VICTOR

Applications Programmer's Tool Kit II

Volume II

Applications Programmer's Tool Kit II

Volume II

VICTOR Publications
380 El Pueblo Road
Scotts Valley, California 95066
(408) 438-6680

Second VICTOR printing April, 1984.

ISBN 0-88182-117-9 Printed in U.S.A.

CONTENTS

OVERVIEW

The *Applications Programmer's Tool Kit II—Volume II* consists of these programming tools:

- PMATE — A full-screen, expandable editing system that allows you to create and maintain text files.
- PLINK — A program that takes individually compiled modules of 8086-88 object code, and links them into one or more relocatable files that can be loaded and executed by your computer's operating system.
- PLIB — A program that manipulates libraries of object files; it supplements the PLINK linkage editor.

PMATE

VICTOR Publications
380 El Pueblo Road
Scotts Valley, California 95066
(408) 438-6680

First VICTOR printing February, 1983.
Second VICTOR printing December, 1983.

ISBN 0-88182-108-X Printed in U.S.A.

CONTENTS

APPENDIXES

TABLES

CHAPTERS

PREFACE

PMATE is a text-editing program that you can use to do word processing and to edit program source code and data files. PMATE's basic editing commands are easy to use, even if you have limited experience with text-editing programs.

PMATE has many advanced editing features that make it a fast, flexible text-editing program. With PMATE you can:

- Store blocks of text and commands in any of several edit buffers. You can edit text in these buffers, move text in and out of the buffers, and execute commands from the buffers.
- Store deleted text with a "garbage stack." When you delete text, you can easily recover it from the garbage stack.
- Format text for printing with nonprinting control lines. These control lines format text with margins, tab stops, and indentation. With control lines, you can easily change text formatting without making changes to the text.
- Write your own commands using variables, functions, and arithmetic and logical operations. These commands, called macros, can resemble small text-editing programs.
- Modify the PMATE program to your own specifications, changing PMATE's editing parameters and incorporating the macros you write.

This manual is written for people with various levels of text-editing experience. Every PMATE user should read Chapters 1, 2, and 3. Chapters 1 and 2 introduce you to the basic procedures of PMATE, and Chapter 3 explains the rest of the PMATE features used for text editing.

Chapter 4 is useful primarily to people with extensive text-editing or programming experience. Chapter 4 shows you how to write your own macros; it also demonstrates the PMATE features useful in macros.

Appendix A provides information necessary to configure PMATE to your own specifications. Appendix B provides macro ideas and examples. Appendix C is a list of ASCII characters.

1

BASIC PROCEDURES

After introducing you to the special terms and notation used in this manual, this chapter describes:

- How to enter PMATE.
- PMATE's screen screen display, modes, and two kinds of commands.
- How to do basic editing with PMATE's instant commands.
- How to write a file, read a file, print text, and leave PMATE.

TERMS USED IN THIS MANUAL 1.1

TERM	DEFINITION
ALT key	The ALT key enables special Alternate functions of keys on your keyboard. Hold the ALT key down while typing the key(s) it is to affect. A reference to the ALT key appears in text as ALT, followed by a hyphen and the key(s) it is to affect. For example, ALT-X refers to the Alternate function of the X key; ALT-FC refers to the Alternate function of the F and C keys, which must be typed consecutively to achieve the desired effect.
argument	An argument is a character or string that you use with certain commands. An argument gives PMATE specific instructions on how you want the task done.

TERM	DEFINITION
buffer	A buffer is a place in memory where PMATE stores text.
character	A character is a letter, number, symbol, or space.
control character	A control character is a nonprinting character that appears in reverse video when you enter the character on the screen. You enter a control character by typing ALT-7 and the character. PMATE uses control characters for text formating, file operations, and wild-card searches. Control characters are underlined in this manual—for example, $\underline{\$}$.
drive name	A drive name is the letter that indicates a specific disk drive. In PMATE, you use a drive name to write a file to (or list a directory of) a drive other than the default drive. You also use a drive name to change the default drive. When using the drive name in a command, follow the letter with a colon (:). PMATE displays the drive name (followed by a colon) of the default drive in the top left corner of the screen.
end-of-file character	The end-of-file character is $\underline{Z}$. Type ALT-7 Z to enter this character at the end of a data file (if required).
execute	In PMATE, you execute a command to cause PMATE to carry it out or put it into effect.
filename	A PMATE filename consists of up to eight letters or numbers, with an optional filename extension of up to three letters or numbers. Separate the filename extension from the filename with a period. If you are using subdirectories, you can include the pathname with the filename.

TERM	DEFINITION
form-feed character	The form-feed character is a control character used to separate pages of text for printing and for certain file operations. It appears on the screen as an L in reverse video. (Type ALT-7 L to enter the form-feed character in text. The form-feed character appears in text as L.)
pathname	A pathname is a subdirectory name.
string	A string consists of two or more characters linked together or grouped as a set.
tag	A tag is a temporary, invisible marker that you enter to mark a place in text.
toggle	A toggle key switches from one value to another. PMATE has keys that toggle modes on and off, toggle between modes, and toggle the cursor between two locations.

SPECIAL NOTATION USED IN THIS MANUAL 1.2

NOTATION	DEFINITION
$	$ represents the Escape key, which is Function key 1. (Refer to "Keyboard Considerations" in the *MS-DOS User's Guide.*) When you type $, PMATE displays a dollar sign in reverse video. Type $ twice to execute a command on PMATE's command line. Type $ again to re-execute the command. (You also use $ to separate commands and command strings on the command line, as explained in Chapters 2 and 3.)

NOTATION	DEFINITION
< >	Angle brackets indicate a user-supplied value. For example, < filename > indicates that you should supply a filename of your choice.
* and ?	* and ? are the standard MS-DOS wild-card characters. * matches any string and ? matches any character.
(cr)	In this manual, (cr) indicates that you should press the Return key.

Notes:

You can type PMATE commands in uppercase or lowercase letters.

When an error occurs while you are using PMATE, PMATE puts an error message in the text area of the screen. To delete the error message and resume editing text, press the Return key.

This manual is written for use with a Standard keyboard (see Appendix F in the *MS-DOS User's Guide*).

1.3 ENTERING PMATE

PMATE's three program files are on one of your *Applications Programmer's Tool Kit II* disks. Put this disk in drive A of your computer. These three files should be listed in the directory:

PMATE.COM
CONPMATE.COM
CONFIG.CNF

At the operating system prompt (A >), type the command to enter PMATE, followed by the name of the file you're going to create. If your filename is TEXTFILE, type the command:

```
pmate textfile(cr)
```

THE SCREEN DISPLAY 1.4

When you enter PMATE and specify a text file named TEXTFILE, the screen displays:


```
A: *,TEXTFILE              BUF=T    ARG=0              LIN=0     COL=0
PMATE-86 rev x.y  Copyright Phoenix Software Associates Ltd. 1982
```

The first line of the screen is the **status line**. The status line provides information about PMATE and your text file while you are editing the file. The A: tells you that you are logged on to drive A. The *,TEXTFILE tells you that PMATE will store the text you create on disk as TEXTFILE. LIN = 0 and COL = 0 indicate the line number and column number of the cursor position. The BUF and ARG indicators are discussed in Chapters 3 and 4.

The second line on the screen is the **command line**. The command line displays most of the PMATE commands you execute, as Chapters 1.5 and 1.6 explain. When you enter PMATE, however, its copyright information appears on the command line. If you type any regular key on your keyboard, the copyright information disappears and the character entered by the key appears on the command line. For now, type ALT-C to clear the command line. The cursor (an underline character) remains on the command line.

The rest of the screen is the **text area**. As you create the contents of TEXTFILE (see Chapter 1.5), the text appears in the text area. Although the screen is only 80 columns (characters) wide, the text area is 250 columns wide. If you type a line of text longer than 80 characters, the display moves to the right to follow what you're typing. When you type past column 249, the display moves back to the left and the text appears at the beginning of the next line of text. (The columns are numbered 0 through 249.)

1.5 PMATE MODES

To create a text file, you should be familiar with PMATE's operating modes. These four modes are:

1

- Command Mode
- Insert Mode
- Overtype Mode
- Format Mode

PMATE always operates in one of the first three modes. The fourth, Format Mode, can be on or off while you are in any of the other three modes. Chapter 1.5 describes each of the four modes.

1.5.1 COMMAND MODE

You are in Command Mode when you enter PMATE. In Command Mode, you type file operations, program operations, and editing commands on the command line. You execute these commands by typing $$. To correct mistakes on the command line, press the Backspace key to erase the last character(s) typed, or type ALT-C to clear the entire command line. Type ALT-X to return to Command Mode from Insert Mode or Overtype Mode.

1.5.2 INSERT AND OVERTYPE MODES

Use Insert Mode or Overtype Mode to enter text in the text area. When you are in Insert Mode, you enter text at the cursor. Existing text moves to the right or down. Type ALT-N to enter Insert Mode. While you are in Insert Mode, PMATE displays INSERT MODE on the command line.

In Overtype Mode, text you enter replaces text at the cursor. Existing text disappears. Type ALT-V to enter Overtype Mode. While you are in Overtype Mode, PMATE displays OVERTYPE MODE on the command line.

FORMAT MODE 1.5.3

Format Mode is off when you enter PMATE. You turn Format Mode on to enable PMATE's text-formatting features (see Chapter 3). While turning Format Mode on, you can also change the maximum line length. For example, you can change the line length from 250 columns to 80 columns. (The width of the screen is 80 columns.)

You turn Format Mode on by typing the command F (Format) in Command Mode. To change the maximum line length, precede the F with the column number of the new right margin. To change the maximum line length to 80 columns, for example, type the command 79F on the command line. Execute the command by typing $$. The command line should look like this:

```
79F$$
```

When Format Mode is on, PMATE displays < in the text area wherever you have entered a carriage return.

While Format Mode is on, you can change the line length again by repeating the F command with a new column number.

To turn Format Mode off, type the F command without specifying a column number. The command line should look like this:

```
F$$
```


When you turn Format Mode off, the carriage return symbols (<) disappear.

1.6 PMATE COMMANDS

PMATE has three kinds of commands:

- Command-line commands
- Instant commands
- Control-line commands

Chapter 1.6 explains how to use command-line and instant commands. (Chapter 3 explains how to use control-line commands.)

Command-line commands are the commands you execute on the command line, in Command Mode. Most PMATE commands are command-line commands, which do simple or complex file operations, program operations, and editing.

After you execute a command-line command, the command and $$ remain on the command line. You can re-execute such a command by typing $ again. The additional $ doesn't appear on the command line.

You execute instant commands either with special function keys or in conjunction with the ALT key. You can execute an instant command in any mode. Almost every instant command duplicates a command-line command. PMATE has these instant commands so you can do simple editing without switching to Command Mode.

In this manual, the term "command" refers to a command-line command. An instant command is referred to by name or is preceded by ALT.

Note: To stop execution of a command-line or instant command, type ALT-C.

EDITING TEXT WITH INSTANT COMMANDS 1.7

Once you are familiar with PMATE's modes, you should be able to enter text in the text area. Use the instant commands in Table 1-1 to edit the text you create. Use these instant commands to move the cursor, scroll the screen, delete and recover text, move text, and change the case of characters.

When using the cursor-movement commands in Table 1-1, note that cursor movement in Overtype Mode is different from cursor movement in Insert and Command Modes. In Overtype Mode, the down-arrow and the up-arrow move vertically in whatever column you put the cursor. In Insert and Command Modes, the down-arrow and the up-arrow move only in column 0 of your screen. In Overtype Mode, the left-arrow stops at column 0 of your screen and the right-arrow stops at column 249. In Insert and Command Modes, the left-arrow moves to the end of the previous line when it reaches column 0, and the right-arrow moves to the beginning of the next line when it reaches the end of the line.

Table 1-1: Instant Commands

COMMAND	DEFINITION
CURSOR MOVEMENT	
ALT-B or ↓	Moves the cursor down one line.
ALT-Y or ↑	Moves the cursor up one line.
ALT-G or ←	Moves the cursor to the left.
→	Moves the cursor to the right.
ALT-U	Moves the cursor up six lines.
ALT-J	Moves the cursor down six lines.
ALT-P	Moves the cursor to the beginning of the next word.
ALT-O	Moves the cursor to the beginning of the current or preceding word.
ALT-FM	Moves the cursor to the beginning of the line.
ALT-A	Moves the cursor to the beginning of text, or moves the cursor to the end of text if it is already at the beginning.
SCROLLING	
ALT-FG	Scrolls the display left one column.
ALT-FH	Scrolls the display right one column.
ALT-FY	Scrolls the display up one line.
ALT-FB	Scrolls the display down one line.
DELETING AND RECOVERING TEXT	
Backspace or ALT-H	Deletes the character preceding the cursor.
ALT-D	Deletes the character at the cursor and moves the text to the right of the cursor one space to the left.
ALT-K	Deletes the current line, beginning at the cursor.
ALT-W	Deletes text up to the next word.
ALT-Q	Deletes the word preceding the cursor.
DEL	Deletes the character at the cursor and moves the cursor to the next character.
ALT-R	Recovers the last text deleted and puts it at the cursor location.

COMMAND	DEFINITION
MOVING BLOCKS OF TEXT	
ALT-T	Tags the beginning of a block of text to be moved.
ALT-E	Moves the text between the tag and the cursor into the special buffer.
ALT-Z	Inserts the contents of the special buffer at the cursor location.
ALT-FT	Toggles the cursor between the tag and the current cursor location.
MISCELLANEOUS INSTANT COMMANDS	
ALT-L	Inserts a line below the cursor.
ALT-M	Inserts a line, and moves the cursor to the beginning of the new line.
ALT-FC	Changes the case of the character at the cursor, and advances the cursor one character position.
ALT-FS	Toggles the default case of text between upper- and lowercase letters.
ALT-S x	Enters character x in text four times. (ALT-SS x enters x in text 16 times.)
ALT-S n < command >	Executes < command > n times.
SWITCHING FROM ONE MODE TO ANOTHER	
ALT-X	Enters Command Mode.
ALT-V	Enters Overtype Mode.
ALT-N	Enters Insert Mode.

