Pascal / MT+™

Language

Programmer's Guide

for the CP/M-86®
Family of Operating Systems

Pascal/MT+™
Language
Programmer's Guide
for the CP/M-86®
Family of Operating Systems

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

The Pascal/MT+ Language Programmer's Guide for the CP/M-86 Family of Operating Systems was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

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

Foreword

The Pascal/MT+™ language is a full implementation of standard Pascal as set forth in the International Standards Organization (ISO) standard DPS/7185. 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 8086- or 8088-based microcomputer with at least 118K 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)

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

- ASMT-86™, a relocating assembler
- DIS-86™, a disassembler
- LIB/MT+86™, a software library-building utility
- a dynamic debugger

The Pascal/MT+ system runs under any of the CP/M-86® family of operating systems on an 8086-based computer. The compiler and linker need at least 118K 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.

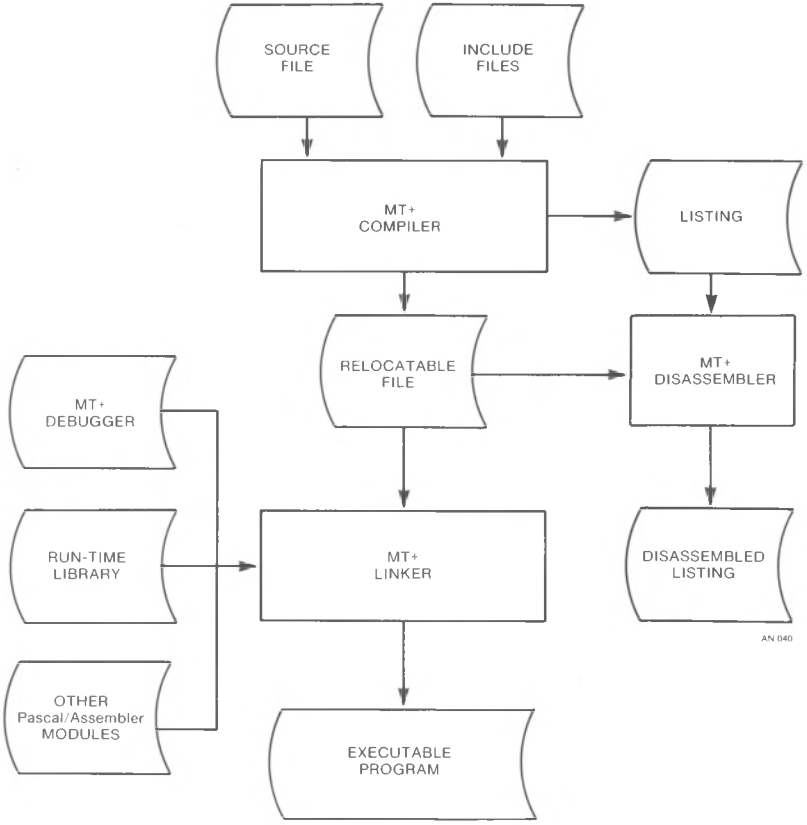


Figure 1-1. Software Development under Pascal/MT+

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 LIB/MT+86
CMD	Command file; directly executable under CP/M-86
DOC	Document file; contains printable text in ASCII form
ERR	Error message file output by compiler
I86	Intel® 8086 file; contains assembly-language source file for ASMT-86
KMD	Linker input command file
LIB	Library file; contains subroutines
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
R86	Relocatable 8086 object file; contains relocatable object code emitted by the compiler
SRC	Pascal source file; contains source code in ASCII form (the compiler also accepts PAS as a source filetype)
SYP	Symbol file used by debugger
SYM	Symbol file used by SID-86™
TXT	Text file; contains text of messages output by compiler, etc.
nnn	Hexadecimal n; used for numbering overlays

Table 1-2. Pascal/MT+ Distribution Disks

Disk 1	
File	Content or Use
DIS86.CMD	Disassembler
MT+86.CMD	Pascal/MT+ Compiler
MT+86.000	Compiler Overlay
MT+86.001	Compiler Overlay
MT+86.002	Compiler Overlay
MT+86.003	Compiler Overlay
MT+86.004	Compiler Overlay
MT+86.005	Overlay used with Debugger
MTERRS.TXT	Compiler Error Message Text File
REL31.DOC	Release Notes for Version 3.1

Disk 2	
File	Content or Use
LINKMT.CMD	Pascal/MT+ Linker
LINKMT.001	Linker Overlay
LINKMT.002	Linker Overlay
LIBMT.CMD	LIB/MT+86 Librarian Utility
STRIP.CMD	Utility program used with LINKMT to eliminate unused entry points
NM.CMD	Utility program used with LIBMT to determine module names
SZ.CMD	Utility program used with LIBMT to determine module sizes
DEBUGGER.R86	Debugging module that can be linked to a program
TRANSCEND.R86	Transcendental arithmetic module
FPREALS.R86	Floating-point arithmetic module
PASLIB.R86	Pascal/MT+ Run-time System module

Table 1-2. (continued)

Disk 3	
File	Content or Use
ASMT86.CMD	Pascal/MT+ Assembler
ASMT.001	Assembler Overlay
ASMT.002	Assembler Overlay
ASMT.003	Assembler Overlay
ASMT.004	Assembler Overlay
AMERS.TXT	Assembler Error Message Text File
CONCAT.CMD	Concatenates TEXT files
MT2INT.CMD	Conversion utility for changing R86 files into Intel relocatable object file format
87REALS.R86	8087 processor arithmetic module
BCDREALS.R86	BCD arithmetic module
FULLHEAP.R86	Heap management module
RANDOMIO.R86	Random I/O file processing module
REALIO.R86	Real arithmetic I/O module
INI3.I86	CP/M-86 initialization routine
8087.I86	8087 math routines
87XOP.I86	8087 miscellaneous routines
87TRS.I86	8087 truncate, round, and square root routines
HLT.I86	CP/M-86 halt routine
OVLMGR3.I86	Overlay Manager
DBUGHELP.TXT	Help file for debugger module
FIBDEF.LIB	File Information Block definition
IOALONE.DOC	Document file explaining stand-alone I/O
87REALS.BLD	Build file for 8087 arithmetic module
ECHO.PAS	Sample Program
IOMOD.PAS	Source file for I/O routine
CPMGET.PAS	Source file for GET routine
CPMINI.PAS	Source file for initialization routine
MOD1.PAS	Sample Program
MOD2.PAS	Sample Program
DEMOPROG.PAS	Sample Program

1.2 Installing Pascal/MT+

The first thing you should do when you receive your Pascal/MT+ system is make a copy of all 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 one 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 MT+86.005 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.

To make a compiler disk and a linker disk, perform the following steps:

1) Install CP/M-86 and the PIP utility on two blank disks. Label one disk as the compiler, and the other, the linker.

2) Put a text editor on the compiler disk.

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

   - MT+86.CMD
   - MT+86.000 through MT+86.005
   - MTERRS.TXT

4) Copy the following files to the linker disk:

   - LINKMT.CMD
   - LINKMT.001
   - LINKMT.002
   - all the R86 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-86 system with two disk drives and that you are familiar with CP/M-86.

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>MT+86 B:TEST1
```

If you examine your directory, you will see a file named TEST1.R86 that contains the relocatable object code emitted 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 will now contain a file named TEST1.CMD that is directly executable under CP/M-86.

5) To run the program, enter the command:

```
B>TEST1
```

Although the test program shown above 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.

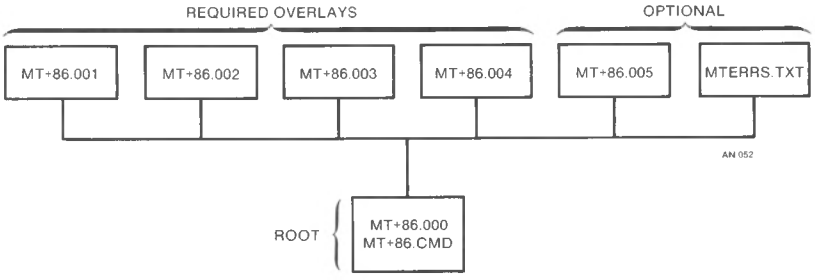


Figure 2-1. Pascal/MT+ Compiler Organization

2.2 Invoking the Compiler

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

```
MT+86 <filespec> {<options>}
```

where <filespec> is the source file to be compiled, and the <options> are 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 then assumes a SRC filetype, then 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 R86 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 module's beginning and the size of the procedure (in decimal) follow the name.

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

```
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 the following messages:

```
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 or PHASE1
002      PH2INIT
003      BLK
004      PH2TERM
005      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 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
B	Use BCD rather than binary for real numbers.	Binary reals
C	Continue on error; default is to pause and let user interact on each error, one at a time.	Compiler stops and asks on each error
D	Generate debugger information in the object code and write 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: d= @,A..O	MTERRS.TXT on default disk
Pd	Put the PRN (listing file) on disk d: d= X,P,@,A..O	No PRN file
Q	Quiet, suppress any unnecessary unnecessary console messages.	Compiler outputs all messages
Rd	Put the R86 file on disk d: d=@,A..O	R86 file on default disk
Td	Put the token file PASTEMP.TOK on disk d: d= @,A..O	PASTEMP.TOK on default disk
V	Print 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	Generate an extended R86 file including disassembler records.	R86 file cannot be disassembled
@	Make the @ character equivalent to the ^ character.	@ not equivalent to ^

The following is an example command line:

