | Value |
|----------|-------|----------|-------|
| MVIH     | 026H  | MOVCA    | 04FH  |
| DAA      | 027H  | MOVDB    | 050H  |
| MOVDC    | 051H  | ADDH     | 084H  |
| MOVDD    | 052H  | ADDL     | 085H  |
| MOVDE    | 053H  | ADDM     | 086H  |
| MOVDH    | 054H  | ADDA     | 087H  |
| MOVDL    | 055H  | ADCB     | 088H  |
| MOVDM    | 056H  | ADCC     | 089H  |
| MOVDA    | 057H  | ADCD     | 08AH  |
| MOVEB    | 058H  | ADCE     | 08BH  |
| MOVEC    | 059H  | ADCH     | 08CH  |
| MOVED    | 05AH  | ADCL     | 08DH  |
| MOVEE    | 05BH  | ADCM     | 08EH  |
| MOVEH    | 05CH  | ADCA     | 08FH  |
| MOVEL    | 05DH  | SUBB     | 090H  |
| MOVEM    | 05EH  | SUBC     | 091H  |
| MOVEA    | 05FH  | SUBD     | 092H  |
| MOVHB    | 060H  | SUBE     | 093H  |
| MOVHC    | 061H  | SUBH     | 094H  |
| MOVHD    | 062H  | SUBL     | 095H  |
| MOVHE    | 063H  | SUBM     | 096H  |
| MOVHH    | 064H  | SUBA     | 097H  |
| MOVHL    | 065H  | SBBB     | 098H  |
| MOVHM    | 066H  | SBBC     | 099H  |
| MOVHA    | 067H  | SBBD     | 09AH  |
| MOVLB    | 068H  | SBBE     | 09BH  |
| MOVLC    | 069H  | SBBH     | 09CH  |
| MOVLD    | 06AH  | SBBL     | 09DH  |
| MOVLE    | 06BH  | SBBM     | 09EH  |
| MOVLH    | 06CH  | SBBA     | 09FH  |
| MOVLL    | 06DH  | ANAB     | 0A0H  |
| MOVLM    | 06EH  | ANAC     | 0A1H  |
| MOVLA    | 06FH  | ANAD     | 0A2H  |
| MOVMB    | 070H  | ANAE     | 0A3H  |
| MOVMC    | 071H  | ANAH     | 0A4H  |
| MOVMD    | 072H  | ANAL     | 0A5H  |
| MOVME    | 073H  | ANAM     | 0A6H  |
| MOVMH    | 074H  | ANAA·    | 0A7H  |
| MOVML    | 075H  | XRAB     | 0A8H  |
| HLT      | 076H  | XRAC     | 0A9H  |
| MOVMA    | 077H  | XRAD     | 0AAH  |
| MOVAB    | 078H  | XRAE     | 0ABH  |
| MOVAC    | 079H  | XRAH     | 0ACH  |
| MOVAD    | 07AH  | XRAL     | 0ADH  |
| MOVAE    | 07BH  | XRAM     | 0AEH  |
| MOVAH    | 07CH  | XRAA     | 0AFH  |
| MOVAL    | 07DH  | ORAB     | 0B0H  |
| MOVAM    | 07EH  | ORAC     | 0B1H  |
| MOVAA    | 07FH  | ORAD     | 0B2H  |

Table F-1.  (continued)

