Pascal/MT+™
Language
Programmer's Guide
for the
CP/M® Family of Operating Systems

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

COPYRIGHT

DISCLAIMER

TRADEMARKS

The Pascal/MT+ Language Programmer's Guide for the CP/M Family of Operating Systems was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

****
* First Edition: March 1983 *
****

Foreword

The Pascal/MT+™ language is a full implementation of standard Pascal as set forth in the International Standards Organization (ISO) standard DPS/7185. 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.

Table of Contents

Table of Contents
(continued)

Appendixes

Tables, Figures, and Listings

Tables

Figures

Listings

Tables, Figures, and Listings (continued)

Listings

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.

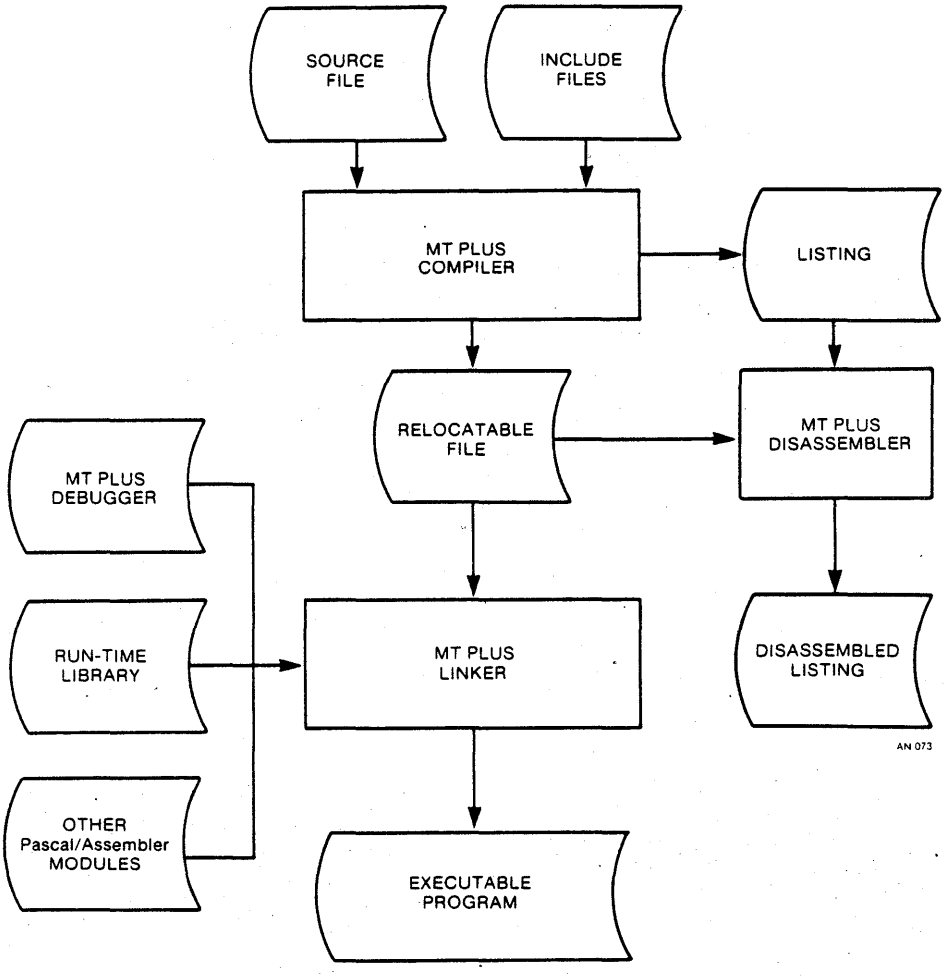


Figure 1-1. Software Development Under Pascal/MT+

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.

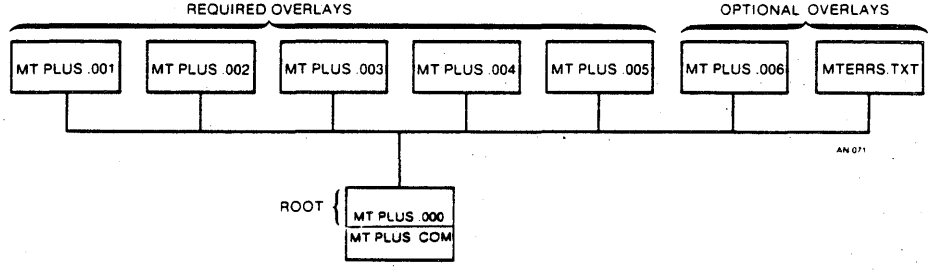


Figure 2-1. Pascal/MT+ Compiler Organization

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.

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 commmand 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.

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).