1.8 FILE AND PROGRAM OPERATIONS

Chapter 1.8 shows you how to print text, write a file to disk, read a file from the disk, and leave PMATE.

1

1.8.1 PRINTING TEXT

To print your text, enter Command Mode by typing ALT-X. If the command line isn't empty, type ALT-C to clear it. Then, move the cursor to the beginning of the text by typing ALT-A.

Use the XT (Type) command to print text. To print the entire file, type:

```
XT$$
```

on the command line. PMATE prints the text exactly as it appears on the screen.

To print a specific number of lines instead of the entire text file, precede the XT command with the number of lines you want to print. For example, to print 25 lines of text, type:

```
25XT$$
```

on the command line. (Chapter 3 shows you how to print text with page breaks.)

WRITING A FILE 1.8.2

To write your text file to the disk, enter Command Mode by typing ALT-X. Then, on the command line, type the XE (End) command:

```
XE$$
```


PMATE writes your text file to the disk, clears the text area, and removes the filename from the status line.

READING A FILE 1.8.3

To read a text file from the disk, enter Command Mode by typing ALT-X. Then, type the command XF, followed by the filename. To read TEXTFILE from the disk, your command-line entries should look like this:

```
XFtextfile$$
```

When you read a text file from the disk, the file appears in the text area, and the status line displays two filenames. The first is the name of the file read from the disk. The second is the name of the file to be written to disk. (Chapter 3 explains the .$$$ filename extension.)

LEAVING PMATE 1.8.4

After you write your file to the disk, you can return to the operating system by typing the XH (Home) command:

```
XH$$
```

2

COMMAND-LINE COMMANDS

Instant commands offer flexibility in text editing by letting you edit text without switching from one mode to another. Command-line commands offer a different kind of flexibility. With command-line commands, you can:

- Specify a numeric argument to tell PMATE how many times (and in which direction) to perform a task.
- Specify a string argument to insert, search for, replace, or change text.
- Link commands together to be executed as a single command.

This chapter introduces you to the command-line commands used for text editing and elaborates on the use of command-line commands for reading, writing, and printing. Before presenting these commands, this chapter explains how to use numeric arguments and string arguments. (Chapter 3 shows you how to link commands together.)

NUMERIC AND STRING ARGUMENTS 2.1

NUMERIC ARGUMENTS 2.1.1

You use a numeric argument in front of most command-line commands to tell PMATE how many times (and in which direction) to perform a task. A numeric argument can be an integer from −32768 to 32767. In most cases, the default numeric argument is 1; if you don't specify a numeric argument, PMATE does the task once, moving forward. (The XT command is an exception; 1XT causes PMATE to print 1 line of text and XT causes PMATE to print the entire text file.)

A minus sign (−) as the numeric argument means − 1; if you use a minus sign as the argument, PMATE does the task once, moving backward.

The L command moves the cursor from line to line. It is used here to show you how to use numeric arguments:

15L	Moves the cursor forward 15 lines.
− 15L	Moves the cursor backward 15 lines.
0L	Moves the cursor to the beginning of the line.
L	Moves the cursor forward 1 line.
− L	Moves the cursor backward 1 line.

The # symbol is a numeric argument that represents the "entire quantity." Use it when you don't want to count how many lines you are operating on. For example, if you are moving a block of text, tag the beginning of text with the instant command ALT-T or with the command-line command T. Then, move the cursor to the end of text to be moved. The command #BM moves the block of text into the special buffer. The command BG inserts the block back into text at the cursor.

2.1.2 STRING ARGUMENTS

String arguments usually follow commands such as I (Insert), S (Search), R (Replace) and C (Change). Follow commands I, S, and R with one string. For example, to search for the word HELLO, type the command SHELLO. Follow the command C with two strings. Separate the two strings with $. For example, the command Ccat$dog changes the first occurrence of cat to dog.

COMMAND DESCRIPTIONS 2.2

Use the command-line commands in Table 2-1 to do basic editing and file operations. (Some of these commands—such as XF, XE, and XT—were introduced in Chapter 1.) Use numeric and string arguments with these commands, where appropriate. Execute a command by typing $$ and re-execute a command by typing $. (The additional $ doesn't appear on the screen.)

Table 2-1: Basic Command-Line Commands

COMMAND	DEFINITION
CURSOR MOVEMENT	
A	Moves the cursor to the beginning of the text.
Z	Moves the cursor to the end of the text.
M	Moves the cursor 1 position.
L	Moves the cursor to the beginning of the next line.
P	Moves the cursor to the beginning of the next paragraph.
W	Moves the cursor to the beginning of the next word.
DELETING TEXT	
D	Deletes the character at the cursor.
K	Deletes the line beginning at the cursor.
INSERTING TEXT	
I	Inserts what follows at the cursor position (IHELLO inserts the word HELLO before the cursor).
REPLACING TEXT	
R	Replaces the text at the cursor with what follows (RHELLO overwrites the five characters at the cursor with HELLO).

COMMAND	DEFINITION
SEARCHING FOR TEXT	
S	Searches for the following string, putting the cursor after the occurrence of the string (SHELLO places the cursor after the next occurrence of HELLO).
−S	Searches backward for the following string, putting the cursor at the beginning of the string.
CHANGING TEXT	
C	Changes the first occurrence of one string to another string (CCAT$KITTEN changes the next occurrence of CAT to KITTEN).
−C	Changes the previous occurrence of one string to another string.
MOVING BLOCKS OF TEXT	
nBC	Beginning at the cursor, copies n lines of text into the special buffer, overwriting text already in the special buffer.
−nBC	Moving backward from the cursor, copies n lines of text into the special buffer, putting the cursor at the beginning of the lines that have been copied. Text already in the special buffer is overwritten.
nBM	Beginning at the cursor, moves n lines of text into the special buffer, overwriting whatever is already in the special buffer.
−nBM	Backward from the cursor, moves n lines of text into the special buffer, overwriting text already in the special buffer.
BG	Gets the contents of the special buffer and inserts it at the cursor position.
T	Tags the beginning of the text to be moved.
Q#	Toggles the cursor between the tag and the current cursor position.
SWITCHING MODES	
0N	Takes you into Command Mode.
2N	Takes you into Overtype Mode.
N	Takes you into Insert Mode (any number preceding N, other than 0 or 2, takes you into this mode).

COMMAND	DEFINITION
FILE OPERATIONS	
XE	Writes the text file to the disk, clearing the text area.
XK	Clears ("kills") the text area without writing the contents to the disk.
XF	Reads the following file from the disk into the text area or declares the name for the file you are about to create. (XFtextfile reads a file called TEXTFILE into the text area. If TEXTFILE doesn't already exist, this command declares TEXTFILE as the name of the file you will create.)
XH	Exits from PMATE, returning to the operating system.
PRINTING TEXT	
XT	Prints the entire text file.
nXT	Prints n lines of text from the cursor forward.

Caution: Instant command ALT-C stops execution of the XT command. When you execute ALT-C while printing, however, you might leave PMATE and return to the operating system.

3

THE MAIN INGREDIENTS

This chapter covers the rest of the PMATE features you should know for text editing. These features include:

- The garbage stack
- Buffers
- Auto Buffer and Manual Modes
- Global commands
- Directory maintenance
- Command strings

This chapter also elaborates on features introduced in Chapters 1 and 2 such as text formatting and file input and output.

THE GARBAGE STACK 3.1

When you delete text, PMATE dumps it onto the **garbage stack**. The garbage stack is stored in a small block of memory space reserved by PMATE for deleted text. The garbage stack can also take up any space in memory not filled by text.

You can recover deleted text from the garbage stack with the instant command ALT-R. This command "pops" the last text deleted off the garbage stack and displays it on the screen again. If you delete four lines of text by typing ALT-KKKK, you can recover the text by typing ALT-RRRR. If you delete an entire file with the XK command, you can recover the file by typing ALT-R. (If the file is larger than the garbage stack, however, you might not be able to recover the entire file.)

You can also use the garbage stack to hold text you want to move. Delete text with the ALT-K or ALT-D command. Then, move the cursor to a new location and use ALT-R to insert the text by popping the garbage stack.

3.2 BUFFERS

PMATE has 11 **edit buffers**, which are areas in memory where text is stored. The main edit buffer is buffer T. When PMATE reads your text file from the disk, it puts the text file in buffer T. At the same time, the text area displays the text and the status line displays BUF = T.

The other buffers, buffers 0–9, are useful for storing and editing text to be moved from or inserted into your text file. When you edit text in buffers 0–9, the status line indicates which buffer you are editing. For example, if you are editing text in buffer 1, the status line displays BUF = 1.

Buffer 0 is the special buffer that PMATE uses to store text to be moved. When you use the command nBC, PMATE stores n lines of text in buffer 0. The command BG moves the contents of buffer 0 back into the edit buffer. The command B0E lets you display the contents of buffer 0 for editing. While you are editing this special buffer, the status line displays BUF = 0. Resume editing buffer T with the command BTE.

Buffer 0 is special because it is the default buffer. PMATE copies or moves text from your edit buffer into buffer 0 unless you specify another buffer.

Buffers 1–9 are like buffer 0 except PMATE moves or copies text into these buffers only when you tell PMATE which buffer to use. The command nB3C copies n lines of text into buffer 3. B3G copies the contents of buffer 3 into the edit buffer, at the cursor. B3E lets you edit buffer 3.

Table 3-1 is a summary of the edit buffer commands. While using these edit buffer commands, you should note these guidelines:

- You can move or copy text into or out of a buffer while editing any other buffer. You can also delete text from a buffer while editing any other buffer. If you delete the contents of buffer T, however, you'll lose your text file.
- You must be editing buffer T to read a file from the disk with the XF command or write your text file to the disk with the XE command. You can, however, read a file from the disk to buffers 0–9 with the XI command or write a file from buffers 0–9 with the XO command. (See Chapter 3.3.2.)
- The contents of buffers 0–9 remain in memory when you write the contents of buffer T to the disk with the XE command or delete the contents of buffer T with the XK command. The contents of buffers 0–9 are erased when you leave PMATE to return to the operating system.
- The size of any buffer depends on the amount of memory space available. If the text file in buffer T is large, less space is available for the other buffers. More space becomes available for one buffer when you delete text from another. If you fill up all memory space, PMATE displays the message MEMORY SPACE EXHAUSTED. Delete text or write it to disk to continue.
- You can store text or command strings in buffers 0–9. The command .3 executes a command string stored in buffer 3. (See Chapter 3.5 for information on command strings.)

Table 3-1: Edit Buffer Commands

COMMAND	DEFINITION
BxK	Deletes ("kills") the contents of buffer x when you are editing in any other buffer.
BxE	Displays buffer x for editing.
nBxC	Copies n lines from the edit buffer to buffer x. The cursor in each buffer is left at the end of the text that is copied. This command destroys the previous contents of buffer x.
nBxD	Copies n lines from the edit buffer to buffer x. These lines are inserted before the cursor position in buffer x. The cursor in the edit buffer is left at the end of the lines that have been copied.
nBxM	Moves n lines from the edit buffer to buffer x. Buffer x's cursor is left at the end of the text. This command deletes the previous contents of buffer x, and deletes these n lines from the edit buffer.
nBxN	Moves n lines from the edit buffer to buffer x. The lines are inserted in buffer x just before the cursor. The lines are deleted from the edit buffer.
BxG	Copies ("gets") contents of buffer x into the edit buffer just before the cursor. The contents of buffer x are not affected.

NOTE: x is equal to T or a number 0–9; the "edit buffer" is the buffer you are editing.

3.3 FILE OPERATIONS

PMATE's file operations consist of reading input files and writing output files. When PMATE reads a file from disk, the file is an input file. When PMATE writes a file to disk, the file is an output file. Thus, a text file can be an input file or an output file, depending on whether you are reading it from or writing it to disk.

When you tell PMATE the name of the input or output file, you **declare** the filename. PMATE offers several options for declaring a filename as well as several options for reading and writing a file.

Chapter 3.3 shows you:

- The ways in which you can declare a filename.
- The various commands for reading and writing files.
- How to modify the way in which PMATE reads and writes files.
- How to list your directory of files and delete files without returning to the operating system.

DECLARING FILENAMES 3.3.1

As demonstrated in Chapter 1, you can declare a filename as you enter PMATE or while you are in PMATE. To declare a filename as you enter PMATE, you type:

pmate < filename > (cr)

To declare a filename while you are in PMATE, on the command line you type:

XF < filename > $$

In both cases, the command syntax is the same for a new filename and an existing filename. In both cases, however, PMATE treats a new file differently from an existing file.

PMATE treats the new file as an output file. The status line displays an asterisk (to indicate that there is no input file), followed by the output filename.

PMATE treats an existing file as an input file. Because there is no output file, PMATE creates one. The status line displays the input filename, followed by the output filename. They are the same, except the output filename has the filename extension .$$$.

When you write the file to the disk, PMATE renames both the input filename and the output filename. Suppose the input filename is TEXTFILE.1. PMATE creates an output file called TEXTFILE.$$$. When you write the file to disk, PMATE stores TEXTFILE.1 on the disk as TEXTFILE.BAK and stores TEXTFILE.$$$ as TEXTFILE.1. Next time you read the file from disk, you'll read the most recent version of the file. The previous version of the file is the backup file.