```
A>MT+86 A:TESTPROG $RBPX
```

This command line tells the compiler to read the source from drive A, write the R86 file to drive B, and display the PRN file on the console.

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	No effect in 8086 version; included for compatibility only	
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	No effect in 8086 version; included for compatibility only	
R +/-	Controls range checking code	R-
S +/-	Controls recursive/static variables	S+

Table 2-2. (continued)

Option	Function	Default
T +/-	Controls strict type checking	T-
W +/-	Generates warning messages	W-
X +/-	Controls exception checking code	X-
Z	Sets the stack pointer	

The following examples show proper source code compiler options:

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

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 first for a SRC, then a PAS, and finally a blank filetype. 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

Table 2-3. (continued)

Group	Routines Removed
13	SEEKREAD, SEEKWRITE
14	unused on the 8086
15	unused on the 8086

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.

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)

In the 8086 implementation, the compiler ignores the S option because it always generates reentrant and recursive programs, unlike the 8080/Z80® version. Global variables within programs or modules are always allocated statically.

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 weak type checking only. 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 suppresses generation of the CP/M-86 type initialization. You should enter the option as $Z+ only once before the PROGRAM line in the main program, and not on the individual modules.

In a CP/M-86 environment, the compiler initializes the hardware stack by loading the SS register with the value in data segment 15H and the SP register with the value in data segment 12H. If you use the Z option, you must provide code to preload the SS and SP registers before executing the program. This code is normally in the form of an assembly-language routine.

2.3 Using the Linker

LINK/MT+86™ is the linkage editor that reads relocatable object modules with filetype R86 and generates an executable command file with filetype CMD. The linker can also generate overlay files.

You invoke LINK/MT+86 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 R86 filetype for the <main module> and all <modules> unless you specify a KMD filetype, see the discussion about the /F option for information about KMD files. LINK/MT+86 can link a maximum of 40 files at one time.

The following examples show valid LINK/MT+86 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+86 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+86 supports 16 options as summarized in Table 2-4.

Table 2-4. Linker Options

Option	Function
S	Search preceding name as a library, extracting only the required routines.
L	List modules as they are being linked.
M	List all entry points in tabular form.
E	List entry points beginning with $, ? or @ in addition to other entry points requiring /M or /W to operate.
P:nnnn	Relocate object code to nnnnH.
D:nnnn	Specify maximum data area of nnnnH bytes.
R:nnnn	Specify maximum code area of nnnnH bytes.
X:nnnn	Specify Extra segment of nnnnH paragraphs for root programs).
Z:nnnn	Specify Stack segment of nnnnH paragraphs.
Y'<filespec>'	Write linker messages to <filespec>.

Table 2-4. (continued)

Option	Function
W	Write a SID-86 compatible SYM file (written to the same disk as the CMD file).
F	Take preceding filename as a KMD file containing input filenames.
C	Continuation flag in KMD file (use on all but last line).
Vm:nnnn	Overlay area starting address.
X:nnnn	Overlay static variable space (when linking overlays).
O:n	Number the overlay and use the previous filename as the root program symbol table. By default, n ranges from 1 to 50 but can be extended (1 to 256) by altering the overlay manager.

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 R86 files supplied with the Pascal/MT+ system are not searchable. You cannot search user-created modules unless they are processed by LIB/MT+86, as described in Section 5.3.

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.

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 will not display module code and data locations if used alone.

Program Origin (/P)

Use the P option to link overlays by controlling the location of the object code within the Code segment. The linker supports relocation of object code so that it can overlay. The P option does not tell the linker to leave space at the beginning of the CMD file.

The syntax of the P option is

```
/P:nnnn
```

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

Maximum Code Size (/R) and Maximum Data Size (/D)

The R option specifies the maximum code area size. The D option specifies the maximum data area size. The R and D options have a single hexadecimal number argument following a colon (/R:nnnn, /D:nnnn). The argument can range from 0 to 0FFFFH, specifying the segment size in bytes.

Generate SYM File (/W)

The W option tells the linker to generate a SID-86 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.

Linker Input Command File (/F)

Normally in a CP/M-86 environment, you must use the SUBMIT facility for typing repetitive sequences such as linking multiple files together. LINK/MT+86 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 KMD 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 KMD file to link the files CALC, TRANCEND, FPREALS, and PASLIB into a CMD file. The command to link the files is

```
A>LINKMT CALC/F/L
```

The file CALC.KMD contains

```
A:CALC,D:TRANCEND,F:PREALS,B:PASLIB/S
```

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

Extra Segment and Stack Segment Size Switches (/X),(/Z)

On the 8086 under CP/M-86, you can specify the size of the Extra segment that the heap uses exclusively and the size of the Stack segment that the return addresses, parameter passing, and local variables use.

The X option controls the size of the Extra segment. Its default value is 0. Therefore, MEMAVAIL returns 0 and no heap is available if you do not specify X.

The Z option default value of Z:200 allocates 8K bytes for the stack. Note that you specify the size of the segments in 16-byte paragraphs, not bytes. The number is in hexadecimal, so x:800 asks for 800H paragraphs, which actually means 8000H bytes, or 32K.

Directing Linker output to a file (/Y)

LINK/MT+86 lets you direct linker output to a file with the Y linker option. The default action directs output to the console. You must follow the Y option with the filespec or device name in single apostrophes. For example,

```
/Y'MYFILE'
```

tells the linker to generate the file MYFILE.MAP, and

```
/Y'LST:'
```

tells the linker to route the output to the system-list device. When errors occur, a message goes to both the output file and the console.

Overlay Options

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

2.3.2 Required Relocatable Files

The distribution disks contain certain R86 files that you must link into any program that loads, stores, assigns, inputs, outputs, or declares any real number. If you have any of these routines as undefined references, link the appropriate relocatable file to resolve them. The following are R86 files:

- TRANCEND: Support for SIN, COS, ARCTAN, SQRT, LN, EXP, SQR. Use only with FPREALS.
- RANDOMIO: SEEKREAD and SEEKWRITE are resolved here.
- DEBUGGER: @NLN, @EXT, @ENT generated when debugger option is requested. If @XOP and @WRL are undefined, see Section 5.4.
- PASLIB: Comparisons, I/O, arithmetic support, etc.

The following files contain the real-number routines:

- BCDREALS: BCD real numbers, @XOP, @RRL, @WRL
- FPREALS: Binary real numbers @XOP,@RRL,@WRL (searchable)
- 87REALS: Hardware real numbers using the Intel 8087 coprocessor

2.3.3 Linker Error Messages

Table 2-5 shows the three linker error messages.

Table 2-5. Linker Error Messages

Meaning	Message
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.

2.4 Using Other Linkers

LINK/MT+86 links Pascal/MT+ main programs, Pascal/MT+ modules, and assembly-language modules created by ASMT-86. The MT2INT program, supplied on distribution disk #3, converts R86 files into Intel format 8086 OBJ files. You can transport these files to other CP/M-86 systems and link them with LINK-86™.

You invoke MT2INT with a command line of the form:

```
MT2INT <filename>
```

where <filename> is the name of an R86 file.

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.

For example,

```
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.

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 CMD 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. All Pascal/MT+ modules are recursive. 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. When you link the overlay, the linker determines the amount of data space required for static variables.

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 only use one overlay area. You can devise more extensive schemes using multiple overlay areas. The overlay number determines the area where LINKMT86 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-loading routine 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 OVLMGR3.I86 on distribution disk #3.

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 main CMD file, use the @OVS routine to specify the drive. Declare the routine as follows:

```
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...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 change the overlay manager to a reloading version. The source for the overlay manager is in the file OVLMGR3.I86 which is on distribution disk #3. 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 are 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 will not be 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 only need 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 the Extra segment. nnnn is the hexadecimal number of 16-byte paragraphs. See Section 2.3.1 for more information.

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 nonoverlayed 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/R:pppp
```

This command line only shows the three required options. 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 size of the data area. The value is the sum of the root's data size and the sizes of the overlay's data. Leave room during development, so that the overlay data areas can grow.

- The R option specifies the total code size, which includes the overlay areas. Use the sum of root program's code, plus 80H, and the size of each separate overlay area, plus 80H for each area.

- Remember to use the X option if your program uses the heap. The default size is 0.

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 has 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 only shows the three options that are required for linking overlays.

- 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 if the overlay has any 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 the message:

  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 the message:

  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 #3. You should compile and link them to get a feel for using overlays. The files are the following:

- DEMOPROG.PAS
- MOD1.PAS
- MOD2.PAS

```
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. DEMOPROG.PAS

```
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.PAS

```
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.PAS

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

```
A>LINKMT DEMOPROG,PASLIB/S/D:1000/V1:4000/R:5000
```

This creates the files DEMOPROG.CMD and DEMOPROG.SYM with the data size set to 1000 (this is arbitrary). The overlay areas, 1 to 16, are at 4000 (again arbitrary), and the total code size is estimated to be 5000.

To link overlay 1, enter this command:

```
A>LINKMT DEMOPROG=DEMOPROG/O:1,MOD1,PASLIB/S/P:4000/L
```

This creates the overlay file DEMOPROG.001. The /O:1 option tells the linker to read DEMOPROG.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 DEMOPROG.CMD.

To link overlay 2, enter this command:

```
A>LINKMT DEMOPROG=DEMOPROG/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 they 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 it 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. You must use the /R and /D linker options, see Section 2.3, to reserve enough space in the first program in the chain for all programs in the chain. The /R value should be the size of the largest program plus 80H, and the /D value should be the size of the largest data requirement plus 80H.

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.

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.

No special facilities are needed to maintain the heap across the chain as are necessary in 8-bit versions of Pascal/MT+. Unlike the 8-bit versions, files cannot remain open across a chain. If you want to leave something open, you must use overlays, not chaining.

Listings 3-6a and 3-6b list 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 [$40:$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.CMD');
  RESET(CHAINFIL);
  IF IORESULT = 255 THEN
     BEGIN
       WRITELN('UNABLE TO OPEN CHAIN2.CMD');
       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 [$40:$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, with assembly-language routines, and with the operating system. It also explains how to write stand-alone programs that run without an operating system.

4.1 Run-time Environment

Figure 4-1 shows the memory layout for a Pascal/MT+ program. The heap grows towards high memory from the low end of the Extra segment. The local-variable stack grows towards low memory from the high end of the Stack segment.

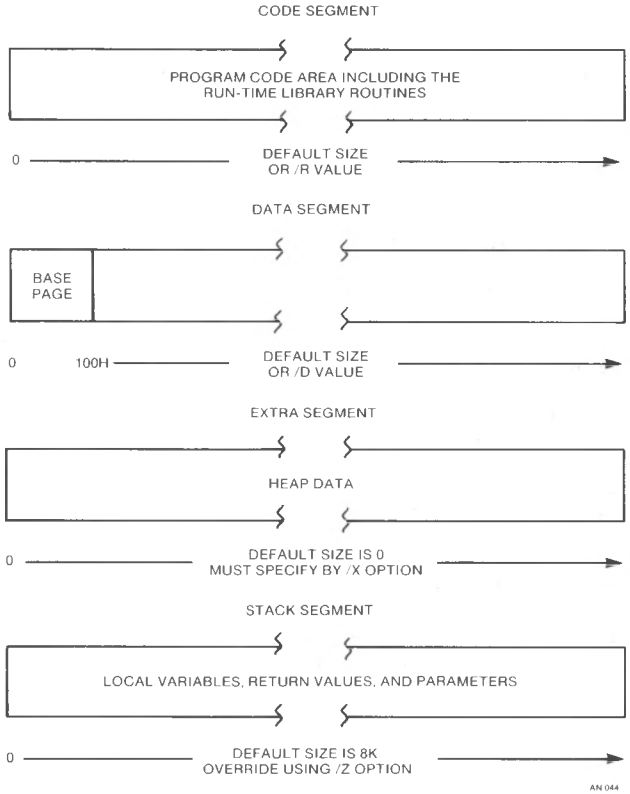


Figure 4-1. Memory Layout

4.1.1 Stack

The hardware and local-variable stacks are separate in 8-bit implementations of Pascal/MT+. In the 8086/8088 implementations, they are the same. If your program fails due to insufficient stack area, you can enlarge the stack with the /Z linker option.