End of Section 2

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.

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 *)
2
(* 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

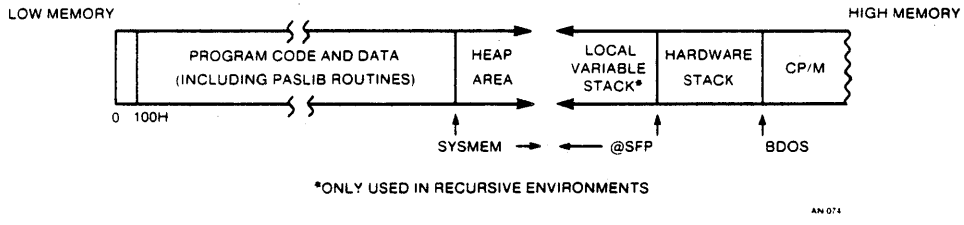
End of Section 3

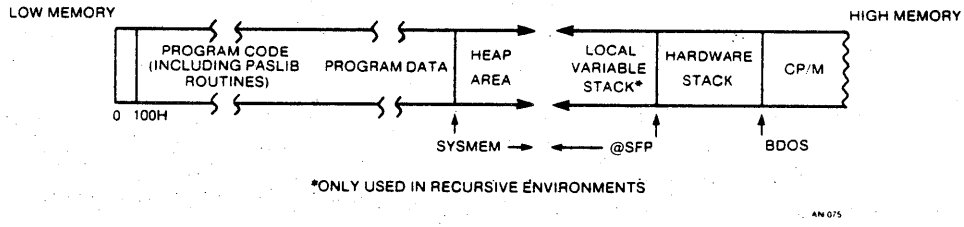
Section 4
Run-time Interface

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.





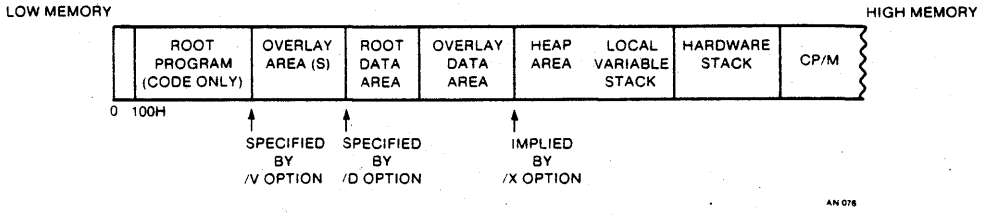


Figure 4-3. Pascal/MT+ Memory Map at Run-time: Program with Overlays

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 'RETI' 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 errrors.

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.

End of Section 4

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.

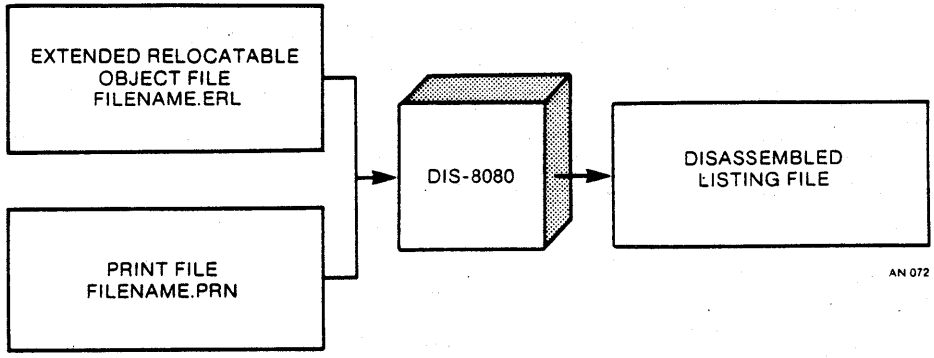


Figure 5-1. DIS8080 Operation

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

Table A-1. (continued)

Message	Meaning
Error # 15 Integer expected	Self-explanatory.
Error # 16 '=' expected	Possibly a : used in a TYPE or CONST declaration.
Error # 17 'BEGIN' expected	Self-explanatory.
Error # 18 Error in declaration part	Typically an illegal backward reference to a type in a pointer declaration.
Error # 19 error in <field-list>	Syntactic error in a record declaration.
Error # 20 '.' expected	Self-explanatory.
Error # 21 '*' expected	Self-explanatory.
Error # 50 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 ' =' expected	Self-explanatory.
Error # 52 'THEN' expected	Self-explanatory.
Error # 53 'UNTIL' expected	Can result from mismatched BEGIN/END sequences.
Error # 54 'DO' expected	Syntactic error.
Error # 55 'TO' or 'DOWNTO' expected in FOR statement	Self-explanatory.
Error # 56 'IF' expected	Self-explanatory.
Error # 57 'FILE' expected	Probably an error in a TYPE declaration.
Error # 58 Error in <factor> (bad expression)	Syntactic error in expression at factor level.