Another option is to declare different input and output filenames. For example, you can declare two filenames, FILEONE and FILETWO, when you enter PMATE or while you are in PMATE. From the operating system, type:


```
pmate fileone filetwo(cr)
```

or, on the command line type:

```
XFfileone filetwo$$
```

The first file is your input file and the second file is your output file. The status line displays FILEONE,FILETWO. When you write the file to the disk, the filenames remain the same.

You could declare two filenames to modify a file and keep both versions, each for a different purpose. Also, you might declare a different drive name for the output file if your input file is larger than half of your disk space.

3.3.2 FILE INPUT AND OUTPUT

Until now you have used the XF and XE commands to read and write files. PMATE has many other commands for file input and output. Table 3-2 is a summary of all the file input and output commands.

Table 3-2: File Input and Output Commands

COMMAND	DEFINITION
XF < filename >	Declares a new (output) filename or declares an existing (output) filename. To declare two filenames, separate them with $.
XE	Writes the text file to disk, clearing the text area.
XE < newname >	Writes the file to disk, renaming the output file and leaving the input file undisturbed.
XJ	Writes the file to disk and reads the file into memory again. This command is equivalent to using the XE command and then the XF command. Use this command frequently to write your data to disk and then resume editing (in case of a power failure).
XJ < newname >	Writes the file to disk using a new filename, and then reads the file back into memory using the new filename. The original input file is not disturbed.
XC	Closes the input and output files as they are (on disk). The contents of the edit buffers are deleted, and the output file keeps the .$$$ filename extension.
XK	Deletes the output file and the contents of the edit buffer, if you are editing in buffer T, leaving the input file undisturbed. Deletes the contents of the edit buffer if you are editing in buffers 0–9.
XI < filename >	Reads (in) a copy of < filename > from disk into the text buffer just before the cursor. < filename > on the disk is unchanged, and currently declared input and output filenames remain the same. You can use this command to merge files, load macros into buffers, and store on disk what you might store in buffers. (See Chapter 3.5.4.)
nXI < filename >	Reads (in) n lines of < filename > from disk into the text buffer just before the cursor without disturbing currently declared input and output files.
XO < newname >	Copies the file in the edit buffer to < newname > . This lets you write (out) the file to the disk under another name without disturbing the currently declared input and output filenames.
nXO < newname >	Copies n lines of the text buffer to < newname > . This lets you write (out) text to disk under another name without disturbing the currently declared input and output filenames.

3.3.3 AUTO BUFFER AND MANUAL MODES

PMATE normally operates in **Auto Buffer Mode**. Auto Buffer Mode lets you edit a file that is larger than PMATE's memory space. When PMATE reads such a large file from disk, it reads in only as much as will fit in memory. Then, as you move the cursor through the file, PMATE writes some of the text into the output file and reads in more of the text from the input file. Auto Buffer Mode works only when you are editing text in buffer T. You can edit a file as large as 512K in Auto Buffer Mode.

As you edit a file that is larger than memory, cursor movement commands, such as L, M, P, and W, move through the entire text file. Other commands, such as A, Z, and S, operate only on text in memory, even though Auto Buffer Mode is on. These commands move the cursor or search for a word in a "limited workspace" so that you can move short distances while editing a large file.

PMATE has **global commands** for moving and searching through an entire large text file. Global commands begin with U (Universal). The US command is the global command that searches through an entire text file for a word or string. The UZ command moves the cursor to the end of a large text file. The UA command scrolls backward to the beginning of a large text file. If you are near the end of a long file and you want to go to the beginning of text, however, XJ gets you there faster than UA. XJ writes the remainder of the file to disk and reopens the file at the beginning.

You can turn Auto Buffer Mode off with the 0QU command. This puts PMATE in **Manual Mode**, and PMATE no longer automatically writes and reads text to and from the disk. Instead, text is divided into pages.

You can specify the number of lines in a page with the QP command. For example, 50QP divides text into pages of 50 lines each. Or, you can divide pages with form-feed characters. You enter a form-feed character in text with the command ALT-7 L. It appears on the screen as L.

Once you specify the number of lines in a page, you can read or write text with Manual Mode commands, shown in Table 3-3.

Table 3-3: Manual Mode Commands

COMMAND	DEFINITION
nXA	Reads n pages of text from the input file into the edit buffer, appending the new pages to the end of the text already in the buffer.
− nXA	Reads n pages of text from the output file to the beginning of text in the edit buffer.
nXW	Writes n pages of text from the beginning of text in the edit buffer to the output file.
− nXW	Writes n pages of text from the end of text in the edit buffer back to the input file.
nXR	Writes n pages of text to the output file and reads n pages of text from the input file into the edit buffer.

To stop PMATE from reading or writing pages at a specific point in the text file, put a form-feed character at that point in the text.

When you are in Manual Mode, the numeric argument preceding the XI and XO commands represents the number of pages instead of a number of lines. For example, the command 3XI < filename > reads in 3 pages from < filename >. Repeat the XI command without the filename to read in more pages. PMATE assumes you are reading from the same file unless you specify a new filename.

3.3.4 DIRECTORY MAINTENANCE

You can look at any of your file directories while you are in PMATE. To list a directory, PMATE inserts a copy of the directory in the current edit buffer at the cursor. After you look at the directory you can delete it just as you delete any text. If you don't want to insert the directory in your text file, display a different edit buffer before listing the directory. The command to list a directory is XL. Table 3-4 shows the options for using the XL command.

Table 3-4: Directory Listing Commands

COMMAND	DEFINITION
XL	Lists all files in the current working directory of the default drive.
XL <d>:*.*	Lists all files in the current working directory of drive <d>.
XL <pthnm>*.*	Lists all files in subdirectory <pthnm> of the default drive.
XL<d>:<pthnm>*.*	Lists all files in subdirectory <pthnm> of drive <d>.

You can delete files in your directory while you are in PMATE. This feature is useful if you get a DISK FULL message when trying to write a file to disk. You can delete unwanted files from the disk to make room.

The directory you read into the edit buffer isn't updated after you delete files because it is only a copy. Read in the directory again to see if the files have been deleted. Wild-card characters such as * and ? don't work when you are deleting files.

The file delete command is XX. Here are the options for using the XX command:

- **XX < txtfl >**
  Deletes file < txtfl > from current working directory of default drive.
- **tXX < pthnm > < txtfl >**
  Deletes file < txtfl > from subdirectory < pthnm > of default drive.
- **XX < d > : < pthnm > < txtfl >**
  Deletes file < txtfl > from subdirectory < pthnm > of drive < d >.

You can change the default drive in PMATE before you have declared an input or output file. Use this feature to reset the disk system after you have taken a diskette out of a drive and replaced it with another. The command to change the default drive is XS. Thus you can change the default drive to drive B by typing:

```
XSb:$$
```

The status line displays B:, and PMATE resets the disk system if you have changed the diskette in that drive.

You can also change the subdirectory (pathname) in PMATE before you have declared input and output files. The command to change the subdirectory is XP (pathname). You can change to a subdirectory called \DATAFILES by typing:

```
XP\datafiles$$
```

3.4 FORMATTING A TEXT FILE

Chapter 3.4 explains how to format text for printing. It shows you commands that temporarily format text with tab stops and commands that indent text. These commands aren't incorporated into the file when you write the file to disk and are in effect only until you exit PMATE.

Chapter 3.4 also shows you how to embed permanent formatting commands in text. The commands are part of a nonprinting **control line**, which formats text with margins, tabs stops, and indentation. The control line is part of the file you write to disk, so the text is always formatted.

3.4.1 SETTING TAB STOPS

PMATE uses tab stops to indent text and to format multiple-column text. You can set a maximum of fifteen tab stops. Default tab stops are every eight columns. Use the commands in Table 3-5 to set tab stops.

Table 3-5: Tab Stop Commands

COMMAND	DESCRIPTION
YK	Deletes all tab stops (each space becomes a tab stop).
nYS	Sets a tab stop at column n.
nYD	Deletes the tab stop at column n.
nYE	Deletes all old tab stops and sets new ones every nth column.
Use the following commands to change tab settings without altering the current position of the text:	
nYF	Replaces all tabs with the appropriate number of spaces in the next n lines.
nYR	Replaces blocks of spaces with tabs (where possible) in the next n lines.

INDENTING TEXT 3.4.2

To indent a single line of text, use the Space bar to move the required number of spaces. To indent more than one line of text, use one of these two methods:

- In Command Mode, type the command YI, preceded by the column number of the tab stop where the text should begin. Thus, if column 5 is a tab stop, you can set the indentation to column 5 by typing 5YI.

  Once you set the indentation, you can indent an entire paragraph by indenting the first line of the paragraph. Do this by putting the cursor at the beginning of the first line. Then, press the Tab key. The entire paragraph moves to the right and begins in column 5.

  To indent part of a paragraph, put the cursor at the beginning of the first line to be indented. Then, press the Tab key. PMATE indents the line where the cursor is located, as well as subsequent lines in the paragraph.

- In Command Mode, Insert Mode, or Overtype Mode, you can indent an entire paragraph. Put the cursor at the first tab stop on the first line of the paragraph. Type the instant command ALT-FI to set the indentation to that tab stop. Then press the Tab key. The entire paragraph moves to the right and begins at the first tab stop.

  If you type the instant command ALT-FP, PMATE moves the indented text and the cursor four columns to the right. Instant command ALT-FO moves the indented text and cursor four columns back to the left.

3.4.3 CONTROL LINES

A control line formats text when Format Mode is on. You embed a control line in text before the block of text to be formatted. Begin the control line with F, which you enter in text by typing ALT-7 F in Insert or Overtype Mode. Separate each command in the control line with a semicolon (;), and end the control line with a carriage return.

Table 3-6 contains the control-line commands that format text with margins, tab stops, and indentation.

Table 3-6: Control-Line Commands

COMMAND	DESCRIPTION
Ln	Sets the left margin to column n.
Rn	Sets the right margin to column n.
K	Deletes all tab stops.
Sn	Sets a tab stop at column n.
Dn	Deletes the tab stop at column n.
En	Deletes old tab stops and sets new tab stops every nth column.
In	Sets indentation to column n if n is a tab stop. Tabbing to this column indents all subsequent lines in the paragraph.

Here is an example of a control line that sets margins:

```
FL5;R60<
```

This control line formats the text that follows with the left margin at column 5 and the right margin at column 60. Without this control line, the left margin is 0. The right margin is the column you specify when you turn Format Mode on. (If you don't specify a right margin, the right margin is 249.)

The right-margin column number in the control line should be less than or equal to the column number you specify when you turn Format Mode on. If it isn't, re-enter Format Mode, specifying a larger column number.

Here is an example of a control line that sets margins, tab stops and indentation:

```
FL5;R60;K;S10;I10<
```


This control line formats the text that follows with a left margin of 5 and a right margin of 60. The K command deletes all tab stops, the S10 command makes column 10 a tab stop, and the I10 command sets the indentation to column 10.

If you omit a formatting command from the control line, PMATE uses the default setting to format text. For example, if you omit the left-margin command from the control line, PMATE formats text with a left margin of 0.

PMATE interprets the control line to format text. When printing a specific number of lines, PMATE counts the control line even though it doesn't print it. For example, suppose the control line is on line 0 of your text file. If you tell PMATE to print 10 lines of text, it prints lines 1 through 9. You can embed up to thirty control lines in a text file.

3.5 LINKING COMMANDS

PMATE lets you link commands together to form a **command string**. A command string is a sequence of commands executed on the command line at the same time. Chapter 3.5 shows you how to:

- Form and execute command strings
- Edit a command string on the command line
- Repeat command strings
- Store command strings in a buffer and execute the strings from the buffer

3.5.1 COMMAND STRINGS

To form a command string, type commands on the command line, each separated by $. Execute the string with $$, just as you execute an individual command. Then PMATE performs the tasks, in order of occurrence, without stopping.

Here is an example of a command string that moves the cursor to the top of the file and then deletes 5 lines:

```
A$5K$$
```

If you discover an error in a command string, you can edit the command line. Type the instant command ALT-_, which puts the command string into the text area. Then, edit the command string in Insert or Command Mode. Finally, return to Command Mode and type ALT-_ again to put the edited command string back on the command line.

REPEATING A COMMAND STRING 3.5.2

Like an individual command, a command string remains on the command line after you execute it. You can re-execute the command string by typing $ again.

To repeat a command string several times, use PMATE's **iteration** feature. To use iteration, enclose the command string in brackets [ ], and precede the bracketed string with a number that indicates how many times you want to execute the command string. Inside the brackets, terminate the last command in the string with $. To execute the command string, follow the bracketed string with $$.

Here is an example of a command string that uses iteration:

```
2[A$IHELLO$]$$
```

This command string moves the cursor to the beginning of the text, inserts HELLO, moves the cursor to the beginning of text, and inserts HELLO again. If you don't use a numeric argument in front of the brackets, PMATE executes the command string as long as it can. In this case, PMATE executes the command until it runs out of memory space.

USING A COMMAND STRING TO PRINT TEXT 3.5.3

You can use a command string with iteration to print text with page breaks. The command string uses the command 12QT, which causes a page break in text. To print 10 pages of text, 50 lines each, type the command string:

```
10[50XT$12QT]$$
```

3.5.4 EXECUTING COMMAND STRINGS FROM BUFFERS

You can store a command string in another buffer if you want to use it more than once. Go into one of the buffers 1 through 9. Type the command string in the text area, in Insert or Overtype Mode. Then, return to the edit buffer.

To execute the command string from the buffer, type the buffer number, preceded by a period, on the command line. Execute this command with $$. For example, if you are storing a command string in buffer 2, execute the command string by typing .2$$ on the command line.

Command strings can be **nested,** which means you can execute a command string in one buffer that executes a command string in another buffer.