| Mnemonic | Value | Mnemonic | Value |
|----------|-------|----------|-------|
| ADDB | 080H | ORAE | 0B3H |
| ADDC | 081H | ORAH | 0B4H |
| ADDD | 082H | ORAL | 0B5H |
| ADDE | 083H | ORAM | 0B6H |
| ORAA | 0B7H | IN | 0DBH |
| CMPB | 0B8H | CC | 0DCH |
| CMPC | 0B9H | | |
| CMPD | 0BAH | SBI | 0DEH |
| CMPE | 0BBH | RST3 | 0DFH |
| CMPH | 0BCH | RPO | 0E0H |
| CMPL | 0BDH | POPH | 0E1H |
| CMPM | 0BEH | JPO | 0E2H |
| CMPA | 0BFH | XTHL | 0E3H |
| RNZ | 0C0H | CPO | 0E4H |
| POPB | 0C1H | PUSHH | 0E5H |
| JNZ | 0C2H | ANI | 0E6H |
| JMP | 0C3H | RST4 | 0E7H |
| CNZ | 0C4H | RPE | 0E8H |
| PUSHB | 0C5H | PCHL | 0E9H |
| ADI | 0C6H | JPE | 0EAH |
| RST0 | 0C7H | XCHG | 0EBH |
| RZ | 0C8H | CPE | 0ECH |
| RET | 0C9H | | |
| JZ | 0CAH | XRI | 0EEH |
| | | RST5 | 0EFH |
| CZ | 0CCH | RP | 0F0H |
| CALL | 0CDH | POPPS | 0F1H |
| ACI | 0CEH | JP | 0F2H |
| RST1 | 0CFH | DI | 0F3H |
| RNC | 0D0H | CP | 0F4H |
| POPD | 0D1H | PUSHP | 0F5H |
| JNC | 0D2H | ORI | 0F6H |
| OUT | 0D3H | RST6 | 0F7H |
| CNC | 0D4H | RM | 0F8H |
| PUSHD | 0D5H | SPHL | 0F9H |
| SUI | 0D6H | JM | 0FAH |
| RST2 | 0D7H | EI | 0FBH |
| RC | 0D8H | CM | 0FCH |
| JC | 0DAH | CPI | 0FEH |
| | | RST7 | 0FFH |

End of Appendix F

# Appendix G
# Comparison of I/O Methods

This appendix illustrates four different ways to implement a single file procedure named TRANSFER.  Listing G-1 shows the main statement body that calls the transfer routine in each of four separate programs.

```
BEGIN
    WRITE('Source? ');
    READLN(NAME);
    ASSIGN(A,NAME);
    RESET(A);
    IF IORESULT = 255 THEN
        BEGIN
            WRITELN('Cannot open ',NAME);
            EXIT
        END;

    WRITE('Destination? ');
    READLN(NAME);
    ASSIGN(B,NAME);
    REWRITE(B);
    IF IORESULT = 255 THEN
        BEGIN
            WRITELN('Cannot open ',NAME);
            EXIT
        END;

    TRANSFER(A,B)
END.
```

**Listing G-1.  Main Program Body for File Transfer Programs**

Listing G-2 shows a transfer program using the BLOCKREAD and BLOCKWRITE procedures.  This program uses untyped files, and a large 2K buffer to transfer data.  Note that the program only works for files whose size is an even multiple of 2K bytes.  Thus, if the size of the source file is 9K, the last 1K is not written because the variable RESULT is nonzero after the call to BLOCKREAD on line 25.  Using a 128-byte buffer guarantees that all the data is transferred.

The program shown in Listing G-3 uses the GNB and WNB routines for byte-level access to the file.

The program shown in Listing G-4 performs the file transfer
using the SEEKREAD and SEEKWRITE procedures. Notice that IORESULT
returns a 1 to indicate end-of-file if the last portion of data from
the source file does not fill the sector, just as in BLOCK I/O. In
this case, the 2K bytes that are the window variable for file
variable A do not fill the sector. However, the end portion of code
that does not fill up the 2K buffer is never written to the
destination file.

Listing G-5 uses GET and PUT to transfer files. This method is
slower than the buffered methods.

Table G-1 shows the code, data size, and execution speed for
each of the file transfer procedures when run on a 4MHz Z80
processor with no wait states, and a single-density, single-sided,
8-inch floppy disk. The sizes are in decimal bytes, the speed is in
seconds, and the size of the file is 8K bytes.

**Note:** these numbers are not identical for all releases of the
compiler. Your version might not produce the same size and speed.
However, the relative size and speed differences should be roughly
the same.