Note: if you are using an interrupt-driven system, you often need to enlarge the stack.

4.1.2 Program Structure

The Pascal/MT+ compiler generates program modules containing 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 sixteen bytes of header code for overlay information. In nonoverlayed programs, these are NOPs.

Under CP/M-86, the linker bases code for loading the stack pointer and segment on the contents of the DS-relative locations 15H and 12H. 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

The ASMT-86 assembler 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.

The Pascal/MT+ compiler ignores the underscore character in names. For example, A_B is the same as AB. The Intel standard ASM-86 language treats an underscore as a significant character. Therefore, do not use underscores in labels in assembly-language modules if the names resolve to entry points in a Pascal program.

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+ option.

The following example shows how an assembly-language module references a variable that is declared in a Pascal/MT+ module.

```
        NAME    DEMO
        ASSUME CS:CODE, DS:DATA
; ASSEMBLY LANGUAGE PROGRAM FRAGMENT
DATA    SEGMENT PUBLIC

        EXTRN   PQR:WORD
DATA    ENDS

CODE    SEGMENT PUBLIC

        .
        .

        MOV     AX,PQR    ;GET contents OF PASCAL VARIABLE
        .
        .
        .
CODE    ENDS

        END

.............................


(* PASCAL PROGRAM FRAGMENT *)

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

4.2.2 Data Allocation

In the global data area, variables are allocated in the order you declare them. The exception is variables appearing 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 contiguously 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;
P     : ^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
8	P offset LSB
9	P offset MSB
10	P segment LSB
11	P segment 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.

Figure 4-2 shows the storage for the set A..Z. In this figure, 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.

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 4-2. Storage for the Set A..Z

Table 4-1 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
LONGINT	4 8-bit-bytes	$2^{32}-1..2^{-32}$
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	$10^{-307}..10^{307}$
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 or a Pascal 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.

Address operands and pointers use two words of stack space. They are stored as offset word on top of segment word, just as in a data area. 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 in a Pascal routine 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 how a typical parameter list appears on the stack 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    offset  ADDRESS OF ACTUAL STRING LSB
                  +7    offset  ADDRESS OF ACTUAL STRING MSB
                  +8    segment ADDRESS OF ACTUAL STRING LSB
                  +9    segment ADDRESS OF ACTUAL STRING MSB
                  +10   J (LSB)
                  +11   J (MSB)
                  +12   I (same as J)
                  +13   I (same as J)
```

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.

Nonreal function values return in registers. Single-byte and single-word values return in the AX register. If a pointer or LONG_INT requires a second word, the high order/segment value returns in the BX register.

Real values return on the stack. They are placed below the return address before the function returns. Therefore, they remain on the top of the stack when the calling program reenters after the return.

Assembly-language functions return only simple types, such as enumerations, INTEGER, REAL, BOOLEAN, LONGINT, pointers, and CHAR, but not arrays, STRINGS, or records.

4.2.4 Assembly-language Interface Example

Listings 4-1 and 4-2 illustrate the interface between a Pascal program and some assembly-language routines.

The Pascal 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 bytes in the specified address.

```
PROGRAM PEEK_POKE;

TYPE
      BYTEPTR = ^BYTE;

      (* THIS IS VERY 8086 SPECIFIC AND IS NOT PORTABLE! *)
      (* BUT ON THE OTHER HAND IT IS EXTREMELY VALUABLE! *)
      POINTERKLUDGE = RECORD
                        CASE BOOLEAN OF
                          TRUE : (P : BYTEPTR);
                          FALSE: (OFFSET : INTEGER;
                                  SEGMENT: INTEGER)
                        END;
VAR
      ADDRESS : INTEGER;
      CHOICE : INTEGER;
      BBB : BYTE;
      PPP : POINTERKLUDGE;

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

BEGIN
  REPEAT
    WRITE('Address? (input as segment< space> offset) ');
    READLN(PPP.SEGMENT,PPP.OFFSET);
    WRITE('1) Peek  OR  2) Poke ');
    READLN(CHOICE);
    IF CHOICE = 1 THEN
      WRITELN(ADDRESS,' contains ',PEEK(PPP.P))
    ELSE
      IF CHOICE = 2 THEN
        BEGIN
          WRITE('Enter byte of data: ');
          READLN(BBB);
          POKE(BBB,PPP.P)
        END
  UNTIL FALSE
END.
```

Listing 4-1. Pascal PEEK_POKE Program

```
        NAME    PEEK_POKE_MODULE
        ASSUME CS:CODE,DS:DATA

DATA    SEGMENT PUBLIC
DATA    ENDS

CODE    SEGMENT PUBLIC

        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    PROC NEAR
        POP BX  ;RETURN ADDRESS INTO BX
        POP DI  ; GET OFFSET INTO DI
        POP ES  ; GET SEGMENT INTO ES
        MOV AL,ES: BYTE PTR [DI] ;GO GET THE BYTE
        XOR AH,AH ; MAKE HI ORDER AX = 0
        JMP BX  ;AND EXIT LEAVING FUNCTION VALUE IN AX
PEEK    ENDP

;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    PROC NEAR
        POP BX   ;GET RETURN ADDRESS INTO BX
        POP DI   ;GET OFFSET
        POP ES   ;GET SEGMENT
        POP AX   ;GET BYTE TO STUFF
        MOV ES:BYTE PTR [DI],AL ;STUFF BYTE AWAY
        JMP BX   ;AND RETURN
POKE    ENDP

CODE    ENDS

        END
```

Listing 4-2. Assembly-language PEEK and POKE Routines

4.3 Pascal/MT+ Interface Features

Pascal/MT+ provides several features that let you control your program's run-time 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 @BDOS86 routine. You declare it in a Pascal program as follows:

```
EXTERNAL FUNCTION @BDOS86(FUNC:INTEGER; PARM:PTR):INTEGER;
```

The first parameter is the BDOS function number. Refer to your specific operating system's manual for the list of functions. The second parameter is a generic pointer (PTR). You can use the ADDR function to generate the value.

To stay compatible with the 8080 version of Pascal/MT+, @BDOS86 translates a single call with Function 26 into a call to set DMA segment and a call to set DMA offset.

The following example shows KEYPRESSED, a function that uses the @BDOS86 function. KEYPRESSED returns TRUE if a key is pressed, and FALSE if not.

```
FUNCTION KEYPRESSED : BOOLEAN;

BEGIN
  KEYPRESSED := (@BDOS86(11,ADDR(KEYPRESSED)) <> 0)
END;
```

The second operand is of type PTR, which is any user-declared pointer type, usually the result of the ADDR function.

Listings 4-3 and 4-4 illustrate calls to BDOS Functions 6 and 23 respectively.

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

PROGRAM BDOS6;
TYPE
  PTR = ^INTEGER;
VAR
  INTEG_TO_PTR : RECORD          (* THE BDOS CALL REQUIRES A POINTER PARM. *)
      CASE BOOLEAN OF             (* THIS RECORD ALLOWS US TO PASS AN *)
         TRUE : (LO : INTEGER;    (* INTEGER AS A POINTER TYPE *)
                 HI : INTEGER);
         FALSE:(POINTR : PTR); (* THIS POINTER OCCUPIES THE SAME *)
      END;                       (* MEMORY AS THE TWO INTEGERS ABOVE. *)

  CH : CHAR;

  I : INTEGER;

  EXTERNAL FUNCTION @BDOS86(FUNC:INTEGER; PARM:PTR):INTEGER;

BEGIN (* ECHO ANY INPUT CHARACTER TO THE CONSOLE UNTIL A : IS READ *)
  INTEG_TO_PTR.HI:=0;
REPEAT
  INTEG_TO_PTR.LO:=$FF;
  REPEAT
    CH:=CHR(@BDOS86(6,INTEG_TO_PTR.POINTR)); (* READ CHARACTER *)
  UNTIL CH <> CHR(0);
  IF CH <> ':' THEN
    BEGIN
    INTEG_TO_PTR.LO:=ORD(CH); (*CONVERT CH TO INTEGER, PASS AS POINTER *)
    I:=@BDOS86(6,INTEG_TO_PTR.POINTR);   (* WRITE CHARACTER *)
    END;
  UNTIL CH= ':';
END.
```

Listing 4-3. Calling BDOS Function 6

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

PROGRAM BDOS23;
TYPE
  PTR = ^INTEGER;
  FCBLK = PACKED ARRAY [0..36] OF CHAR;
VAR
  F1,
  F2 : FCBLK;
  I : INTEGER;
  OLDNAME,NEWNAME : STRING;

  (* PARSE IS A PROCEDURE TO CONVERT STRINGS INTO INTERNAL *)
  (* CP/M FILE NAME FORMAT *)
  EXTERNAL PROCEDURE @PARSE(VAR F:FCBLK; S:STRING);

  EXTERNAL FUNCTION @BDOS86(FUNC:INTEGER; PARM:PTR):INTEGER;

BEGIN
  WRITE('ENTER OLD FILE NAME: '); (* GET THE OLD FILE NAME *)
  READLN(OLDNAME);
  @PARSE(F1,OLDNAME);

  WRITE('ENTER NEW FILE NAME: '); (* GET THE NEW FILE NAME *)
  READLN(NEWNAME);
  @PARSE(F2,NEWNAME);

  MOVE(F2,F1[16],12);  (* CREATE THE FCB REQUIRED BY BDOS CALL 23 *)

  (* CALL THE RENAME FUNCTION. PASS A POINTER TO THE FCB *)
  (* CONTAINING THE OLD AND NEW FILE NAMES *)
  IF @BDOS86(23,ADDR(F1)) = 255 THEN
    WRITELN('RENAME FAILED. ',OLDNAME,' NOT FOUND.')
  ELSE
    WRITELN('FILE ',OLDNAME,' RENAMED TO ',NEWNAME);

END.
```

Listing 4-4. 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, LONGINT, 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.

On the 8086, local variables specified by name evaluate to a word containing the offset into the appropriate Stack segment (based upon BP). Global variables evaluate into a word containing their offset in the Data segment (from the DS register).

The *+n and *-n of 8-bit versions of Pascal/MT+ are unnecessary on the 8086, because all jumps are relative to the base of the segment.

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

Because of the complexity of the assembly language, Pascal/MT+ does not have a built-in mini-assembler.