When you leave PMATE, a command string stored in a buffer disappears. To save a command string, store it in a text file on disk. After you enter PMATE, use the XI command to read the file into one of the buffers 1 through 9. Then, execute the command string from the buffer by typing the buffer number, preceded by a period.

4

MACROS

This chapter shows you how to create, store, and execute macros. Then the chapter presents the advanced text-editing features that you might use in macros. These features include:

- The advanced use of numeric arguments
- The ARG (argument) indicator
- Input and output radixes
- String arguments
- Iteration, branching, and exiting from a macro
- Tracing macro errors
- Processing input from the keyboard
- Inserting comments in a macro
- Q commands used in macros

INTRODUCTION 4.1

A **macro** is a sequence of commands executed as a single command. A command string stored in a buffer is an example of a simple macro. A macro can also resemble a small program. It can be several lines long and it can contain complicated editing commands.

When a macro is stored in a buffer, you execute it by typing a period and then the buffer number. You store the buffer contents in a file on disk to save such a macro to use the next time you enter PMATE (see Chapter 3.5.4).

Macros you use often can be stored as a macro file in PMATE's **permanent macro area**. You execute a permanent macro by typing the macro name, preceded by a period. (A permanent-macro name can be any single character except 0–9.)

After you create the macro file it can be incorporated as a permanent part of PMATE by making a new copy of the PMATE program.

4.1.1 CREATING A PERMANENT MACRO FILE

To create a permanent macro file, make sure you have no input or output files declared. Empty your edit buffer of text. Then, read the contents of your permanent macro area into the text area with the command QMG (Macros Get). Nothing appears on the screen unless you already have a macro file in your permanent macro area.

Follow these guidelines when creating a permanent macro file:

- Begin each macro with control character X. (Enter this character by typing ALT-7 X.)
- Type the macro name after the X.
- End each macro in the file with $.
- End your permanent macro file with X.

This is an example of macro file format:

```
Xa A$IHELLO WORLD$
Xb A$C $*$
X
```

The first macro in the file is macro a. It puts the cursor at the beginning of the file and inserts HELLO WORLD in text. The $ at the end of the macro separates macro a from the next macro. Macro b moves the cursor to the beginning of text and changes the first occurrence of a space to an asterisk. The $ at the end of the macro separates macro b from the X, which marks the end of the macro file. Without the $ at

the end of macro b, PMATE thinks the X is part of macro b.

After you create a macro file, send it to the permanent macro area with the command QMC (Macros Change). The permanent macro file you have created overwrites any previous file in the permanent macro area. After you send the macro file to the permanent macro area, delete the macros from the screen with the XK command.

Once you send your macro file to the permanent macro area, you can execute any macro in the file. After you send the macros shown above to the permanent macro area, you can insert HELLO WORLD in text by executing the command .a. You can change the first occurrence of a space to an asterisk by executing the command .b.

SAVING PMATE WITH A PERMANENT MACRO FILE 4.1.2

To save the permanent macro file, make a new copy of PMATE with the command XD (Duplicate). Follow XD with a new filename. To make a copy of PMATE called PMATE02, type this on the command line:

```
XDpmate02$$
```

This command creates a new version of PMATE, PMATE02.COM, which contains your permanent macros. (The filename extension .COM is automatically appended when you use the XD command.) To enter the new version of PMATE from the operating system, either type the new filename instead of PMATE, or rename the new PMATE from the operating system after deleting the old PMATE.

4.1.3 EXECUTING A MACRO WHEN ENTERING PMATE

Precede the first macro in the permanent macro area with I (Initialization) instead of X to execute the first macro every time you enter PMATE.

Precede the first macro with I and end the macro with the XH command to generate a program that edits a file and returns to the operating system without displaying anything on the screen.

Precede the first macro with S to execute the macro as you enter PMATE and process commands given directly from the operating system. Input and output filenames, however, are not opened until the macro has finished execution. Instead, any strings that follow "pmate", when you enter PMATE from the operating system, appear on the command line.

The strings on the command line can be used as string arguments during macro execution. Use the control character A to get a string argument from the command line and put the string in text. (See Chapter 4.4.) After the macro has finished execution, the first two strings typed from the operating system become your input and output filenames.

4.1.4 NESTING MACROS

Macros can call any other macro stored in the permanent macro area or in a buffer. Macros can be nested to a maximum depth of 15 levels.

NUMERIC ARGUMENTS 4.2

A numeric argument is defined as an integer from −32768 to 32767. Its integer value can also be expressed as an arithmetic or logical operation, a variable, or a function. In a macro, it is often useful to express a numeric argument in one of these forms. Chapter 4.2 explains how to use arithmetic and logical operations, variables, and functions as numeric arguments. It also shows you how to use a numeric argument with the Insert and Replace commands.

On the status line, the ARG indicator displays the value of numeric arguments expressed as arithmetic and logical operations, variables, and functions.

ARITHMETIC OPERATIONS 4.2.1

An example of an arithmetic operation is 5 + 4. You can use this expression, which has an integer value of 9, as a numeric argument. For example, you can type the command 5 + 4L to move the cursor forward 9 lines. When you execute this command, the status line displays ARG = 9.

PMATE performs arithmetic operations from left to right. Multiplication and division don't take precedence over addition and subtraction. Use parentheses to change the way PMATE performs an operation. For example, 5 + 3*2 has the value 16, but 5 + (3*2) has the value 11. You can use up to 15 levels of parentheses in an arithmetic operation.

Table 4-1 shows the arithmetic operators you can use in a numeric argument.

Table 4-1: Arithmetic Operators

OPERATOR	DEFINITION
+	Addition
−	Subtraction
*	Multiplication
/	Integer division

Note: You can display the remainder of the last division operation on the status line with the function @R.

4.2.2 LOGICAL OPERATIONS

The value of a logical operation is either true or false. The value of a numeric argument expressed as a true logical operation is −1. (The status line displays ARG = −1.) The value of a numeric argument expressed as a false logical operation is 0. (The status line displays ARG = 0.)

Table 4-2 lists the logical operators you can use in a numeric argument. Table 4-3 shows some examples of the use of logical operators.

Table 4-2: Logical Operators

OPERATOR	DEFINITION
=	Equal (true if the two operands are equal).
<	Less than (true if the first operand is less than the second operand).
>	Greater than (true if the first operand is greater than the second operand).
&	And (true if both operands are true).
!	Or (true if either operand is true).
'	Logical complement (gives the opposite value of an expression).

Table 4-3: Examples of Logical Operators

EXAMPLE	DESCRIPTION
3 < 2	False, has the value 0.
3 < 2'	True, has the value −1. (The expression is false but the ' gives it the opposite value.)
2 < 3	True, has the value −1.
2 < 3!(5 = 2)	True, has the value −1. (At least one of the operands is true.)
2 < 3&(5 = 2)	False, has the value 0. (Only one operand is true.)
5 + 3 = (1 + 7)	True, has the value −1. (The two operands are equal.)
5 + 3 = (1 + 7)'	False, has the value 0. (The two operands are equal, making the expression true, but the ' gives the expression the opposite value.)

VARIABLES 4.2.3

You can use ten numeric variables as numeric arguments (labeled 0–9). Set the value of a variable with the command V. For example, this command sets the value of variable 0 to 10:

```
10V0$$
```

To use the value of the variable as a numeric argument, precede the variable label with @. For example, if variable 0 is set to 10, you can delete 10 lines of text with the command:

```
@0K$$
```

The status line displays ARG = 10.

Use the command VA to add to the value of a variable. If the value of variable 0 is 10, the command:

```
3VA0$$
```

adds 3 to the value of variable 0. The value of variable 0 becomes 13.

During execution of a macro, you can put up to 20 variable values on the **number stack**, a reserved block of PMATE's memory. The last value put on the number stack is the first value "popped" off. You pop the values off the top of the stack. If you set the value of variable 0 to 10 by typing:

```
10V0$$
```

you can put the value 10 on the number stack by typing:

```
@0,$$
```

Pop the value 10 off the number stack for use elsewhere in the macro by typing:

```
@S$$
```

PMATE clears the number stack when the execution of the macro is completed.

A PMATE function is an operation that requires the input of an argument and returns a value based on that argument. (The value is displayed by the ARG indicator on the status line.) Arguments preceded by @, such as those in Chapter 4.2.3, are examples of functions. Table 4-4 is a summary of the PMATE functions, which you can use as numeric arguments.

Table 4-4: PMATE Functions

FUNCTION	DESCRIPTION
@n	Returns the value of variable n, where n is a digit 0–9.
@n,	Puts the value of variable n on the number stack.
@A	Returns the value of the numeric argument preceding the last macro call (this may be used to pass an argument to a macro).
@B	Returns the value of the current edit buffer (0 if buffer T, 1 if buffer 0, 2 if buffer 1,...,10 if buffer 9, 11 if editing the command line, also known as buffer C).
@C	Returns the value that represents the current location of the cursor. This value is the character-position number of the cursor location in the edit buffer. The first character position is 0.
@D	Returns the number of lines scrolled by instant commands ALT-U and ALT-J.
@E	Returns the value of the error flag.
@F < filename >	Returns the value − 1 if < filename > exists in the working directory. Returns the value 0 if < filename > is not in the working directory.
@G	Returns the length of the string argument just referenced by an I, S, R, or C command.
@H < string > $	Returns the value 0 if < string > matches the characters at the cursor, returns the value − 1 or 1 if there is no match.
@I	Returns the number of pages read from the input file. Pages are counted only if they are delimited by form-feed characters.

FUNCTION	DESCRIPTION
@J	Returns the number of lines available on the screen for text display (not counting the three lines at the top of the screen).
@K	Returns the ASCII value of the key entered after a G or QR command.
@L	Returns the line number of the cursor location. (The first line number is 0.) This returns the line number of the text file if Auto Buffer Mode is on. If Auto Buffer Mode is off or if you are not editing buffer T, this returns the line number of text in memory.
@M	Returns the amount (in bytes) of working memory space available.
@O	Returns the number of pages written to the output file. Pages are counted only if they are delimited by form-feed characters.
@P	Returns the value of the absolute memory address of the cursor location.
@Q	Returns the column number of the previous tab stop.
@R	Returns the remainder of the last arithmetic division performed.
@S	Returns the top number in the number stack and pops the number off the stack.
@T	Returns the ASCII value of the character at the cursor.
@U	Returns the value −1 if Auto Buffer Mode is on and 0 if Auto Buffer Mode is off.
@V	Returns the value of the current mode (0 if Command Mode, 1 if Insert Mode, 2 if Overtype Mode).
@W	Returns the column number of the right margin.
@Y	Returns the column number of the left margin.
@Z	Returns the column number of the next tab stop.
@@	Returns the value of the byte in memory pointed to by whatever value you assign to variable 9.
@/	Returns the column number of the current indentation setting.
"x	Returns the ASCII value of x, where x is any character.

INSERT AND REPLACE ARGUMENTS 4.2.5

You can insert a character in text by using its ASCII value as the numeric argument of the I command. For example, if you type 92I on the command line, PMATE inserts a backslash (\) in text at the cursor (if you are using a Standard keyboard). (Appendix C lists the ASCII characters and their values.)

You can replace a character in text by using its ASCII value as the argument of the R command. For example, if you type 65R on the command line, PMATE replaces the character at the cursor with the letter A.

You can use a numeric argument with the I command to insert the value of a variable or a value related to the variable, in text. To insert the value of the variable in text, precede the I command with @, the variable label, and a backslash (\). For example, if variable 0 has the value 20, the command @0\I inserts 20 in text. To insert the value of variable 0 in text, followed by a space and the value of variable 0 plus 5, type the command:

```
@0\I $@0+5\$$
```

The numbers 20 25 appear in text.

INPUT AND OUTPUT RADIXES 4.3

PMATE recognizes the numbers you enter on the command line as decimal numbers; base 10 is the default input radix (base). The default output radix is also base 10. Thus, both the values displayed by the ARG indicator and the operations performed on the text file are in base 10.

You can change the input and output radixes with the command QI. For example, if you execute the command 8QI, the input radix changes to base 8 (octal). You can then do conversions from octal to decimal. If you type 10 on the command line, the ARG indicator

displays 8. To change the input radix back to base 10, execute the command 12QI. The numeric argument for the QI command is 12 because 12 (octal) is equal to 10 (decimal).

If the input radix is base 10, you can change the output radix to base 8 with the command 8QO. If you type 8 (decimal) on the command line, the ARG indicator displays 10 and the line and column indicators display the cursor position in octal. Change the output radix back to decimal with the command 10QO.

Because you can change the input and output radixes, you can perform conversions between any two radixes. Many programmers need to do such conversions between decimal and hexadecimal (base 16). When the radix is greater than 10, however, some values are represented by letters and numbers.

When PMATE sees a letter in the command line, it recognizes the letter as a command unless it is preceded by a digit (0–9). PMATE interprets each succeeding character as a number (if the character can be interpreted as a number). If the input radix is hexadecimal, the command DDK deletes 2 characters and erases 1 line. The command 0DDK erases 221 lines since DD (hex) equals 221 (decimal). Use $ to terminate a hex character if the next character can also be interpreted as part of the hex number. The command 0D$D deletes 13 characters. 2$D deletes 2 characters. 2D is interpreted as 45 (decimal).

4.4 STRING ARGUMENTS

Two advanced editing features involve string arguments. The first lets you store a string argument in a buffer. You can then use the string argument when you execute an I, S, C, or R command. When a string argument is stored in a buffer, use control character A, followed by @ and the buffer number, as the command argument.

For example, if the word AUTOMOBILE is stored in buffer 1, type the command:

```
IA@1$$
```

to insert the word AUTOMOBILE in text, at the cursor. (The command IA@0 is the same as the BG command. It inserts the contents of the special buffer in text.)

The second feature, which you use with the I command, lets you store string arguments on the command line to insert in a block of text when you execute a macro. The text is contained in the macro, which gets the string arguments from the command line. You use the control character A and command QA, along with I in the macro, to call the strings.