**Table G-1.   Size and Speed of Transfer Procedures**

| Transfer Method | BLOCK I/O | GNB/WNB | SEEK I/O | GET/PUT |
|-----------------|-----------|---------|----------|---------|
| Compiled Code   | 520       | 519     | 530      | 477     |
| Compiled Data   | 2532      | 2534    | 4584     | 482     |
| Total Code      | 7317      | 7161    | 9243     | 6764    |
| Total Data      | 3576      | 3577    | 5697     | 1494    |
| Total Size      | 10893     | 10738   | 14940    | 8258    |
| Speed           | 7.8       | 18.4    | 8.6      | 35.1    |

```
Stmt   Nest     Source Statement

   1    0        PROGRAM FILE_TRANSFER;
   2    0
   3    0        (*----------------------------------------------------*)
   4    0        (* Transfer A to B using BLOCKREAD and BLOCKWRITE *)
   5    0        (*----------------------------------------------------*)
   6    0
   7    0        CONST
   8    1          BUFSZ = 2047;
   9    1        TYPE
  10    1          PAOC = ARRAY[1..BUFSZ] OF CHAR;
  11    1          FYLE = FILE;
  12    1
  13    1        VAR
  14    1          A,B : FYLE;
  15    1          NAME : STRING;
  16    1          BUF : PAOC;
  17    1
  18    1        PROCEDURE TRANSFER(VAR SRC: FYLE; VAR DEST : FYLE);
  19    1        VAR
  20    2          RESULT,I : INTEGER;
  21    2          QUIT  : BOOLEAN;
  22    2        BEGIN
  23    2          I := 0;
  24    2          REPEAT
  25    3            BLOCKREAD(SRC,BUF,RESULT,SIZEOF(BUF),I);
  26    3            IF RESULT = 0 THEN
  27    3              BEGIN
  28    4                BLOCKWRITE(DEST,BUF,RESULT,SIZEOF(BUF),I);
  29    4                I := I + SIZEOF(BUF) DIV 128
  30    4              END
  31    4            ELSE
  32    3                QUIT := TRUE;
  33    3          UNTIL QUIT;
  34    2          CLOSE(DEST,RESULT);
  35    2          IF RESULT = 255 THEN
  36    2              WRITELN('Error closing destination file')
  37    2        END;
  38    1                    (* MAIN PROGRAM IN LISTING G-1 *)
```

**Listing G-2.  File Transfer with BLOCKREAD and BLOCKWRITE**

```
Stmt   Nest      Source Statement
  1     0        PROGRAM FILE_TRANSFER;
  2     0
  3     0        (*------------------------------------------------*)
  4     0        (* Transfer file A to file B using GNB and WNB    *)
  5     0        (*------------------------------------------------*)
  6     0
  7     0        CONST
  8     1          BUFSZ = 2047;
  9     1        TYPE
 10     1          PAOC = ARRAY[1..BUFSZ] OF CHAR;
 11     1          TFILE = FILE OF PAOC;
 12     1          CHFILE = FILE OF CHAR;
 13     1        VAR
 14     1          A : TFILE;
 15     1          B : CHFILE;
 16     1          NAME : STRING;
 17     1
 18     1        PROCEDURE TRANSFER(VAR SRC: TFILE; VAR DEST : CHFILE);
 19     1        VAR
 20     2          CH : CHAR;
 21     2          RESULT : INTEGER;
 22     2          ABORT  : BOOLEAN;
 23     2        BEGIN
 24     2          ABORT := FALSE;
 25     2          WHILE (NOT EOF(SRC)) AND (NOT ABORT) DO
 26     2            BEGIN
 27     3              CH := GNB(SRC);
 28     3              IF WNB(DEST,CH) THEN
 29     3                BEGIN
 30     4                  WRITELN('Error writing character');
 31     4                  ABORT := TRUE;
 32     4                END;
 33     3            END;
 34     2          CLOSE(DEST,RESULT);
 35     2          IF RESULT = 255 THEN
 36     2            WRITELN('Error closing ')
 37     2        END;
 38     1                  (* MAIN PROGRAM IN LISTING G-1 *)
```