The following listing shows how to use INLINE to store values in the ES register after calling @BDOS86.

```
TYPE
  PTR = ^INTEGER;

EXTERNAL FUNCTION @BDOS86(FUNC:INTEGER; PARM:PTR):INTEGER;

FUNCTION ES_REG(FUNC:INTEGER; PARM:PTR):INTEGER;
VAR
   ESVAL : INTEGER;      (* SO WE CAN STORE IT HERE *)
   (* ASSUME A GLOBAL VARIABLE CALLED BDOSVAL *)
   (* IN WHICH TO STORE THE RESULT FROM @BDOS86 *)

 BEGIN
   BDOSVAL := @BDOS86(FUNC,PARM);
   (* NOW USE INLINE TO STORE THE VALUE OF ES *)
   INLINE($8C/     (* MOV large_offset[BP],ES opcode *)
          $86/     (* second byte of opcode *)
          ESVAL); (* referencing var places a word of offset here *)
   ES_REG:= ESVAL;       (* SET FUNCTION VALUE *)
 END;
```

Listing 4-5. Using INLINE to Store Values in ES Register

The listing on the next page demonstrates how INLINE constructs compile-time tables.

```
PROGRAM DEMO_INLINE;

TYPE
    IDFIELD = ARRAY [1..4] OF ARRAY [1..10] OF CHAR;
    IDPTR   = ^IDFIELD;
    (* THIS WORKS ONLY ON THE 8086 *)
    POINTERKLUDGE = RECORD
                      CASE BOOLEAN OF
                        TRUE : (P : IDPTR);
                        FALSE: (LOWORD : WORD;
                                HIWORD : WORD)
                      END;

VAR
  TPTR : IDPTR;
  P    : POINTERKLUDGE;

PROCEDURE TABLE;
BEGIN
  INLINE(    'MTMICROSYS' /
             'SOFTWARE  ' /
             'POWER     ' /
             'TOOLS.....' );
END;

BEGIN (* MAIN PROGRAM *)
  P.P := ADDR(TABLE);
  P.LOWORD := P.LOWORD + WRD(8);
  TPTR := P.P;

  WRITELN(TPTR^[3]); (* SHOULD WRITE 'POWER     ' *)

END.
```

Listing 4-6. Using INLINE to Construct Compile-time Tables

Here, the ADDR of TABLE must be added to its offset. This is because ADDR does not give the address of TABLE, due to additional code that recursion management produces. An extra eight bytes of code is generated.

Note: the table must be in the same module as the statement that takes the ADDR of TABLE.

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 [$40:$8000] INTEGER;
SCREEN: ABSOLUTE [$2000:$C0] ARRAY[0..15, 0..63] OF CHAR;
```

Note that you must put the address of the variable in brackets [...].

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

String variables might not exist at all locations. On the 8086, strings must not be in segment 0FFFFH, in order that the run-time subroutines 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>] <identifier> ;
```

Interrupt procedures are not restricted to the main program; modules can also contain interrupt procedures.

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.

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.

Unlike most 8-bit implementations, the 8086 implementation of Pascal/MT+ generates reentrant code. However, some language facilities, specifically Console I/O, File I/O, COPY, and CONCAT, require statically-allocated data. While you can access these facilities from an interrupt procedure, nothing prevents interrupting a program segment that uses these facilities.

CP/M-86 is not reentrant; therefore Console I/O and File I/O cannot be used in an interrupt procedure. If you use CP/M-86, note that I/O through the CP/M-86 BDOS reenables interrupts only if they were enabled when BDOS was entered.

To disable interrupts around sections of Pascal code, use INLINE to place CLI ($FA) and STI ($FB) instructions around the code.

The following program waits for an interrupt on one of four switches, 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 interrupts $22, $23, $24 and $25.

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

  SWITCH1 = $22            (* DEFINE INTERRUPT VECTORS *)
  SWITCH2 = $23;
  SWITCH3 = $24;
  SWITCH4 = $25;

TYPE
  PTR = ^INTEGER;          (* FOR USING ADDR FUNCTION *)

VAR     (* define the low memory we want to use *)

  VEC22 : ABSOLUTE [0:$88] PTR;
  VEC23 : ABSOLUTE [0:$8C] PTR;
  VEC24 : ABSOLUTE [0:$90] PTR;
  VEC25 : ABSOLUTE [0:$94] PTR;

  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
6END;

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

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

Listing 4-7. Using Interrupt Procedures

```
BEGIN (* MAIN PROGRAM *)

  (* FIRST INITIALIZE THE INTERRUPT VECTORS *)

  VEC22 := ADDR(INT1);
  VEC23 := ADDR(INT2);
  VEC24 := ADDR(INT3);
  VEC25 := ADDR(INT4);

  (* 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;

  INLINE($FB);  (* STI INSTRUCTION *) (* LET THE USERS HAVE AT IT! *)

  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;  (* DO THIS LOOP FOREVER *)

END. (* OF MAIN PROGRAM *)
```

Listing 4-7. (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.R86. 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.R86 run-time library. When you use this method,

- you treat the heap as a stack, and 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 UCSD Pascal's MARK and RELEASE routines by using the built-in routines @MRK and @RLS, as shown in this example:

```
MODULE UCSDHEAP;

EXTERNAL FUNCTION @MRK : LONGINT;
EXTERNAL FUNCTION @RLS (L:LONGINT);

PROCEDURE MARK(VAR P:LONGINT);
BEGIN
  P := @MRK
END;

PROCEDURE RELEASE(P:LONGINT);
BEGIN
  @RLS(P)
END;

MODEND.
```

4.4 Recursion/Nonrecursion

Pascal/MT+ always produces recursive code, because degradation in code size and execution speed is minimal on the 8086.

Return addresses and local variables for all procedures are stored on the hardware stack. If recursion is deeply nested, and the default stack size is too small, the program can overwrite local or global data as recursion continues. You can solve this problem by specifying a larger hardware stack, using the /Z linker option.

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 not 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 subroutines @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-86 BDOS in your PROM. If you are constructing your program to run in a totally stand-alone environment such as an Intel SBC-86/12 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-86 BDOS for Functions 1: Console Input, 2: Console Output, and 5: List Output.

     The function number is in the CL register; the data for output is in DL.

     For input, Function 1, return the data in the AL 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) If you use a ROM-based system, you might shorten or eliminate the INPUT and OUTPUT FIB (File Information Block) 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 (wait for EOLN to be TRUE on a file).

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

If your program does not do READLN or WRITELN calls and does not use the heap or overlays, you can rewrite the @INI procedure in your program as:

```
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) Link any changed run-time routines before you link PASLIB.R86, the run-time library, to resolve the references. Use the /S linker option, as in the following example.

```
A>LINKMT USERPROG,MYWNC,MYRNC,MYGET,MYINI,PASLIB/S
```

4.6 Error and Range Checking

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

Error checks and routines set Boolean flags. These flags, along with an error code, load onto the stack and call the predefined routine @ERR that 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 takes appropriate action, as described in Table 4-2.

Table 4-2. @ERR Routine Error Codes

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

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 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. The conditions checked for are the following:

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

In the current release, $X- does not disable exception checking.

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: @ERR prints FLOATING-POINT OVERFLOW. The result of the operation is a large number.

- Division by zero: @ERR prints DIVIDE BY ZERO DETECTED. The result is a representation of the largest-possible number.
- Heap overflow: nothing happens. You should test the value of @HERR to detect heap overflow.
- 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, @BDOS86, does not handle I/O errors. However, it returns the CP/M-86 error code in IORESULT. You can rewrite @BDOS86, using the supplied assembly-language source, to check further for disk I/O errors.

End of Section 4

Section 5
Pascal/MT+ Programming Tools

Pascal/MT+ provides four programming tools designed to increase programming productivity: an assembler, a disassembler, a debugger, and a librarian.

- ASMT-86 is an assembler that is upward-compatible with the Intel MCS-86™ assembler. The ASMT-86 assembly language is a subset of the MCS-86 language, see Section 6.
- DIS-86 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.
- LIB/MT+86 is a software librarian utility that concatenates relocatable files into a searchable library file.
- The debugger is a relocatable file that you link into a program, enabling you to step through the program as it runs.

5.1 ASMT-86, the Assembler

The ASMT-86 assembler supports a subset of the MCS-86 assembly language. ASMT-86 does not provide codemacros, macros, records, or structures. There are other restrictions to the ASMT-86 language that are summarized at the end of Appendix F. For a detailed description of the MCS-86 language, see the MCS-86 Macro Assembly Language Reference Manual.

ASMT-86 consists of an executable command file and four overlays. Your Pascal/MT+ distribution disk #3 contains the following five files:

- ASMT86.CMD
- ASMT86.001
- ASMT86.002
- ASMT86.003
- ASMT86.004

When assembling, all five files must be on one logged-in logical drive. The assembler has an error-message file, ASMERS.TXT, that you can place on any logical drive.

5.1.1 Assembler Operation

ASMT-86 takes assembly-language programs as input and generates a relocatable object file and a print file. During assembly, ASMT-86 creates the following temporary work files:

- ASMTMP.TOK
- ASMTMP.LST
- ASMTMP.ERS

ASMT-86 automatically erases these temporary files when the assembly reaches a normal completion. ASMT-86 also erases the temporary files if they are on disk before you start the assembly.

ASMT-86 generates a relocatable object file with filetype R86 and the same filename as the assembly-language source file. If you specify the P option in the command line, ASMT-86 generates a print file with filetype PRN and the same filename as the source file. Figure 5-1 illustrates the operation of ASMT-86.

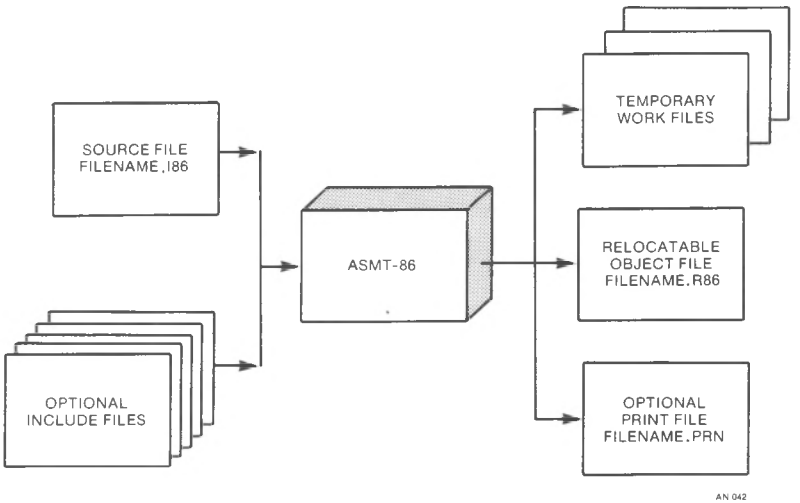


Figure 5-1. ASMT-86 Operation

5.1.2 Invoking ASMT-86

You invoke ASMT-86 with a command line of the following format:

ASMT86 <filespec> {$<options>}

where the <filespec> must be a standard Digital Research filespec with filetype I86, and <options> are optional parameters that control the assembly.

The assembler assumes an I86 filetype if you omit it in the filespec. The dollar sign separates the <options> from the rest of the command line. You can also use a pound sign, #, instead of the dollar sign. You do not have to use either sign if you do not specify any options.

5.1.3 ASMT-86 Command Line Options

The ASMT-86 supports six command line options, as described in the following table.

Table 5-1. ASMT-86 Command Line Options

Option	Meaning
C	The assembler continues on errors. The default is to wait for your response.
Ed	The error list file ASMERS.TXT is on drive d. The default is the logged-in drive.
Pd	The print file goes on drive d. The default, disk Z, produces no print file. X refers to the console.
Q	Quiet operation. The assembler writes fewer messages to the console.
Rd	The relocatable object file goes on drive d. The default is the logged-in drive.
Td	The temporary files ASMTMP.TOK, ASMTMP.LST, and ASMTMP.ERS go on drive d. The default is the logged-in drive.

5.2 DIS-86, the Disassembler

The Pascal/MT+ disassembler consists of one executable file, DIS86.CMD which is on distribution disk #1.

DIS-86 generates a file showing the assembly language for each Pascal/MT+ source line. When you compile a program using the X option, the Pascal/MT+ compiler generates an extended relocatable file that contains 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. Appendix C contains a listing of a sample disassembly. Figure 5-2 illustrates the operation of DIS-86.

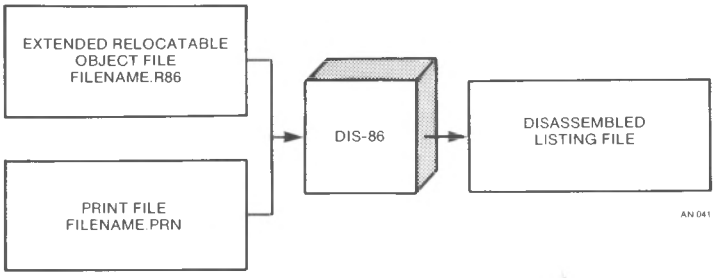


Figure 5-2. DIS-86 Operation

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