If buffer 1 contains:

2QAIDear Mr. IAAI,
You, Mr. IAAI, have the opportunity to be the first on your block in beautiful IABI to own your own copy of an exciting new editor. Imagine what you and Mrs. IAAI can do with it. The rest of IABI will be so jealous...$

then the command:

```
.1Jones$Cambridge$$
```

inserts the following text:

```
Dear Mr. Jones,
      You, Mr. Jones, have the opportunity to be
the first on your block in beautiful Cambridge to
own your own copy of an exciting new editor.
Imagine what you and Mrs. Jones can do with it.
The rest of Cambridge will be so jealous...
```

You begin the macro with the command 2QA to tell the macro how many string arguments are stored on the command line. Without the QA command, the macro doesn't know how many string arguments to look for. Consequently, it tries to execute the J in Jones as a command.

The next command in the macro is the I command, which inserts the text that follows in the text area.

The command IAA inserts Jones, the first string stored on the command line. PMATE identifies the strings on the command line in alphabetical order. Thus, the first is string A and the command IAA copies string A from the command line and inserts it in text. After this command, the I inserts the following text up to another IAA command, which again inserts string A in text. Then, the I command again inserts the text that follows, up to the command IAB, which inserts string B (Cambridge), the second string on the command line.

As demonstrated in this example, you can insert the strings stored on the command line, over and over. You can store up to 26 strings (A–Z) on the command line. Remember to use the QA command to tell PMATE how many strings are stored on the command line.

When macros are nested, string arguments can be nested.

WILD-CARDS USED IN STRING ARGUMENTS 4.5

You can use wild-cards in nonspecific search strings. Uppercase characters in the search strings match only uppercase characters in text. Lowercase characters match either upper- or lowercase characters. Type ALT-7 with the letter to enter wild-card characters. Table 4-5 lists these wild-card characters.

Table 4-5: Search-String Wild-Card Characters

WILD-CARD	DESCRIPTION
E	Matches any character. (SMAEE matches MALE, MADE, and MATE.)
L	Matches the character that follows. This lets you search for a wild-card character. (SMALEE matches only MAEE.)
N	Matches anything but the character that follows. (SMANTE finds MALE or MADE, but not MATE.)
S	Matches a space or a tab.
W	Matches any character except a letter or number.

ITERATION AND BRANCHING 4.6

You can use PMATE's iteration feature with the numeric arguments presented in Chapter 4 to create complex macros consisting of if-then loops, and conditional and unconditional branching. Chapter 4.6 reviews iteration. Then it shows you how to use iteration in if-then loops and branching statements. It also shows you how to exit from a macro.

4.6.1 ITERATION

The syntax for iteration is as follows:

< num. arg. > [< command string > < opt. num. arg. >]

The numeric argument preceding the brackets [ ] tells PMATE how many times to execute the command string inside the brackets. If this numeric argument is missing, PMATE executes the command string 64,000 times, or until an error occurs. If the numeric argument is 0, PMATE ignores the command string inside the brackets. If the numeric argument is −1, PMATE executes the command string once. If the numeric argument is a logical operation, PMATE executes the command string if the operation is true, and ignores it if the operation is false.

If you use the optional numeric argument, PMATE stops iteration of the loop as soon as the optional numeric argument becomes nonzero (true). If you omit the optional numeric argument, PMATE stops iteration of the loop if the error flag is set (see Chapter 4.7.3).

Commands 5[D], 5D, and 5V0[D − VA0@0 = 0] all have the same effect. In the third command string, 5V0 sets the value of variable 0 to 5. Inside the iteration brackets, D deletes a character and −VA0 decrements the value of variable 0 by one. The optional numeric argument @0 = 0 is a logical operation. The loop continues until the value of variable 0 is 0, and the logical operation is true.

The command string [Chello$goodbye$] changes all occurrences of hello to goodbye. The command string [Chello$goodbye] changes the first occurrence of hello to goodbye]. PMATE interprets the bracket at the end of the iteration as part of the string, because the string is not terminated by $. And without the closing bracket, PMATE cannot continue the change command.

Iterations can be nested to a maximum depth of 15.

As mentioned in Chapter 4.6.1, when a command string within brackets is preceded by a logical operation, PMATE executes the command string if the operation is true and skips the command string if the operation is false. PMATE can also execute one command string if a logical operation is true and another command string if the operation is false. Follow the first command string within brackets by the second command string within brackets. Don't put spaces between the sets of command strings. Here is an example of such an if-then loop:

```
@0 < 3[Ihello$][Igoodbye$]$$
```

In this example, if the value of variable 0 is less than 3, PMATE inserts hello in text at the cursor. If the value of variable 0 is 3 or greater, PMATE inserts goodbye in text.

You can use the **next** and **break** commands inside iteration brackets to terminate execution of a loop. The next command is an argument followed by ^. If the argument is nonzero (true) or missing, PMATE goes to the next set of iteration brackets. If the argument is 0, PMATE continues executing the current command string. The break command is an argument followed by _. If the argument is nonzero or missing, PMATE exits from the iteration brackets. If the argument is 0, PMATE continues executing the command string.

You can use brackets [ ] or braces { } to enclose iteration loops except when using break and next commands. Next and break commands skip braces { } and move to the next bracket. Enclose if-then-else constructions in braces so that a next or break command exits the iteration loop, not just the clause.

Make sure you match every left bracket with a right bracket. Also, make sure you use $ to distinguish between brackets used as part of a search or insert string, and brackets used for iteration.

4.6.3 CONDITIONAL AND UNCONDITIONAL BRANCHING

In conditional branching, PMATE jumps to a **branch point** in the macro if a specific condition is met. Designate a branch point in a macro with a label (consisting of any character), preceded by a colon (:).

The branch command is J (Jump), followed by the branch-point label. If no numeric argument precedes J, or if the argument is nonzero (true), PMATE executes the command following the label. If the numeric argument is 0, PMATE continues with the command it is executing. For example, in the command string @M > 100JL$10K:L, PMATE jumps to label L if there are more than 100 bytes of memory left. Otherwise, PMATE deletes 10 lines of text.

4.6.4 EXITING FROM A MACRO

You can exit from a macro at any point with the command %. If no numeric argument precedes the % command, or if the numeric argument is nonzero (true), PMATE exits from the macro.

4.7 TRACING ERRORS

PMATE's Trace Mode and error traceback features help you to "debug" macros. Its error flag feature lets you test for and control errors. Chapter 4.7 explains how to use these features.

TRACE MODE 4.7.1

In **Trace Mode**, PMATE stops after each command in the macro so you can see how the command affects your text. Also, the ARG indicator displays the value of numeric arguments expressed as arithmetic and logical operations, variables, and functions. To use Trace Mode, put a question mark (?) at the beginning of the macro. When you execute the macro, the cursor stops before execution of every command. Press the Space bar to execute the next command.

You can also debug sections of a macro using **breakpoints**. Breakpoints mark specific sections of a macro that you want to trace. Put a question mark (?) before and after sections you want to trace. PMATE goes into Trace Mode when it reaches the first question mark. Press the Space bar to execute each command. PMATE leaves Trace Mode when it reaches the next question mark. You can put any number of breakpoints in a macro.

ERROR TRACEBACK 4.7.2

When a fatal error occurs while PMATE is executing a macro, the error message appears in the text area of the screen. The command string that has caused the error appears on the command line. The cursor points to the character just after the command that has caused the error. The status line tells you which buffer or permanent macro has executed the erroneous command. For example, if buffer 2 stores the command string with the error, the status line displays COMMAND FROM 2.

The error might be part of a macro that calls a macro. You can use the **error traceback** feature to find out which macro has called the command string with the error. With error traceback you pop a level by pressing the Space bar to see which command string executes the troublesome macro. If this command string is also a macro, you can pop another level by pressing the Space bar again. You can continue viewing the macros that call the macros until you reach the original

command string that executes the first macro. If you don't want to use error traceback, press Return instead of the Space bar just as you do with any other error.

4.7.3 THE ERROR FLAG

The three kinds of errors that occur in PMATE are:

- Fatal
- Optional
- Nonfatal

A **fatal error** occurs when PMATE does not recognize a command. It stops the execution of a macro. You can use Trace Mode or the error traceback feature to debug the macro and correct the mistake.

An **optional error** can be fatal or nonfatal. An example of a fatal optional error is when your macro executes a Change command and PMATE runs out of strings to change. Once PMATE can't find another occurrence of the string, execution of the macro stops and the STRING NOT FOUND error message is displayed on the screen. You can use error traceback to find the error. Or, you can press the Return key to eliminate the error message and execute another command.

This optional error can be made nonfatal by suppressing the error message with the E command. When the error message is suppressed, execution of the macro continues after an optional error. If the error message is suppressed and an error NGcurs, PMATE sets the value of the error flag to -1 and continues execution of the macro. You can use a logical operation as a numeric argument to test for the error. The function @E returns the value of the error flag. Thus, use the numeric argument @E = -1 to test for an error. PMATE resets the error flag to 0 both after you test with the @E function and after completion of the macro.

A **nonfatal error** occurs when a macro executes an M, L, P, or W command while the cursor is already at the end of the file. No real error occurs. The cursor has just gone as far as it can go. In this case, PMATE sets the value of the error flag to − 1 and continues with the execution of the macro. You can test for the occurrence of an error with the @E function, just as you do when you suppress the error message.

PROCESSING KEYBOARD INPUT 4.8

You can create a macro that stops in the middle of an editing operation and waits for keyboard input. Use the command G, followed by a string argument, in the macro. PMATE pauses during execution of the macro and updates the screen. During the pause, the string argument appears on the command line. You can edit the text with instant commands. Execution of the macro continues after you enter any character, other than an instant command, from the keyboard. Use the @K function if you want PMATE to return the ASCII value of this key.

If you precede the G command with a numeric argument that has a value of 0, the string argument that follows G is displayed on the command line, but the macro continues execution without keyboard input.

You can use character strings and multiple-digit numbers as keyboard input. Store the strings in buffers and the numbers as variables.

INSERTING COMMENTS IN MACROS 4.9

Because macros can be complex editing programs, PMATE lets you format your macros for easy modification. You can insert spaces, carriage returns, and tabs between commands. You can also put comments in the macros to document tasks performed by the commands. PMATE ignores all characters preceded by a semicolon (;) and followed by a carriage return. This macro, which changes all uppercase

alphabetical characters to lowercase, is an example of how a macro can be formatted with comments:

```
A                ;start at beginning of edit buffer
[                ;begin iteration
@T < "A JA       ;if the character at the cursor is not an alphabetical
                 ;character (if the ASCII value is less than the ASCII
                 ;value of A), jump to label A

@T! < "V0        ;assign the lowercase ASCII value of the character
                 ;to variable 0 by ORing it with the ASCII value of a
                 ;space (20H) since the uppercase ASCII value of a
                 ;character is 20H less than the lowercase value.

D @0I            ;delete character at cursor and insert uppercase value
                 ;of character

-M               ;move back to same character

:A M             ;move cursor to next character, setting error flag if
                 ;cursor is at the end of text

]                ;begin iteration with next character unless error flag
                 ;has been set
```


4.10 Q COMMANDS

Q commands, which are used to make changes to PMATE, are useful in macros. As demonstrated in Chapters 3 and 4, you use Q commands to:

- Switch between Auto Buffer and Manual Modes
- Divide text into pages while in Manual Mode
- Send a form-feed character to the printer

- Move the permanent macro file to and from the permanent macro area
- Change the input and output radixes
- Indicate the number of strings stored on the command line

Table 4-6 is a complete list of Q commands.

Table 4-6: Q Commands

COMMAND	DESCRIPTION
nQA	Indicates that n strings are stored on the command line.
QB	Rings bell. This is useful for telling you when a long command string or macro has finished execution.
nQC	Sets the control-shift character to the character represented by the ASCII value of n. The shift character is ignored when entered, but the character that follows is a control character. The control-shift character is currently set to ASCII 94.
nQD	Delays for a time proportional to n. This command can be used with L and QR to implement variable-speed scrolling.
0QE	Sets Type Out Mode to 0, which prints text as it is displayed. Control lines are printed, $ is printed as a dollar sign ($), and all control characters are printed as the character preceded by an up-arrow (^). Use this mode for printing macros.
1QE	Sets Type Out Mode to 1 (the default mode), which prints text on a regular printer. Tabs are expanded to spaces. Control lines are not printed, but affect the margins and tab stops. Other control characters are sent through to the printer.
2QE	Sets Type Out Mode to 2, which is used with printers that do their own formatting. Carriage returns are sent only at the end of a paragraph. Tabs are not expanded to spaces and all control sequences are sent through to the printer.
nQF	Sets the form-feed character to that represented by the ASCII value of n.
nQG	Turns garbage stacking off if n equals 0. Turns garbage stacking on if n is nonzero or missing.
nQH	Inserts n spaces at the cursor. This is useful for centering text. Because all spaces are inserted at once, this operation is faster than n[I $].