Table A-1. (continued)

Message	Meaning
Error # 59 Error in variable	Syntactic error in expression at variable level.
Error # 99 MODEND expected	Each MODULE must end with MODEND.
Error # 101 Identifier declared twice	Name already in visible symbol table.
Error # 102 Low bound exceeds high bound	For subranges, the lower bound must be <= high bound.
Error # 103 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 Undeclared identifier	The specified identifier is not in the visible symbol table.
Error # 105 Sign not allowed	Signs are not allowed on noninteger/nonreal constants.

Table A-1. (continued)

Message	Meaning
Error # 106 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 Incompatible subrange types	For example, 'A'..'Z' is not compatible with 0..9.
Error # 108 File not allowed here	File comparison and assignment is not allowed.
Error # 109 Type must not be real	Self-explanatory.
Error # 110 <tagfield> type must be scalar or subrange	Self-explanatory.
Error # 111 Incompatible with <tagfield> part	Selector in a CASE-variant record is not compatible with the <tagfield> type.
Error # 112 Index type must not be real	An array cannot be declared with real dimensions.

Table A-1. (continued)

Message	Meaning
Error # 113 Index type must be a scalar or a subrange	Self-explanatory.
Error # 114 Base type must not be real	Base type of a set can be scalar or subrange.
Error # 115 Base type must be a scalar or a subrange	Self-explanatory.
Error # 116 Error in type of standard procedure parameter	Self-explanatory.
Error # 117 Unsatisfied forward reference	A forwardly declared pointer was never defined.
Error # 118 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 Respecified params not OK for a forward declared procedure	Self-explanatory.

Table A-1. (continued)

Message	Meaning
Error # 120 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 File value parameter not allowed	Files must be passed as VAR parameters.
Error # 122 A forward declared function's result type cannot be respecified	Self-explanatory.
Error # 123 Missing result type in function declaration	Self-explanatory.
Error # 125 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 Number of parameters does not agree with declaration	Self-explanatory.
Error # 127 Illegal parameter substitution	Type of parameter does not exactly match the corresponding formal parameter.

Table A-1. (continued)

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

Table A-1. (continued)

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

Table A-1. (continued)

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

Table A-1. (continued)

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

Table A-1. (continued)

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

Table A-1. (continued)

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

Table A-1. (continued)

Message	Meaning
Error # 253 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 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 Too many FOR or WITH statements in a procedure	Only 16 FOR or WITH statements are allowed in a single procedure.
Error # 398 Implementation restriction	Normally used for arrays and sets that are too big to be manipulated or allocated.
Error # 407 Symbol Table Overflow	
Error # 496 Invalid operand to INLINE	Usually due to reference that requires address calculation at run-time.
Error # 497 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 Non-ISO Standard feature. Not fatal.	
Error # 999 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)

```
Pascal/MT+  Release 5.5  Copyright (c) 1981 by MT MicroSYSTEMS  Page #   3
Disassembly of: PPRIME


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)

```
Pascal/MT+  Release 5.5  Copyright (c) 1981 by MT MicroSYSTEMS  Page #   5
Disassembly of: PPRIME


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

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
(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.
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.SYM
```

Load the symbols.

```
Use BEgin or TRace to start a program
+>SB BREAK
+>BE

ENTER INTEGER TO CONVERT:
5
Breakpoint reached
```

Set breakpoint, then start the program.

Enter data.

```
+>DV I
Address: 0272 Contains: 5
```

Examine I. It is correct.

```
+>DV CH
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
Address: 0263 Contains: 4 == 34
```

Examine HEXARR[5]. It is not 5.

```
+>DX HEXARR
Address: 025E Contains:
025E= 10 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45    .0123456789ABCDE
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

Reader Comment Form

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

Date _____ Manual Title _____ Edition _____

1. What sections of this manual are especially helpful?

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

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

COMMENTS AND SUGGESTIONS BECOME THE PROPERTY OF DIGITAL RESEARCH.

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS / PERMIT NO. 182 / PACIFIC GROVE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL RESEARCH™

P.O. Box 579
Pacific Grove, California
93950

Attn: Publications Production

IMPORTANT NOTICE

THANK YOU FOR BUYING OUR PRODUCT. YOU MAY HAVE A READ.ME FILE ON YOUR DISK. PLEASE LOCATE THE FILE AND READ IT.

Post Office Box 579 ■ 160 Central Avenue ■ Pacific Grove, California 93950 ■ (408) 649-3896 ■ TWX 910 360 5001