```
DIS86 <filename> [<destination name>][,L=nnn]
```

You do not have to specify a filetype. The disassembler searches for both the R86 and PRN files with the specified <filename>. Note that 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. nnn is an integer value. The L=nnn parameter requires that you specify a <destination name>.

When the disassembler finds something unexpected in the R86 file, it generates an error message. Continuing at this point produces more errors because the sequence is off. An R86 file should have no errors. To correct errors, recompile the program. Be sure you are disassembling Pascal code only.

5.3 LIB/MT+86, the Software Librarian

LIB/MT+86 is the Pascal/MT+ software librarian that logically concatenates R86 files together to construct a searchable library such as PASLIB.

5.3.1 Invoking LIB/MT+86

You invoke the librarian with a command of the form:

```
LIBMT <filename>
```

where <filename> contains only the name, not the type of the file. LIB/MT+86 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 valid as input. You must specify the filetype but it need not be R86. If the output file is to be processed by LINK/MT+86, it must be of type R86.

Note: LIB/MT+86 cannot process a Pascal 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+86 compatible library:

```
MYLIB.R86
MYMOD1.R86
MYMOD2.R86
MYMOD3.R86
```

This file first deletes any existing copy of MYLIB.R86. It then concatenates the files MYMOD1.R86, MYMOD2.R86 and MYMOD3.R86 and places the output into the file MYLIB.R86.

5.3.2 Searching a Library

LINK/MT+86 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 module cannot be extracted. All entry points, both code and data, are used as a basis for searching when the /S option is used. Only one entry point in a module need be referenced to force loading that entire module.

You cannot use LIB/MT+86 to alter PASLIB because of its special construction. If you want to replace modules in PASLIB, 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 it because the references to the replacement routines sometimes are not made until PASLIB is processed.

5.4 Debugger

The Pascal/MT+ debugger simplifies program maintenance. The debugger consists of one relocatable object file, DEBUGGER.R86, which is on distribution disk #2.

To use the debugger, you must link the DEBUGGER.R86 file into a source program along with the run-time subroutine library. The debugger then takes charge of the source program execution. The debugger can perform the following tasks:

- display variables
- 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 only works 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.

5.4.1 Debugging Programs

The compiler generates a PSY file containing debugger information when you specify the D option in the command line. 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 R86 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 procedure or function for debugger breakpoint logic. Address fields for each item are module relative.

The linker uses the R86 and PSY file to create a SYP file that contains 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.R86 file first in the list of files in the LINK/MT+86 command line. The following example links the debugger, a user program, and run-time library into an executable file named USERPROG.CMD.

```
A>LINKMT USERPROG=DEBUGGER,USERPROG,PASLIB/S
```

The example above generates two undefined symbols required to write real numbers, @XOP and @WRL. The undefined symbols cause no problem if USERPROG does not use real numbers. If USERPROG uses real numbers, you must link the real number run-time library file with the other files in the command line.

To start the debugging session, run the program. The debugger takes control of the program, 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 the other debugger commands.

5.4.2 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 1 to 8 characters long and follow the same syntax as 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. Pointers are 32-bit segment pairs. For example, A refers to the four-byte pointer at A, not just two bytes as in 8-bit versions of Pascal/MT+.
- 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> | $<num1A> : <num1B>] {^} {[+|-] <num2>}
```

If you do not use a <name>, you can specify an address in Data-segment relative or absolute mode. A single <num> indicates an address that is relative to the value of the DS register. If you use two numbers, <num1A> is the segment number and <num1B> is the offset from that segment. <num2> specifies the number of bytes to add or subtract from the address already attained in the parameter. Table 5-2 shows examples of parameters given the following declarations:

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

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

Table 5-2. Examples of Parameters

Parameter	Meaning
ABC	an integer
PTR	contents of PTR
PTR^	entire array
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	data segment relative location
$423B^	32 bytes pointed to by DS:$423B
$3FFD+$5B	32 bytes at DS:$4058
$423B^+49	32 bytes pointed to by contents of DS:$423B + 49
$34F:2500	absolute location
PROC1:I	local variable in PROC1
PROC2:J^+9	offset from local pointer

The command to display a variable by <name> is 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 5-3 shows commands used when symbols are not available or when you want to display fields within record or array elements.

Table 5-3. Debugger Display Commands

Command Syntax	Meaning
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 command to alter the contents of a memory address is SE<parm>. The command displays the byte at the specified address in decimal. Enter a new value in either decimal or hex 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.

The following table describes the other commands that enable control of your program in a debugging session.

Table 5-4. Debugger Control Commands

Command Syntax	Meaning
BE	BEgin execution, start program from beginning.
E+	Enable display entry and exit of each procedure or function during execution (default on).
E-	Disable entry / exit display.
GO	Continue execution from a breakpoint.
PN	Display procedure names from SYP file.
RB <name>	Remove breakpoint at procedure <name>.
SB <name>	Set breakpoint at beginning of procedure <name>.
SE <parm>	Modify contents of memory at <parm>. A '.' terminates this command.
TR or T	Trace - Execute one line and return.
T<num>	Trace <num> lines and return.
VN <name>	Display variables associated with procedure <name>.
??	HELP! List of commands found in DBUGHELP.TXT.

End of Section 5

Section 6
ASMT-86 Assembly Language

You can create assembly-language source files using any text editor or word processor that produces standard CP/M-86 text files.

Identifiers can be any length, but only the first 31 characters distinguish one identifier from another. Lower-case characters are equivalent to upper-case characters except when enclosed in single apostrophes.

A source line cannot exceed 132 characters, and strings enclosed in apostrophes must fit one physical line.

An assembly-language program is a sequence of statements followed by an end-of-file mark. Assembly-language statements use the following general format:

<label identifier>[:]<opcode><arguments>

The <label identifier> must begin in column 1. Certain <opcodes> require a <label identifier> and others require that you do not specify one. Use the colon only when declaring a label for a machine opcode.

ASMT-86 assembly language is a subset of the Intel MCS-86 assembly language. Appendix F lists the syntax and reserved words for the ASMT-86 language.

6.1 Pseudo-opcodes

The symbolic opcode names are the instruction mnemonics used to generate the corresponding bit patterns in the object file. Opcodes that are not standard 8086 machine opcodes are called pseudo-opcodes.

ASMT-86 assembly language supports several pseudo-opcodes, described as follows. Refer to Appendix F for the pseudo-opcode syntax diagrams.

- SEGMENT/ENDS: the SEGMENT statement and the ENDS statement control assembly-program segmentation. The SEGMENT directive includes an optional alignment type, optional combine mode, and a class name string. The following examples are commonly used forms of the SEGMENT directive:

```
CODE          SEGMENT PUBLIC
DATA          SEGMENT PUBLIC
OTHER_NAME    SEGMENT AT <number>
```

The label on an ENDS statement must match the label on the corresponding SEGMENT statement. You cannot nest segments. However, you can code a portion of a segment, start and end a second segment, then continue coding the first segment. The resulting code does not actually contain nested segments; the assembler puts the separate parts back together.

- ASSUME: the ASSUME statement indicates which segment register points to a specified segment. Following the key word ASSUME, enter the segment register and the segment name with an optional variable.

- LABEL: the LABEL statement creates a label in the current location of the program. Specify the type or distance of the label following the keyword LABEL.

- PROC: the PROC statement enables procedure declarations. You can specify the distance following the keyword PROC. The distance defaults to NEAR. You must match each PROC statement with an ENDP statement. The label preceding each ENDP statement must match the label on the corresponding PROC statement. Declaring procedures, as opposed to simply calling labels, informs the assembler about the distance needed to determine how to assemble the RET instruction.

- NAME: the NAME statement specifies a new name for the assembled relocatable object file.

- PUBLIC: the PUBLIC statement declares certain labels PUBLIC. Other modules can reference PUBLIC labels.

- EXTRN: the EXTRN statement can reference labels in other modules that are PUBLIC. The EXTRN statement includes the label followed by a type.

- END: the END directive establishes a starting address for the program. The identifier following END must be a label in the Code segment.

- ORG: the ORG directive establishes a new location counter offset within the current segment. The expression must result in a defined value.