COMMAND	DESCRIPTION
nQI	Sets the current input radix to base n.
nQJ	Shifts the text display up or down n lines without changing the cursor location in text. The display is shifted as far as possible without moving the cursor beyond the screen boundaries set during configuration.
nQK	Sets Backup Mode for files. If n is 0, .BAK files are not created from old input files. If n is nonzero or missing, .BAK files are created.
nQL	Sets number of lines that instant commands ALT-U and ALT-J scroll.
QMC	Sends the permanent macro file to the permanent macro area.
QMG	Gets the permanent macro file from the permanent macro area.
tnQO	Sets the current output radix to base n.
nQP	Divides pages into n lines each.
nQQ	Shifts the text display left or right n columns without changing the cursor location in text. The display is shifted as far as possible without moving the cursor beyond the boundaries set during configuration.
nQR	Redraws screen if n is 0. Checks the keyboard without redrawing screen if n is −. @K returns the value of any key struck, or 0 if none. Use this command to create interactive macros where PMATE continues doing something, showing you the results, until you tell it to stop.
nQS	Sets the uppercase/lowercase shift character to the character represented by the ASCII value n. The shift character is ignored when input, but shifts the case of the next character entered.
nQT	Sends the character represented by the ASCII value n to the listing device.
nQU	Turns Auto Buffer Mode off if n is 0. Turns Auto Buffer Mode on if n is not 0.
nQV	Enables tab fill unless n is 0. When a character is inserted past the end of an existing line, PMATE inserts as many tabs and spaces as needed to fill out the line (see QY). If tab fill is not enabled, only spaces are used.
nQW	Turns the command-line error display off if n is nonzero and turns the command-line error display on if n is 0.
nQX	Moves the cursor to column n on the current line. Depending on the state of the free space flag (see QY), the cursor might not be able to go past the last character in a line.
QY	Sets the free space flag if n is 0, letting the cursor move past the end of a line. When a character is inserted at such a cursor position, the necessary amount of spaces (or tabs—see QV) is inserted to extend the line to the new cursor location. If n is nonzero, the free space flag is reset, restricting cursor movement to existing text.

COMMAND	DESCRIPTION
nQZ	Prevents the cursor from moving past column n. Use this command to control the width of text when you need clean output on a limited-width printer. When the cursor reaches the restricted column, it stops advancing and a warning tone sounds. If n is missing, the default width of 250 columns is restored.
Q#	Toggles cursor between tag and current cursor position.
nQ −	Sets a flag to indicate whether numbers are displayed as signed integers or positive integers only. If n is 0, numbers are displayed as positive only. Otherwise they are displayed as signed. This affects the ARG display on the status line as well as numbers inserted in text with a backslash (\). When numbers are expressed as signed integers and more than 32K of memory remains, you see ARG = − 30536 displayed after you type the command @M (in order to see how much memory remains). You get a more understandable response to @M when numbers are expressed as positive integers only.
nQ/	Sets indentation to column n. After you press the Return key in Overtype or Insert Mode, PMATE moves the cursor to column n. Set the free space flag (see QY) to use this feature, because spaces and tabs are not inserted until a character is typed (so that blank lines do not contain unnecessary spaces). If n is missing, Q/ increments the indentation by one column and − Q/ decrements it by one column.
Q <	Saves the current edit buffer, Format Mode setting, and garbage stack. They are restored once, after the next error message. 0Q < disables this.
nQ >	Gets the character represented by ASCII value n as if you type the character from the keyboard. If there are other characters in the keyboard buffer, the character represented by n goes to the top of the queue. (− G returns the numeric representation of instant commands as command code + 256.)
nQ!	Stores n in memory in the location pointed to by variable 9.
nQx	Sets user variable x (0–9) to n. Use these 10 user variables with user-written I/O drivers.

A

CUSTOMIZATION GUIDE

GENERATING A CUSTOM CONFIGURATION FILE A.1

CONFIG.CNF is a file that contains the configuration parameters for PMATE. You can use PMATE to change these configuration parameters. Then you can use the edited CONFIG.CNF file to create a custom version of PMATE.

CONFIG.CNF contains a series of questions or statements that prompt you to make or change a parameter assignment. Three asterisks (***) follow each question or statement. Enter (or change) the parameter after the asterisks. The parameter is a number or yes or no. Enter the number in decimal or hex; identify a hex number with "H". If more than one number is required, separate them by spaces.

You can give your custom version of CONFIG.CNF a new filename, as long as it has extension .CNF. When running CONPMATE, you must specify your custom configuration file after entering the command; otherwise, CONFIG.CNF is used.

To configure a version of PMATE, obtaining information from the file MYCONFIG.CNF, type:

```
conpmate myconfig
```

Upon completion, you must save this custom version of PMATE on disk. To do so, type:

```
XDpmate$$
```

If PMATE.COM already exists on this disk, you can use PMATE1 (XDPMATE1$$) and rename it later.

CONFIGURATION INFORMATION

CONFIG.CNF asks you these questions during the configuration process.

- How many lines from the center of screen can cursor wander?

  Since the display screen can hold only a small portion of the text file being edited, you need to scroll the display as the cursor moves off of it. Typically, the display scrolls to prevent the cursor from moving down past the bottom line or up past the top. Keep one or two lines above or below the cursor at all times, so you can easily see the context you are working in.

  The number you enter in response to the question indicates how far from the center line of the text display the cursor is allowed to move before a scroll occurs. If this number is 0, the cursor remains on the middle line of the display. Any up or down cursor motion causes a screen scroll. Using 0 (or a small number) keeps maximum context and requires the most screen scrolling. For example, on a 24-line screen, 21 lines are dedicated to text display. Entering 10 (don't use anything bigger) produces a display that scrolls only at either limit; 8 leaves two lines on top or bottom before scrolling; and 1 restricts the cursor to the three center lines.

- How many lines do you want redrawn in foreground?

  This sets the number of lines to be redrawn on the screen before PMATE responds to the next keystroke. (In other words, this many lines are kept up to date at all times; the rest are redrawn when PMATE has the time.) The smaller this number, the faster PMATE's overall response is, but the less you can see the effect of your keystrokes.

- Should display proceed from top to bottom (or from cursor outward)?

  PMATE screen redraws proceed in one of two ways. The traditional method is to start at the top, and work down. PMATE can also start drawing on the line containing the cursor, and work outward, alternately displaying lines on either side. If the cursor is on the bottom line, the display proceeds from bottom up; if the cursor is at the top,

the display proceeds in the usual top-down manner. This second method has the advantage of showing you the text in which you are most interested—that near the cursor.

Answer yes to get a top-down display, and no to get a display proceeding from the cursor outward.

▶ Should cursor be displayed before each line is redrawn?

By addressing the cursor to its final position before each line is redrawn, you don't lose track of where the cursor is as the screen redraw proceeds. As usual, there is a trade-off. Twice as many cursor addressing sequences now need to be performed. If your display requires a significant delay after each cursor addressing operation, this can slow down a screen redraw noticeably.

▶ Maximum number of instant commands to buffer.

PMATE constantly polls the keyboard to keep from missing any keystrokes while it is doing other tasks. This buffering, however, can allow certain instant commands (such as deletes or cursor motion) to run away when used with auto-repeat. You can limit the severity of this run-away by answering this question with a small number (at least 1). If you quickly enter four ALT-Ds and only two characters are deleted, you will know why. As always, compromise.

▶ Number of characters to shift for horizontal scroll.

PMATE allows lines of up to 250 characters in length. Since displays rarely show more than 80 of those, PMATE shifts the entire display over to keep the cursor from moving off the right end.

Enter the number of characters to be shifted at one time. If you enter 1, the display scrolls one character at a time as you enter a long line. This is very natural, but you'll notice continual screen activity as the line progresses. If this bothers you, choose a larger number.

▶ Are carriage returns and tabs to be inserted while in Overtype Mode?

If you answer no, Returns are inserted only at the end of text, and tabs are inserted only at the end of a line. Except in Overtype Mode, these characters just move the cursor—to the beginning of the next line, or to the character following the next tab. If you answer yes,

these characters are inserted any time they are typed (and the cursor motion keys must be used for moving the cursor).

- Do you want .BAK files to be generated automatically?

  Most text editors do not delete the original input file after a completed edit pass. Instead, they rename it, giving it the extension .BAK (any old file by that name is deleted). If you answer yes to this question, PMATE does this, too. If you don't like to clutter your disks with two copies of every file, answer no. You can use the QK command to change this while editing.

- Reserved size of garbage area.

  PMATE stacks its garbage in any available memory space so it can be retrieved later if needed. By permanently reserving some space for garbage, you ensure that you can recover at least a small item or two. Reserving space for garbage also lets you use the stack for moving text. Enter the number of bytes you want to reserve (it must be at least 1). Remember to leave some memory to edit text.

- Size of permanent macro area.

  Enter the amount of memory (in bytes) you want to reserve for permanent macros. PMATE doesn't let you load permanent macros requiring more space than you have allocated.

- Should disk buffering be automatic?

  Answer yes if you want automatic disk buffering; no if you don't. This can be later changed by the QU command.

- Start in Command Mode (0), Insert Mode (1), or Overtype Mode (2)?

  Your answer to this question sets the mode that PMATE is in when you initialize the program. This mode is also entered after ALT-C abort and after any errors. By choosing 1 or 2 and adding appropriate permanent macros (with associated instant commands), you can eliminate Command Mode.

CUSTOMIZING THE KEYBOARD A.2

PMATE lets you determine the keystroke required to execute instant commands. To suit your preferences and hardware, CONPMATE can create a version of PMATE that assigns any keystrokes you want to any one of a list of commands.

CONPMATE asks for the following information during the configuration process.

- Maximum number of codes entered for instant commands below.

  You can enter as many as eight codes before an instant command executes. This can be a series of keystrokes or the multi-code sequence sent out by function keys. Enter the maximum number of codes entered for any of the commands below.

- Control shift character.

  If you are using control codes for instant commands, you need to designate a "control shift character" if you want to enter these control characters in text (see the QC command). Enter the ASCII code for your control shift character in response to this question (up-arrow is the usual choice).

After these questions, you'll see a list of instant command functions. After each function, enter the ASCII codes of the required keystroke sequence. Not all functions must be implemented (leave the function blank if you choose not to implement it). You can assign several different sets of keystrokes to the same instant command using the configuration file. CONPMATE interprets all lines that start with *** as subsequent entries for the instant command listed previously. .

An example should make this clearer:

```
Delete character *** 4

Delete line *** 11
*** 29 49
*** 29 50

Delete word forwards *** 23
*** 30

Delete word backwards *** 17
```

The CONFIG file provided implements the standard PMATE instant command set.

A

The PMATE cursor motion commands require more explanation. Line-oriented cursor motion is implemented as follows:

Left: Move cursor one character to the left. If cursor is at the beginning of a line, it moves to the last character of the preceding line.

Right: Move cursor one character to the right. If cursor is at the last character of a line, it moves to the beginning of the following line.

Up: Move to the beginning of the current line. If cursor is at the beginning of a line, it moves to the beginning of the preceding line.

Down: Move cursor to the beginning of the following line.

This combination of cursor motion is selected by entering codes next to Move left, Move right, Move up, and Move down. These commands make it easy for you to move the cursor to either end of a line, and they are well-suited to editing programs. These commands do not, however, let you easily move the cursor down through columnar material.

Another way to move the cursor vertically is geometric motion. If the cursor is at column 5, moving up one line does not move the cursor out of column 5. Normally, the cursor can't go past the Return at the end of a line or move to the middle of a tab space; the cursor lands

only on a text character. If you answer "Allow cursor to move into free space?" with yes, the cursor can move anywhere on the screen as long as it stays in the same column it occupied in its original location. If you insert a character while the cursor is "floating," the appropriate number of spaces (and possibly tabs—see the QV command) are inserted so that the character actually appears where you expect.

Move right (geometric) and Move left (geometric) always keep the cursor on the same line and always move it one column at a time. This causes trouble if the cursor has not been allowed into free space. Whenever the cursor reaches a tab, it tries to move over another column but can't land there. When this happens, the cursor goes back to the beginning of the tab and stays there.

A final option mixes the two approaches just mentioned. Overtype Mode works well with a column format since it is a geometric cursor (a Return moves the cursor to the beginning of a line). When working on line-oriented material, you usually use Insert Mode. When you enter codes in the Move up (mixed) and other (mixed) categories, the line-oriented cursor routines are used in Insert Mode, and the geometric routines are used in Overtype Mode.

The move-multiple-lines commands also have geometric and mixed variants. The number of lines moved by any of these commands is set by the QL command. The Move Page Up and Move Page Down commands move up or down exactly one screen, independent of the QL setting.

The instant commands that move the cursor to the top and bottom of text have several more varieties. You can configure PMATE so that ALT-A (or another chosen keystroke) moves the cursor to the beginning or end of the file, or to the beginning or end of the text in memory. The first choice is the default. If you want better control over what is in memory and what is on disk, you can choose the latter (then a UA or UZ command moves the cursor to the beginning or end of the file).

The next section of the configuration file lets you redefine the codes that control certain built-in PMATE functions. If you want to redefine one of these, enter the new code (or codes) following the *** as for any of the instant commands. The Escape, Tab, and Return keys can also be redefined, but you will rarely want to do this.

The end of the keyboard configuration section lets you define your own instant commands by assigning keystrokes to permanent macros 0–9. The macro named "0" in the permanent macro area can be executed every time an assigned key is pressed. Macros 0–9 are used because they cannot be executed from the command line and serve no other purpose (.0$$ executes buffer 0, not permanent macro 0). However, additional macros can be added to the list. For example, permanent macro a can be invoked every time you type ALT-a by adding the line:

```
a *** 1
```

A

B

MACRO EXAMPLES AND IDEAS

This appendix contains examples of macros which you can use as presented or as a guide for building your own macros. Some of the examples are relatively simple macros; they are explained in more detail than later ones. None of the examples, however, are intended to be polished final products. Instead, they should give you an idea of the types of operations you can perform with macros, and provide you with a foundation on which to build.

The best way to understand how and why these macros work is to enter them, execute them, and then run them in Trace Mode. You should read up on Trace Mode and breakpoints in Chapter 4 before using the sample macros. To refresh your memory, though, here's a summary: Put a question mark (?) at the beginning of the macro or at the place where you stop understanding what's going on. At this point, the macro executes one step at a time, showing you the results of its latest operation. The macro continues only when you press the Space bar.