**Listing G-3.  File Transfer with GNB and WNB**

```
Stmt  Nest     Source Statement

  1    0        PROGRAM FILE_TRANSFER;
  2    0
  3    0        (*-----------------------------------------------*)
  4    0        (* Transfer A to B using SEEKREAD and SEEKWRITE*)
  5    0        (*-----------------------------------------------*)
  6    0
  7    0        CONST
  8    1          BUFSZ = 2047;
  9    1
 10    1        TYPE
 11    1          PAOC = ARRAY[0..BUFSZ] OF CHAR;
 12    1          TFILE = FILE OF PAOC;
 13    1          CHFILE = FILE OF PAOC;
 14    1        VAR
 15    1          A : TFILE;
 16    1          B : TFILE;
 17    1          NAME : STRING;
 18    1        PROCEDURE TRANSFER(VAR SRC: TFILE; VAR DEST : TFILE);
 19    1        VAR
 20    2          CH : CHAR;
 21    2          RESULT2,RESULT,I : INTEGER;
 22    2          ABORT   : BOOLEAN;
 23    2        BEGIN
 24    2          CH := 'A';
 25    2          RESULT := 0;
 26    2          I := 0;
 27    2          WHILE RESULT <> 1 DO
 28    2            BEGIN
 29    3              SEEKREAD(SRC,I);
 30    3              RESULT := IORESULT;
 31    3              IF RESULT = 0 THEN
 32    3                BEGIN
 33    4                  DEST^ := SRC^;
 34    4                  SEEKWRITE(DEST,I);
 35    4                END;
 36    3              I := I + 1;
 37    3            END;
 38    2
 39    2          CLOSE(DEST,RESULT);
 40    2          IF RESULT = 255 THEN
 41    2            WRITELN('Error closing destination file')
 42    2        END;
 43    1            (* MAIN PROGRAM IN LISTING G-1 *)
```

**Listing G-4.  File Transfer with SEEKREAD and SEEKWRITE**

```
Stmt  Nest    Source Statement

  1    0       PROGRAM FILE_TRANSFER;
  2    0
  3    0       (*------------------------------------------------*)
  4    0       (* Transfer file A to file B using GET and PUT *)
  5    0       (*------------------------------------------------*)
  6    0
  7    0       TYPE
  8    1          CHFILE = FILE OF CHAR;
  9    1       VAR
 10    1          A,B : CHFILE;
 11    1          NAME : STRING;
 12    1
 13    1       PROCEDURE TRANSFER(VAR SRC: CHFILE; VAR DEST : CHFILE);
 14    1       VAR
 15    2          RESULT : INTEGER;
 16    2       BEGIN
 17    2          WHILE NOT EOF(SRC) DO
 18    2             BEGIN
 19    3                DEST^ := SRC^;
 20    3                PUT(DEST);
 21    3                GET(SRC);
 22    3             END;
 23    2
 24    2          CLOSE(DEST,RESULT);
 25    2          IF RESULT = 255 THEN
 26    2             WRITELN('Error closing destination file')
 27    2       END;
 28    1                    (* MAIN PROGRAM IN LISTING G-1 *)
```

**Listing G-5.  File Transfer with GET and PUT**

End of Appendix G

# Index

# ⅏ DIGITAL RESEARCH

# SOFTWARE PERFORMANCE REPORT

The Registration Manager
Digital Research (UK) Limited,
Oxford House, Oxford Street,
Newbury, Berkshire, RG13 1JB

### Report Type

☐ Problem/Possible Error
☐ Suggested Enhancement
☐ Document Suggestion
☐ Other

### Rate Problem Impact

☐ 3  Shuts Down System
☐ 2  Impairs System
    Performance
☐ 1  Causes Inconvenience
☐ 0  Needs Suggested
    Enhancement

| |
|---|
| Product Name and Version |
| Date |
| User Name |
| Company Name |
| Address |
| Country |
| Phone                  Ext. |

| |
|---|
| Computer Type |
| No. of Disk Drives |

Problem Description: (Please describe the problem and how it can be reproduced, your diagnosis and your cure, if known. Be concise and attach a listing, if available).