- DB, DW, DD: the DB, DW, and DD statements allocate and initialize data space. Place the initializing data after the keyword. The following examples show the types of initializing data.

```
?
expression
constant-expression DUP (expression or ?)
```

The question mark indicates that the data space is to be allocated, but not initialized. An expression initializes the data space with the value of the expression. The DUP form

enables space allocation to repeat the number of times indicated by the constant-expression. The constant expression can be any expression that evaluates to an absolute number. Unlike MCS-86 assembly language, you cannot nest DUP expressions.

DB accepts strings enclosed in single apostrophes. DW and DD only accept strings enclosed in single apostrophes up to 2 characters. DB allocates characters in low-to-high order. DW and DD allocate characters in high-to-low order.

Note that the linker currently permits only uninitialized data in the Data segment.

- EQUATE: you can equate identifiers to expressions using the EQUATE statement. The form using ON and OF is for conditional assembly options in IF, ELSE, and ENDIF.

- IF, ELSE, ENDIF: IF, ELSE, and ENDIF let you assemble conditionally. You must equate the identifier following IF with ON or OFF. If the identifier equals on, the text up to ELSE assembles and the text between ELSE and ENDIF is ignored. If the identifier equals OFF, the text between ELSE and ENDIF assembles and the text up to else is ignored. ELSE is optional. IF, ELSE, and ENDIF sets cannot be nested.

- INCLUDE: you can textually include a separate file in the assembly with INCLUDE. The string following INCLUDE is the name of the file you want to include. If you do not specify a drive name, the assembler uses that of the original filename. If you do not specify a filetype, the original filetype is used.

  You cannot nest INCLUDE files, nor can an included file contain other INCLUDE files. The following examples show proper INCLUDE statements.

```
INCLUDE 'incfile'         ;drive and filetype are the
                          ;same as the main input file.
INCLUDE 'b:2ndfile.txt' ;drive is b and filetype is .txt
```

6.2 Fundamental Values

Fundamental values are expressed as numbers, character strings, or variables. Numbers can be binary, decimal, octal, or hexadecimal.

- Decimal numbers must start and end with a digit from 0 - 9.
- Binary numbers end with B.
- Decimals end with D.
- Hexadecimals end with H.
- Octals end with O or Q.
- A $ is a break character that does not effect the number.

All numbers are stored internally as 16-bit signed two's complement numbers. Only the low-order 16-bits of their representation is kept for numbers greater than 32767 or less than -32768. The following examples show valid numbers:

```
1111$0000$0000$1101B        = -4083
10000H                      = 0
65537                       = 1
377777q                     = -1
```

You can use strings of 2 or fewer characters as numbers. The characters convert to their ASCII numerical representation. For a single-character string, the character goes in the low-order byte of the word. The high-order byte is 0. For a 2-character string, the first character goes in the high-order byte of the word and the second goes in the low-order part. The following examples show numbers expressed as literal character strings:

```
'A'         = 0041H
'AB'        = 4142H
```

Variables contain attribute information. A variable's length is the number of units, not necessarily bytes, allocated for the variable. This value derives from the repeat factor that declares the variable in a DB, DW, or DD statement.

A variable's type is 1 if you declare it with DB, 2 with DW, and 4 with DD. The size of a variable is the number of bytes allocated for it. Variable size equals length times type. The following examples show variable declarations:

```
Byte_variable   DB      15 DUP (0)

Word_variable   DW      'hi'

; For Byte_variable, TYPE = 1, LENGTH = 15, and SIZE = 15

; For Word_variable, TYPE = 2, LENGTH = 1, and SIZE = 2
```

You can use this information with expressions of the form TYPE id, LENGTH id, or SIZE id.

When an identifier appears in an expression, its value is the offset of a variable or label or the base value of a segment. If the segment with the identifier declaration is relocatable, the value is relocatable.

6.3 Operators

You can use the value of a current assembly-location counter as a fundamental value through the THIS operator. Follow THIS with a type, BYTE, WORD, DWORD, NEAR, or FAR. The offset of the resulting expression has the indicated type and an offset equal to the current location counter. For example,

```
JMP             THIS NEAR       ;infinite loop
MOV             AL,THIS BYTE    ;moves a MOV opcode byte into
                                ;AL (assuming DS points to
                                ;this segment)
```

The $ operator is equivalent to THIS NEAR.

Logical operators operate only on absolute numbers. OR, XOR, and AND are infix dyadic operators. NOT is a monadic operator. The following examples show the use of logical operators.

```
DB      0 OR 0FFH         ;result is 0FFH
DB      0101B XOR 0001B   ;result is 0100B
DW      'AB' AND 00FFH    ;result is 42H = 'B'
8DW     NOT 0             ;result is -1
```

Like the logicals, the shift and multiplicative operators operate only on absolute numbers. These dyadic infix operators are * , / , MOD , SHL , and SHR. Examples of each are shown below.

```
MOV     AX,2*2          ;put 4  in AX
MOV     AX,2/2          ;put 1  in AX
MOV     AX,5 MOD 2      ;put 1  in AX
MOV     AX,5 SHL 2      ;put 20 in AX
MOV     AX,5 SHR 2      ;put 1  in AX
```

The operation of MOD is undefined for negative numbers.

You can compare expressions with the comparison operators LT, LE, GE, GT, EQ, and NE. The expressions must be absolute numbers or relocatable relative to the same segment. The following examples show the use of comparison operators.

```
A:      DW      0 LT 1  ;result is 0FFH for TRUE
B:      DW     -1 GT 0  ;result is 0 for FALSE
        DW      A LT B  ;result is TRUE since OFFSET A < OFFSET B
```

HIGH and LOW extract the high- or low-order byte from an expression. You can evaluate absolute or relocatable expressions, but the code generation cannot handle relocatable high or low bytes. The following examples show the HIGH and LOW operators.

```
A:      DB      HIGH 0FF00H             ;result is 0FFH
        DW      HIGH LOW OFFSET A       ;result is 0
```

You can add and subtract expressions. At least one of the expressions must be absolute or an index register in brackets, see Section 6.6. To subtract, the second expression must be either absolute or relocatable relative to the same base as the first. The following examples show the addition and subtraction operators.

```
A:      DW      5-3         ;result is 2
B:      DW      A + 2       ;result is relocatable and equivalent to B
        DW      B - 2       ;result is relocatable and equivalent to A
        DW      B - A       ;result is 2
```

6.4 Expressions

A sequence of fundamental values and operators is an expression. Operators connect fundamental values and expressions to form new expressions. ASMT-86 classifies expressions by the kind of information they represent; an expression is either a variable or a number.

- The label on a LABEL statement or on a data-initialization statement is a variable expression.
- A constant such as 2 is a number expression.

Whether an expression is a variable or a number, it can be relocatable or absolute. To be absolute, a variable must be declared in an absolute segment.

To be relocatable, a number must derive from a relocatable number through OFFSET, HIGH, LOW, or SEG operators on a relocatable variable or segment identifier. The expression computation allows all these forms of relocatable numbers. The object format, however, allows only offset relocatability.

6.5 Attribute Overrides

You can override the segment attribute of an expression two ways.

- You can use a segment name to override the owner of an expression. The offset of the resulting expression is the same as in the original, but its relocation base is the named segment. The form of this override is

  segment-name : expression.

- You can override a segment attribute with the form:

  segment-register : expression

  This form lets you choose which segment register to use in accessing the expression. For example,

```
                ASSUME ES:EXTRA_SEGMENT
EXTRA_SEGMENT   SEGMENT AT 0FFF0H
A_VARIABLE      DW      ?
EXTRA_SEGMENT   ENDS
                  .
                  .
                  .
CODE            SEGMENT PUBLIC
                MOV     AX,EXTRA_SEGMENT
                MOV     ES,AX
; move 9999H into the word ES points to
; (in two slightly different ways)
                MOV     EXTRA_SEGMENT : WORD PTR 0 , 9999H
                MOV     ES : WORD PTR 0 , 9999H
                  .
                  .
                  .
CODE            ENDS
```

SEG lets you find the value of a segment with a variable or label. In the example above, SEG A_VARIABLE returns 0FFF0H.

OFFSET converts a variable or label to a number. It does not change the relocatability of the expression.

PTR changes an expression's type or distance attribute. The form is type or distance PTR expression. For example,

```
A_BYTE  DB      0
B_BYTE  DB      1

        MOV     AX,WORD PTR A_BYTE   ;A goes to AL, B goes to AH

NEAR_LABEL:
          .
          .
          .
        CALL    FAR PTR NEAR_LABEL   ;Generates a long call
                                     ;to NEAR_LABEL
```

6.6 Indexing Expressions

Put indexing expressions in brackets, []. They follow other expressions to indicate an index or base register for accessing those expressions. The form is

[expression]

An expression within brackets can contain BX, BP, SI, and DI. You can use multiple indexing expressions. Only one base, BP or BX, and one index, SI or DI, are allowed for each expression. For example,

```
A   DB    200 DUP (0)
    .
    .
    .
    MOV  A[BX],0 ;move a 0 into the BX'th byte of A
    MOV  A[BP+SI],0 ;move a 0 into the BP+SI 'th
                    ;byte of A
    MOV  AL,A[DI+5] ;fetch the DI + 5 'th byte of A
```

Use parentheses to specify the precedence of operations in an expression, as in the following example.

```
DB 2*(3+5)          ;do the addition before the multiply
                    ;result is 16
```

End of Section 6

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, etc. 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	(e.g. '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.
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.

Table A-1. (continued)

Message	Meaning
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, etc.
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.
Error # 128 Result type does not agree with declaration	When assigning to a function result, the types must be compatible.

Table A-1. (continued)

Message	Meaning
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.
Error # 136 Set element type must be scalar or subrange	Self-explanatory.
Error # 137 Set element types must be compatible	Self-explanatory.

Table A-1. (continued)

Message	Meaning
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.
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.

Table A-1. (continued)

Message	Meaning
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.
Error # 151 Assignment to formal function is not allowed	Self-explanatory.
Error # 152 No such field in this record	Self-explanatory.

Table A-1. (continued)

Message	Meaning
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.
Error # 159 Real or string tagfields not allowed	Self-explanatory.
Error # 160 Previous declaration was not forward	

Table A-1. (continued)

Message	Meaning
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	
Error # 170 Value parameter expected	

Table A-1. (continued)

Message	Meaning
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 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.
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.

Table A-1. (continued)

Message	Meaning
Error # 259 Expression too complicated	Your expression is too complicated (that is, too many recursive calls 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 .OBJ file was closed. Make more room on the destination disk and try again.
Error # 500 Non ISO Standard feature. Not fatal.	

Table A-1. (continued)

Message	Meaning
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 8086 version 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+86 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, because 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 was 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.