ADDING OR DELETING COMMENTS B.1

Programmers often "comment out" sections of code—a way of deleting a section from the program, but keeping the code in memory just in case it has to be replaced. In many programming languages, this is done by putting a semicolon at the beginning of each line. In PMATE, you can go into Insert Mode, enter a semicolon, move the cursor down, enter another semicolon, move the cursor, and so on. This isn't much trouble for a few lines, but the macro I;$L$$ works better if you need to enter a lot of lines. This macro inserts the semicolon and moves the cursor all at once. If you enter a series of Escapes, the command repeats until you reach your last line. Finally, try 20[I;$L]$$.

This command repeats the above sequence 20 times, commenting out 20 lines at a time. Any time you need to perform a repetitive sequence, think macro.

What if you need to delete all the comments from a file? If you've ever done that by hand, you will appreciate a macro which does it for you automatically. This macro assumes that comments begin with a semicolon; it deletes the comment starting at the semicolon, as well as any preceding tabs. Use it on programs, or on PMATE macros themselves:

```
[S;$ −M−SNI$ M K I
 $]
```

The left bracket starts a loop that deletes all comments. The first S finds a comment by searching for ;. Then, the macro looks for the tabs preceding the semicolon. Since the S left the cursor on the character just after the semicolon, the macro must move back one (− M) before looking for tabs. The next S searches backwards until it arrives at the first character that isn't a tab (NI matches anything except ALT-I, which is a tab) and leaves the cursor on that non-tab character. Then, the cursor points to the entire comment to be deleted. K deletes the comment, as well as the Return at the end of the line. The Return is then restored by the I. The right bracket loops back to the start of the macro. The macro terminates when the first S command cannot find any more comments.

B.2 SEARCH AND REPLACE MACROS

Escape characters in text present problems when a macro string needs to operate on those characters. If you want to put an Escape into text, I$$$ doesn't work, but 27I does. To avoid this problem, here's a macro that changes all Escapes in text to dollar signs (in case you ever need to write a section like this one):

```
[@T = 27[36R][M]@T = 0]
```

The first bracket starts iteration, for we want to change the entire text buffer. @T = 27 tests the character under the cursor to see if it's an Escape (ASCII code 27). If it is an Escape, the expression in the first set of brackets (36R) is executed. This replaces the Escape with a dollar sign (ASCII code 36).

If the character at the cursor is not an Escape, the expression in the second set of brackets moves the cursor on to the next character. @T = 0 tests to see if the cursor has reached the end of the text buffer (always a null). If the end has been reached, the iteration ends; if not, the macro goes back and checks the next character.

The command [Cblah$blew$] changes all occurrences of "blah" in the text buffer to "blew". Sometimes, though, you will want to replace only some of the occurrences. You can write a macro that stops at each "blah" and asks you whether you want to replace it. Put this command string in buffer 1:

```
2QA
[
        SAA$
        GType space to replace$
        @K = 32[ - CAA$AB]
]
```

Then type .1blah$blew$$.

The first line of the macro sets the number of string arguments required from the calling command (in this case, "blah" is the first and "blew" is the second). The next line searches for the first argument (blah). The G command then gives a prompt, displays the text buffer with the cursor located just past the next "blah", and waits for you to respond. If you respond with a space, @K = 32 is true, and the expression in brackets is executed. The "blah" changes to "blew" (the −C is necessary because the cursor has already been moved past "blah"). If you press anything other than the Space bar, the expression in brackets is ignored. The last line iterates back to the first bracket and the macro keeps looking for the "blah"s. The process will continue until the last "blah" or until you enter ALT-C.

B.3 TEXT OUTPUT PROCESSING

By itself, PMATE does not perform many print functions often associated with word processors. However, you can use PMATE with a separate output processor or you can write macros to do these functions. Here are a few ideas to get you started.

LINE CENTERING AND MARGIN ALIGNMENT

In Format Mode, this macro centers a line. Start by putting the cursor anywhere on the line to be centered.

```
L – M          ;move to end of current line
@W–@X/2V0      ;get one half the distance from right margin
               ;to current cursor position
               ;save it in variable 0.
0L             ;back to beginning of line
@0QH           ;insert number of spaces computed above
L              ;move on to next line
```


It's easy to make a macro that moves the line flush with the right margin—just get rid of the /2 after the @W – @X.

This next macro copies the character at the cursor position, leaving the rest of the line flush with the right margin. Use it, for example, on a table of contents. Start with:

Chapter 1.pg 1
Chapter 2.pg 24
Chapter 3.pg 30

Put the cursor on each decimal point in turn, execute the macro three times, and you are left with:

Chapter 1.....pg 1
Chapter 2.....pg 24
Chapter 3.....pg 30

```
@XV0          ;save the current column in variable 0
L − M         ;find end of line
@W − @XV1     ;amount of space needing fill to variable 1
@0QX          ;back to original cursor position
@TV2          ;save the character there in V2
@1QH          ;fill out line with spaces
@0QX          ;back to original cursor position again
@1[@2R]       ;now overtype spaces with the original character
```

The last three lines could have been replaced with @1[@2I]. However, replaces require much less memory than inserts; the suggested method executes faster.

PAGE HEADINGS AND PAGE NUMBERS

Here is an easy way to write a macro for page headings and numbering. Suppose buffer 1 contains a one-line heading which you want printed at the top of every page. And suppose you have put a # in that line at the place you want a page number inserted. Buffer 1 might contain:

Chapter 2 EXCITING DOCUMENT! page #

Enter into variable 0 the first page number: 5V0$$ is appropriate if Chapter 2 starts on page 5. Then, the following macro prints your file, using the above header and printing page numbers:

```
[             ;start iteration—will type till end of buffer
B2K           ;empty buffer 2
B2E           ;edit buffer 2
B1G           ;get prototype page header from buffer 1
A             ;find its beginning
S#$ − D       ;find "#" and delete it
@0\\          ;insert page number there instead
```

```
VA0          ;increment page number—ready for next page
XT           ;type header
10QT         ;send a linefeed to skip line after header
BTE          ;back to text buffer
60XT         ;type next 60 lines of document
4[10QT]      ;send 4 linefeeds to complete a 66 line page
@T = 0]      ;keep typing until the text buffer is finished
```

There are lots of ways to expand on this. For documents larger than available memory, have the macro read in successive pages. Define a print format line, starting with a unique character (maybe P). The print macro does not type this line, but uses its information for further formatting. The print format can include output functions like double space, center (see the preceding macro), and so on. Header information no longer needs to be put in a buffer beforehand, but can be moved there from the print format line as the macro proceeds.

B.4 FORMS AND MATH

Sometimes you need to get a whole string from the keyboard. The next example macro gets a string from the keyboard, echoes what is typed in the command/prompt line, and saves that string in buffer 9. The string ends on a Return. To correct mistakes on entry, a backspace deletes the last character entered.

```
B9K                ;delete old contents of buffer 9
[                  ;start iteration
GA@9$              ;get a character, displaying contents
                   ;of buffer 9 on command line
@K = 13_           ;if character is a Return, break (all done)
B9E                ;now go into buffer 9
@K = 8[ - D][@KI]  ;if character is a backspace
                   ;delete previously entered character
                   ;otherwise, insert new character
BTE                ;back to text buffer
]
```

You can use this macro to create an interactive macro for filling out forms. For instance, a preexisting invoice skeleton can be read in. You can then use the full capabilities of PMATE to fill in the blanks, or an invoice macro can set the cursor into each field and prompt for information. The entry is accumulated in buffer 9 and inserted in the text when finished. The invoice macro can check for illegal entries and prevent you from totally destroying the invoice form. Furthermore, the macro can be used by someone unfamiliar with PMATE.

You frequently need to add up numbers when you're filling out a form. Here's a macro that helps you do this. It adds the number pointed to by the cursor to a number stored in buffer 9.

```
[ M (@T > "9) ! (@T < "0)]  ;Move cursor until end of number
                            ;is found
0V1                         ;initialize carry
B9E                         ;number to add to is in buffer 9
Z                           ;move to end of that number
|                           ;iterate one digit at a time
                            ;starting with least significant
BTE                         ;back to first number
 - M                        ;get next most significant digit
(@T > "9) ! (@T < "0)       ;not a digit?
[M 0V0]|                    ;no, don't move past it
                            ;0 to V0 is number to be added
@T - "0V0]                  ;a digit—gets its numeric
                            ;value to V0.
B9E                         ;now go to buffer 9
 - M                        ;get next most significant digit
@E_                         ;done if out of digits
@T + @0 + @1V0              ;add digit from text, and carry
                            ;to it result to V0
@0 > "9[1V1 @0 - 10 R       ;if greater than 9, set carry to 1,
                            ;subtract 10 and store result in text
|[0V1 @0R]                  ;not greater than 9, set carry to
                            ;0 and store in text
 - M                        ;R has moved cursor, so move back
|                           ;on to next digit
BTE
```

The number of digits stored in buffer 9 controls the precision of the result. If you start with 000000000, numbers up to 999,999,999 can be accumulated. The result can be moved back into the main text buffer.

B.5 TWO PRINT MACROS

This simple macro lets you type directly on your printer, using the keyboard as if it were a typewriter. The third line implements an automatic linefeed. If the macro finds a Return, it sends a linefeed also. Any other character is sent as is.

```
[
GDIRECT TYPE$
@K = 13[13QT 10QT][@KQT]
]
```

Here's a macro that prints an alphabetized directory listing; it should suggest many other applications:

```
B1K                 ;clear buffer 1 to hold directory list
B1E                 ;go into buffer 1
XL$                 ;get a directory listing
A                   ;go to beginning of directory
[                   ;begin overall loop
BC                  ;copy first filename to buffer 0— will try to
                    ;find filenames earlier alphabetically.
[                   ;this loop finds earliest filename
@HA@0$ < 0[BC][L]   ;compare next filename to earliest already
                    ;found—if this one is earlier, copy it to
                    ;buffer 0, otherwise, advance to next
@T = 0]             ;iterate until end of directory list
A                   ;back to top of directory list
SA@0$               ;match the earliest entry stored in buffer 0
 -1XT               ;type it out
 -K                 ;and then delete it
A@T = 0]            ;back to beginning—continue unless
                    ;list is now empty
BTE                 ;back to text buffer when all done
```

CURSOR MOTION B.6

Here (without comment) are the macros that PMATE uses to implement the cursor motion instant commands. If you want to customize cursor motion to your own taste, this gives you a place to start.

Up:	**@V = 2[@X, − L@SQX][− M0L]**
Down:	**@V = 2[@X,L@SQX][L]**
Left:	**@V = 2[@X > 0[@X − 1QX][− M]**
Right:	**@V = 2[@X + 1QX][M]**

ASCII CONVERSION CHART

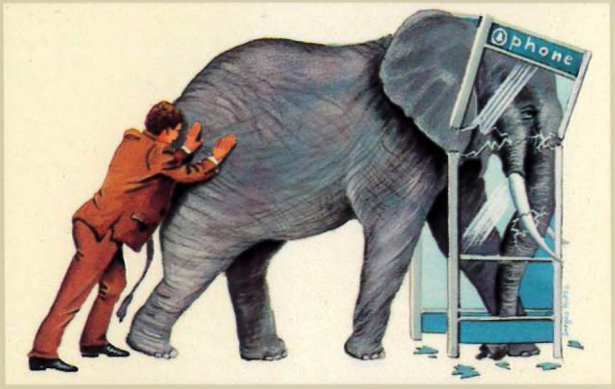
Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	040	28H	(
001	01H	SOH	041	29H	)
002	02H	STX	042	2AH	*
003	03H	ETX	043	2BH	+
004	04H	EOT	044	2CH	,
005	05H	ENQ	045	2DH	-
006	06H	ACK	046	2EH	.
007	07H	BEL	047	2FH	/
008	08H	BS	048	30H	0
009	09H	HT	049	31H	1
010	0AH	LF	050	32H	2
011	0BH	VT	051	33H	3
012	0CH	FF	052	34H	4
013	0DH	CR	053	35H	5
014	0EH	SO	054	36H	6
015	0FH	SI	055	37H	7
016	10H	DLE	056	38H	8
017	11H	DC1	057	39H	9
018	12H	DC2	058	3AH	:
019	13H	DC3	059	3BH	;
020	14H	DC4	060	3CH	<
021	15H	NAK	061	3DH	=
022	16H	SYN	062	3EH	>
023	17H	ETB	063	3FH	?
024	18H	CAN	064	40H	@
025	19H	EM	065	41H	A
026	1AH	SUB	066	42H	B
027	1BH	ESCAPE	067	43H	C
028	1CH	FS	068	44H	D
029	1DH	GS	069	45H	E
030	1EH	RS	070	46H	F
031	1FH	US	071	47H	G
032	20H	(sp)	072	48H	H
033	21H	!	073	49H	I
034	22H	"	074	4AH	J
035	23H	#	075	4BH	K
036	24H	$	076	4CH	L
037	25H	%	077	4DH	M
038	26H	&	078	4EH	N
039	27H	'	079	4FH	O

Dec	Hex	CHR	Dec	Hex	CHR
080	50H	P	104	68H	h
081	51H	Q	105	69H	i
082	52H	R	106	6AH	j
083	53H	S	107	6BH	k
084	54H	T	108	6CH	l
085	55H	U	109	6DH	m
086	56H	V	110	6EH	n
087	57H	W	111	6FH	o
088	58H	X	112	70H	p
089	59H	Y	113	71H	q
090	5AH	Z	114	72H	r
091	5BH	[	115	73H	s
092	5CH	\	116	74H	t
093	5DH	]	117	75H	u
094	5EH	^	118	76H	v
095	5FH	_	119	77H	w
096	60H	`	120	78H	x
097	61H	a	121	79H	y
098	62H	b	122	7AH	z
099	63H	c	123	7BH	{
100	64H	d	124	7CH	|
101	65H	e	125	7DH	}
102	66H	f	126	7EH	~
103	67H	g	127	7FH	DEL

NOTE: In the column headings, DEC means decimal, Hex means hexadecimal (H) and CHR means character. LF = Linefeed, FF = Form feed, CR = Carriage return, and DEL = Delete.