Output from compiler:

```
Pascal/MT+86  Release 3.0  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+86  Release 3.1  Copyright (c) 1982 by Digital Research
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
   2    0     PROGRAM PPRIME;

0000            DB      90,90,90,90,90,90,90,90
0008            DB      90,90,90,90,90,90,90,90
0010            JMP 0013

   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

0013            CALL 0000
0016            MOV BP,SP
0018            DEC BP
0019            DEC BP
001A            PUSH BP

  12    1      COUNT := 0;

001B            MOV WORD PTR 2008",0000
```

Listing C-2. Disassembly of PPRIME

```
  13    1       writeln('10 iterations');

0021            PUSH DS
0022            MOVI AX,0000
0025            PUSH AX
0026            CALL 0000
0029            PUSH CS
002A            CALL 000E
002D            DB      0D,31,30,20,69,74,65,72
0035            DB      61,74,69,6F,6E,73
003B            MOVI AX,FFFF
003E            PUSH AX
003F            MOVI AX,FFFF
0042            PUSH AX
0043            CALL 0000
0046            CALL 0000

  14    1       FOR ITER := 1 TO 10 DO

0049            MOV WORD PTR 2000",0000
004F            MOV WORD PTR 202A",000A
0055            INC WORD PTR 2000"
0059            DEC WORD PTR 202A"
005D            JGE 0062
005F            JMP 00E8

  15    1        BEGIN
  16    2          COUNT:=0;

0062            MOV WORD PTR 2008",0000

  17    2
  18    2            FILLCHAR(FLAGS,SIZEOF(FLAGS),CHR(TRUE));

0068            PUSH DS
0069            MOVI AX,0000"
006C            PUSH AX
006D            MOVI AX,1FFF
0070            PUSH AX
0071            MOVI AX,0001
0074            PUSH AX
0075            CALL 0000
```

Listing C-2. (continued)

```
  19    2
  20    2          FOR I:=0 TO SIZE DO

0078            MOV WORD PTR 2006",FFFF
007E            MOV WORD PTR 202C",1FFF
0084            INC WORD PTR 2006"
0088            DEC WORD PTR 202C"
008C            JGE 0091
008E            JMP 00E5

  21    2            IF FLAGS[I] THEN

0091            NOP
0092            MOVI AX,0000"
0095            ADD AX,2006"
0099            XCHG AX,DI
009A            TEST BYTE [DI],01
009D            JNZ 00A2
009F            JMP 00E3

  22    2              BEGIN
  23    3                PRIME:=I+I+3;

00A2            MOV AX,2006"
00A5            ADD AX,2006"
00A9            ADDI AX,0003
00AC            MOV 2004",AX

  24    3                K:=I+PRIME;

00AF            MOV AX,2006"
00B2            ADD AX,2004"
00B6            MOV 2002",AX

  25    3                WHILE K<=SIZE DO

00B9            CMP 2002",1FFE
00BF            JLE 00C4
00C1            JMP 00DC

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

00C4            NOP
00C5            MOVI AX,0000"
00C8            ADD AX,2002"
00CC            XCHG AX,DI
00CD            MOVE BYTE PTR [DI],00
```

Listing C-2. (continued)

```
  28    4                    K:=K+PRIME;

00D0          MOV AX,2002"
00D3          ADD AX,2004"
00D7          MOV 2002",AX

  29    4                  END;

00DA          JMPS 00B9

  30    3               COUNT:=COUNT + 1;

00DC          MOV AX,2008"
00DF          INC AX
00E0          MOV 2008",AX

  31    3             END
  32    3       END;

00E3          JMPS 0084
00E5          JMP 0055

  33    1     writeln(count,' primes');

00E8          PUSH 2008"
00EC          PUSH DS
00ED          MOVI AX,0023'
00F0          PUSH AX
00F1          CALL 0027'
00F4          MOVI AX,FFFF
00F7          PUSH AX
00F8          MOVI AX,FFFF
00FB          PUSH AX
00FC          CALL 0000
00FF          PUSH CS
0100          CALL 0008
0103          DB      07,20,70,72,69,6D,65,73
010B          MOVI AX,FFFF
010E          PUSH AX
010F          MOVI AX,FFFF
0112          PUSH AX
0113          CALL 0044'
0116          CALL 0047'

  34    1   END.

0119          CALL 0000
```

Listing C-2. (continued)

```
External reference chain @WIN    --> 00FD
External reference chain @CRL    --> 0117
External reference chain @SFB    --> 00F2
External reference chain @INI    --> 0014
External reference chain @WRS    --> 0114
External reference chain @HLT    --> 011A
External reference chain OUTPUT  --> 00EE
External reference chain FILLCH  --> 0076
```

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>MT+86 B:DEBUG $D                                  Compile the program with the Debug option.

------------------------------------------------
MT+86 - Pascal  V3.1  Serial. No. XXXX-0000-100001  System displays banner.
Copyright (C) 1982
Digital Research, Inc.          All Rights Reserved
------------------------------------------------

A>LINKMT B:DEBUG=DEBUGGER,B:DEBUG,PASLIB/S          Link the object file with the debugger.

------------------------------------------------
LINKMT - MT+86  V3.1  Serial No. XXXX-0000-654321   System displays banner.
Copyright (C) 1982                                  Note: the linker might display @WRL as an
Digital Research, Inc.          All Rights Reserved undefined symbol. If your program does not
------------------------------------------------    use real numbers, you can ignore it.

A>B:DEBUG                                           Run program.

------------------------------------------
MT+DEBUGGER  V3.1 Serial No. XXXX-0000-654321       System displays banner.
COPYRIGHT (C) 1982
Digital Research, Inc      All Rights Reserved
------------------------------------------

Symbol table filename (<return> only for none)?  B:DEBUG.SYP   Load the symbols.

Use BEgin or TRace to start a program               Set breakpoint, then
+>SB BREAK                                          start the program.
+>BE

ENTER INTEGER TO CONVERT:                           Enter data.
5
Breakpoint reached

+>DV I                                              Examine I.  It is correct.
Address: 3D4B:0272 Contains: 5

+>DV CH                                             Examine CH.  It is wrong.  Why?  Because
Address: 3D4B:0270 Contains: 0 == 30                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                                       Examine HEXARR[5].  It is not 5.
Address: 3D4B:0263 Contains: 4 == 34

+>DX HEXARR                                         Examine all of HEXARR.  All the digits are
Address: 3D4B:025E Contains:                        off by 1.  Note that HEXARR is a string and
3D4B:025E= 10 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45   .0123456789ABCDE  therefore HEXARR[0] is the length field.  The
3D4B:026E= 46 00 30 00 05 00 00 00 00 00 00 00 00 00 00 00   F.0.............  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.
```

End of Appendix D

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, you should 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

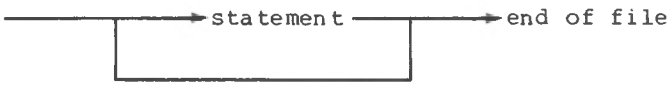
Appendix F
Syntax of ASMT-86

F.1 Syntax Diagrams

The following diagrams represent the complete syntax of ASMT-86. The structure of each syntactic item listed on the left can be determined by following the path, making decisions at each branch.

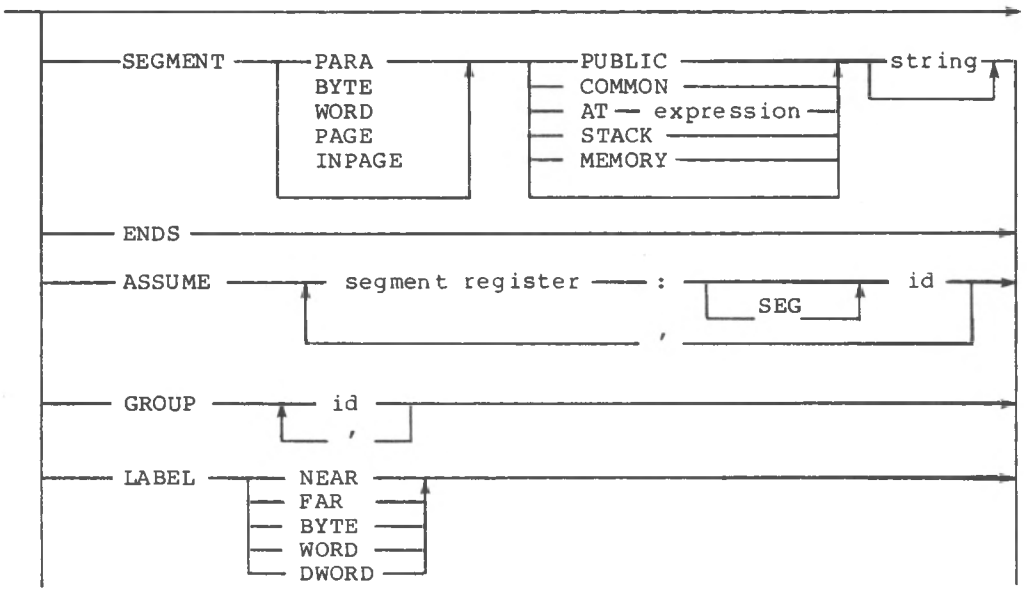
For example, a program is one or more statements followed by the end of the file.

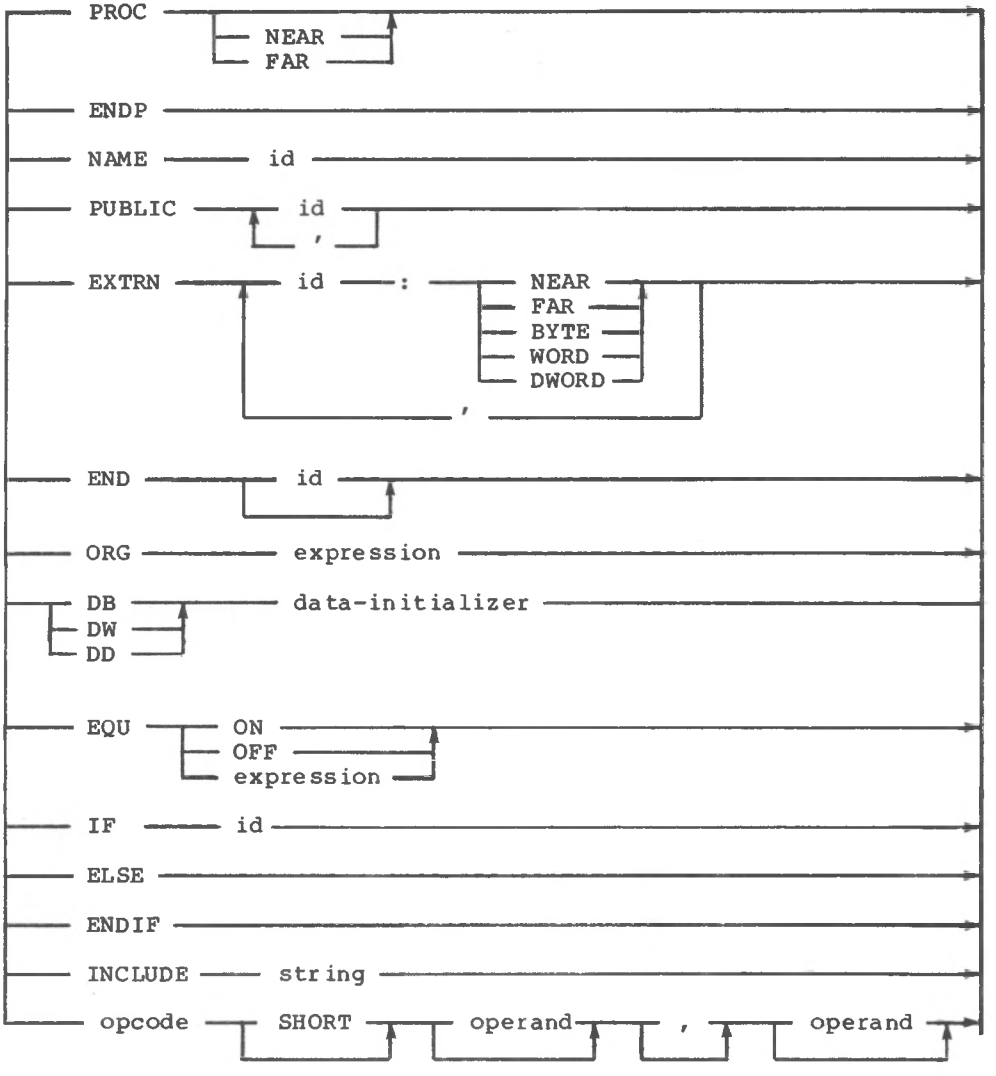
program



statement {note 1}

directive




PROC
NEAR
FAR
ENDP
NAME
id
PUBLIC
id
,
EXTRN
id
:
NEAR
FAR
BYTE
WORD
DWORD
,
END
id
ORG
expression
DB
DW
DD
data-initializer
EQU
ON
OFF
expression
IF
id
ELSE
ENDIF
INCLUDE
string
opcode
SHORT
operand
,
operand

operand

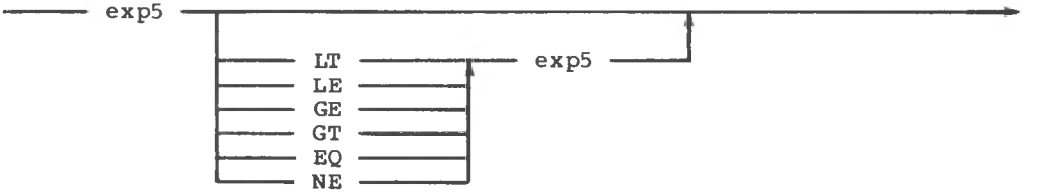

register
expression

expression

exp2

exp3

exp4

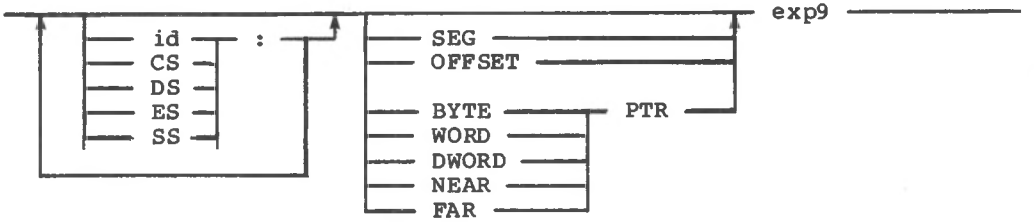


exp5

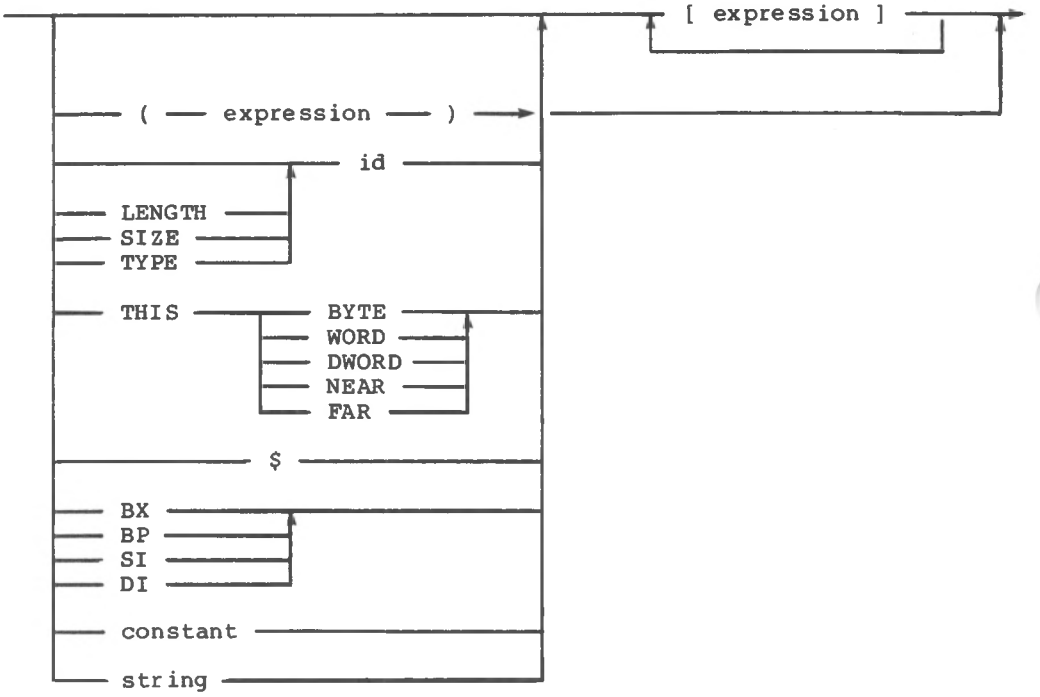
exp6

exp7

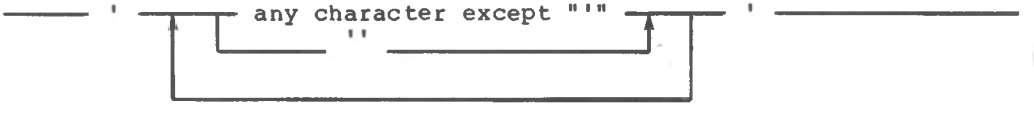
exp8 (attribute overrides)



exp9 (fundamental values and indexing) {note 2}

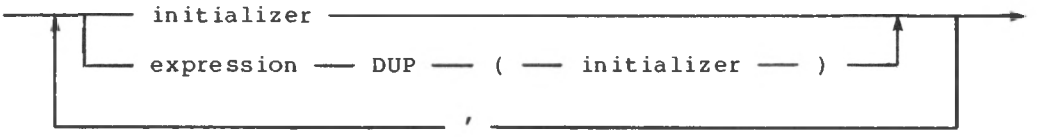


string



jump-operand {for all jump, call and loop instructions}

data-initializer



initializer

Note: label identifiers must start in column 1.

F.2 Reserved Words

The following are reserved words in the assembly language, ASMT-86 and cannot be used as identifiers.

AH	CS	GE	NEAR	SHR
AL	CX	GT	NOT	SI
AND	DH	HIGH	NOTHING	SIZE
AT	DI	INPAGE	OFFSET	SP
AX	DL	LE	OR	SS
BH	DS	LENGTH	PAGE	STACK
BL	DUP	LOW	PARA	THIS
BP	DWORD	LT	PTR	TYPE
BYTE	DX	MASK	PUBLIC	WIDTH
BX	EQ	MEMORY	SEG	WORD
CH	ES	OD	SHL	XOR
CL	FAR	NE	SHORT	

The following words are reserved for use in the operator field. Although you can use these words as identifiers, it is not recommended because of potential confusion.

AAA	END	JNB	LOOPNZ	ROL
AAD	ENDS	JNC	LOOPZ	ROR
AAM	EQU	JNBE	MOV	SAHF
AAS	ESC	JNE	MOVS	SAL
ADC	EXTRN	JNG	MOVSB	SAR
ADD	GROUP	JNGE	MOVSW	SBB
AND	HLT	JNL	MUL	SCAS
ASSUME	IDIV	JNLE	NAME	SCASB
CALL	IF	JNO	NEG	SCASW
CBW	IMUL	JNP	NIL	SEGMENT
CLC	IN	JNS	NOP	SHL
CLD	INC	JNZ	NOT	SHR
CLI	INCLUDE	JO	OR	STC
CMC	INT	JP	ORG	STD
CMP	INTO	JPE	OUT	STI
CMPS	IRET	JPO	POP	STOS
CMPSB	JA	JS	POPF	STOSB
CMPSW	JAE	JZ	PROC	STOSW
COMMON	JB	LABEL	PUBLIC	STRUC
CWD	JBE	LAHF	PUSH	SUB
DAA	JC	LDS	PUSHF	TEST
DAS	JCXZ	LEA	RCL	WAIT
DB	JEL	LES	RCR	XCHG
DD	JG	LOCK	RECORD	XLAT
DEC	JGE	LODS	REP	XLATB
DIV	JL	LODSB	REPE	XOR
DWL	JLE	LODSW	REPNE	
ELSE	JMP	LOOP	REPNZ	
END	JNA	LOOPE	REPZ	
ENDIF	JNAE	LOOPNE	RET	

F.3 Object Format Restrictions

The relocatable object format output by the assembler has the following restrictions:

- Relocatable quantities must be words. That is, relocatable segment base address, low-order bytes of words, or high-order bytes of words are not allowed.

- The only segment that can have initialized data is the segment named CODE. This means that all DB, DW, and DD directives in the DATA, or any other segment must use the indeterminate initializer ?. Furthermore, segments CODE and DATA must be declared PUBLIC as in the sample program.

- A segment not declared PUBLIC is not combinable with other segments of the same name because variables and labels are assembled having absolute offsets within the segment.

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 values reflected in Table G-1 indicate the approximate values you can expect from the 8086 implementation. 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 Card

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

Date _____ First Edition: February 1983

1. What sections of this manual are especially helpful?

_____

_____

_____

_____

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

_____

_____

_____

_____

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

_____

_____

_____

_____

Pascal/MT+™ Language Programmer's Guide for the CP/M-86® Family of Operating Systems

COMMENTS AND SUGGESTIONS BECOME THE PROPERTY OF DIGITAL RESEARCH.

From: _____

_____

_____

_____

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