INDEX

Plink86™



SQUEEZING A LARGE PROGRAM INTO A SMALL MEMORY SPACE?

The Overlay Linkage Editor That Brings Modular Programming to 8088/86-Based Micros.

If you're tired of shoe-horning large programs into small memory, you're ready for Plink86. Now you can write a program as large and complex as you want and not worry about whether it will fit within available memory constraints, especially if you're porting software down from minicomputers.

You can divide your program into any number of tree-structured overlay areas. Handle diskette changes while running large programs...Segment the program for add-on packages...Work on modules individually...Then link them into executable files. And, there's no need to make changes to the source program modules.

Phoenix

Plink86™

Two-Pass Linkage Editor – Plink86 accepts any object file conforming to the Intel or Microsoft format and outputs executable program files. Each input file is read twice. During the first pass Plink86 determines which modules are to be loaded and allocates memory segment addresses. During the second pass, the output file is created. All information about program modules is available before the output file is created, giving you greater flexibility in assigning memory addresses.

Modules created by the Microsoft assembler or any of Microsoft's 8088/86 compilers may be linked. Plink86 also works with other popular languages, like Lattice C, Computer Innovations' C86, or mbp/COBOL.

Automatic disk buffering ensures that Plink86 won't run out of space for symbol names. Plink86 allows three different kinds of paginated, sorted memory map reports to be written to the printer or disk file. These reports enable symbols to be defined as absolute addresses or as offsets to other symbols, and produces symbolic information for Phoenix's Pfix86 Plus™ debugger.

Overlay Management – Plink86's easy-to-use overlay description language lets you specify your overlay structure in one place in your program. Up to 4095 overlays can be stacked 32 deep. And, you don't have to recompile to change the overlay structure.

Once your structure is defined, the overlay manager is bound into your compiled program. Automatically swapping modules between disk and memory. Without the need for calls from the source program. So that each module can temporarily occupy the same memory space.

It's easy to use the same module in different programs, or to quickly experiment with changes to the overlay structure of an existing program. You can use one overlay to access code and data in other overlays. Each overlay may contain as many entry points as you want. You can write overlays making up a program to the same file as the main program, or to one or more other files which may be stored on separate diskettes.

The Library Manager – Plib86™ is included as part of the Plink86 package. With it you can build libraries from scratch. Add or delete modules from existing libraries. Merge libraries. And produce cross-reference listings. Modules may be chosen for processing using the same library search algorithm used by Plink86, so that portions of one or more libraries can be cross-referenced or merged into a new library.

I/O – During processing, Plink86 uses all available memory to hold I/O buffers and the description of the program. The program description is automatically paged to disk if there is not enough free memory to hold it. The program may define up to about 35,000 program description objects (symbols, segments, groups, etc.) So Plink86 can link almost any size program you can write.

Language Interfaces – Microsoft C; Microsoft FORTRAN; Microsoft Pascal; Microsoft BASIC; Microsoft COBOL; Microsoft Assembler; Lattice C; Computer Innovations' C86; mbp/COBOL.

Operating Systems – PC DOS, MS-DOS™ and CP/M-86™ (Lattice C and Digital Research's Assembler only).

Plink86. One in a series of software development tools by Phoenix. It's the right tool for the job.

Phoenix

Phoenix Computer Products Corporation
1416 Providence Highway, Suite 220
Norwood, MA 02062
(800) 344-7200
In Massachusetts (617) 769-7020

PLINK

VICTOR Publications
380 El Pueblo Road
Scotts Valley, California 95066
(408) 438-6680

First VICTOR printing November, 1983.

ISBN 0-88182-040-7 Printed in U.S.A.

CONTENTS

APPENDIXES

FIGURES

TABLES

CHAPTERS

1

INTRODUCTION

The PLINK program takes individually compiled modules of 8086/8088 object code and links them into one or more relocatable files that can be loaded and executed by your computer's operating system. PLINK can handle the output from various popular compilers available for the 8086/8088 microprocessor, and an overlay feature lets you run programs larger than your computer's available memory.

ABOUT THIS MANUAL 1.1

The manual assumes that you have some programming experience. It's designed to be read from front to back—each chapter assumes that you understand the information in all the previous chapters.

Chapter 2 gives an introduction to linkage editors in general and to PLINK specifically. Start with Chapter 2 if you are not familiar with linkage editors; otherwise, start with Chapter 3.

Chapter 3 shows how to get started with PLINK. It describes the PLINK package and some simple checkout procedures.

Chapter 4 is a complete listing of the commands and features offered by PLINK.

Chapter 5 shows how to use PLINK with several popular compilers. The chapter gives an informal explanation of the most commonly used commands, and describes the commands peculiar to MS-DOS. This chapter gives sufficient information and examples for many applications of PLINK.

Error codes, warning messages, and overlay tables are discussed in the appendixes.

1.2 USING PLINK WITH COMPILERS NOT MENTIONED IN THIS MANUAL

This manual tells how to use PLINK with the most popular compilers on the market at the time of this writing. If you want to use PLINK with a compiler not mentioned in this manual, you are pretty much on your own. You may find that a new compiler offered by a particular software company is similar to one already sold by that company. If this is the case, there's a good chance that PLINK is compatible with the compiler, although you may have to use some obscure commands to structure your linked program. Ask your dealer for help in adapting PLINK for use with the compiler.

2

LINKAGE EDITOR CONCEPTS

This chapter describes the basic functions of a linkage editor and defines some terms used throughout this manual. It also shows how PLINK interacts with the 8086/8088 microprocessor, and how PLINK links and stores programs.

THE BASICS 2.1

WHAT DOES A LINKAGE EDITOR DO? 2.1.1

A linkage editor has two main purposes: it helps solve the memory management problem for programs, and it supports modular programming.

Memory Management

Computer memories fall into two categories. One kind is fast, but also small and expensive (semiconductor memories, for example). The other kind is slow, but has a lot of memory space and costs much less per bit of information stored (for example, disks or diskettes). Because of these design constraints, most modern computers have a two-tiered memory system. To run programs quickly, the programs and the data they require are loaded into a fast "primary" memory. A large, inexpensive "secondary" memory provides adequate storage at a reasonable cost.

Some interpretive languages offer memory management help for program code. For example, many BASIC interpreters let you "chain" one program to another. The new program is then loaded into

memory on top of the old one and execution continues. Some fancier interpreters load individual procedures automatically from disk as the executing program needs them. The interpreter has full control over program execution, and can easily catch references to objects not in primary memory and then load those objects. However, if a program is executing in the form of machine code, this task is difficult without some help from the processor. Because of this, noninterpretive language compilers do not provide for automatic memory allocation themselves. Since most major software packages are not written as interpretive systems, the memory management problem remains unsolved.

A linkage editor such as PLINK handles the memory management problem for noninterpreted programs. When used with programs that are executed directly by the processor, PLINK offers automatic methods for swapping various parts of the program in and out of memory as required during execution. Because portions of the program share the same primary memory space, the program's total primary memory requirement can be greatly reduced.

Modular Programming

It is often convenient (if not essential) to divide a large programming job into smaller pieces called "modules" that can be edited and compiled separately. Because many microcomputer compilers can only compile a limited number of code lines at once, you may have to use modules regardless of your preferences. Modular programming also lets you organize a program into pieces that are easier to understand and work with. Using a linkage editor to "link" the pieces of the program saves time because only the affected modules are recompiled when you make a change. The linkage process is generally faster than compilation.

Once you create a modular program, you may find that some of the modules are useful in a different program. With a little work, you can make these modules more general and use them in many programs. In this way, you can make a "library" of useful routines that can be linked in by the linkage editor whenever needed. In fact, most compilers are sold with a library that supports functions not supported by

the hardware and have to be implemented as procedure calls (such as arithmetic on real numbers). The compiler library also contains modules that support high-level language features, such as formatted output in FORTRAN. This library is often called the "run-time support," because its modules are required while the program executes.

TERMINOLOGY 2.1.2

This section contains a glossary of some basic terms commonly used when discussing linkage editors.

Object File

The compiler (or assembler) produces the object file as output after compiling a program. The linkage editor takes the object file as input.

Relocatable File

This is another term for object file. "Relocatable" means the file contains code that the linkage editor can modify to execute at any address in the computer memory. This term is often shortened to "REL FILE."

Module

This is the smallest unit of code that can be compiled at one time. The relocatable file created by a compiler typically contains one module. You usually create a module for each major function within a program; however, a compiler usually lets you put several procedures or functions into a single module. Although the functions and procedures in a module can be called separately, the linkage editor treats the module as a single entity, and the procedures can no longer be separated.

Library

A library is a relocatable file that contains more than one module. A library often has an index to its modules; this helps the linkage editor to quickly find the modules it needs. (This kind of library is called an "indexed library.")

Segment

A segment (or logical segment) is the basic unit of code or data manipulated by the linkage editor. A module is made up of segments. Usually there is at least one segment for the module's code and another one for data.

A main job of the linkage editor is to determine how much memory each segment of each module requires and where in memory to put it. One segment may need to access code or data in another. For example, the code segment of a module may need to load some data from the data segment into a register. These references are "fixed up" by the linkage editor after the segment addresses have been selected; i.e., the address is plugged into the right place in the segment making the reference.

Common Block

FORTRAN and other compilers often create a segment for each common block accessed by a module. A common block is a way for several modules to share a data area. A common block referenced by a particular module is overlapped by common blocks referenced by other modules. Each module references the common block using the same name. The size of the common block is the same as that of the largest version of the common block specified by any module.

Symbol

A symbol is much like an identifier in a program. It is a name for a value which can be a constant, the address of a piece of code, or data somewhere in the program. The first kind is called an "absolute symbol." The second is called a "relative symbol," because it is defined relative to a segment. If a segment is 100 bytes long, for example, you could define a relative symbol as the address of that segment plus 50 bytes. The linkage editor then determines the address of the symbol by adding 50 to the address it selects for the segment. When segments contain references to a symbol, those references are modified by the linkage editor to contain the assigned address or value of the symbol.

Public Symbol

Sometimes called an "internal symbol," a public symbol is a symbol whose value is defined inside the module that contains it. It is called "public" because its value is available to other segments in other modules. Only those symbols deliberately made public by the compiler are visible to other modules.

External Symbol

An external symbol is a symbol that has its value supplied by a module other than the one containing it. All references to an external symbol are fixed up by the linkage editor when its value becomes known. If an external symbol is never defined by any module, the linkage editor gives an error message.

Library Search

This term refers to the way libraries are processed by the linkage editor. When the linkage editor encounters an undefined external symbol, it looks up the symbol in the library index. If a module in the library defines the symbol (i.e., specifies the symbol as public), that module is included in the program. If the library does not have an index, the

linkage editor usually scans the entire library one or more times to find the needed module. A module loaded from the library can contain undefined external symbols of its own; the linkage editor keeps searching to find those symbols as well.

Section

This part of the program is loaded from disk into memory as a single unit. A section is also called a "load module."

The linkage editor breaks modules into segments and then regroups those segments into sections for execution. The program section that contains the main program modules usually is loaded into memory by the operating system when the program is executed. Other sections are loaded at the same time, or loaded later by a run-time routine supplied by the linkage editor.

Overlay

An overlay is a section that shares all or part of its memory with at least one other section. Overlays are brought into memory as the program runs, usually destroying other overlays using the same memory space. This reduces the memory requirement of the program.

Overlay Loader

Sometimes the operating system loads overlays into memory as required; otherwise, an overlay loader is included in the program by the linkage editor. PLINK invokes the overlay loader automatically each time it is needed.

BASIC 8086/8088 CONCEPTS 2.2

The last section discussed what linkage editors do and defined some terms associated with linking. Because linkage editors are unavoidably involved in the basic addressing mechanisms of the computer they are run on, we also need to introduce some concepts specific to the Intel 8086/8088 processor used by your computer.

The addressing scheme of the Intel 8086/8088 processor is based on a 16-bit address, providing a 64K byte address space for the processor (1 Kbyte is equal to 1024 bytes). To access a full megabyte of main memory, the 8086/8088 provides four segment registers that provide a 20-bit address. The word "segment" has a different meaning here than in the last section. Here it refers to a 64K "physical segment," as opposed to the "logical segment" discussed earlier.

The segment registers (CS, DS, SS, and ES) are each only 16 bits long, but each contains a paragraph address, where a paragraph is 16 bytes. In this way, the segment registers address 16 * 64K bytes or 1 megabyte. The 16-bit address used by most instructions are treated as an offset to one of the segment registers. Although a program can access only four 64K pieces of memory at once, the segment registers can be moved around to access any part of the megabyte address space.

Whenever a memory address is needed, this dual addressing scheme requires the linkage editor to know the address in the segment register that will be used as a base. To provide this information, the 8086/8088 processor uses public segments, private segments, groups, and classes.

PUBLIC SEGMENTS 2.2.1

A public segment can be "combined" with other segments of the same name. Combined segments are concatenated so that they occupy adjacent locations in memory. Often a combined segment is accessed by putting a base register at the low end. PLINK combines segments on

request, but only if they are in the same section. Even when they are deliberately kept apart, the segments are still addressed by assuming the base address is the address of the segment lowest in memory.

PLINK creates reports called "memory maps" that describe the address allocation of the program. A public segment within a section causes only one entry to be listed in the map, even if several modules define the segment. The size of the public segment is the sum of the sizes of the original segments.

2.2.2 PRIVATE SEGMENTS

Private segments cannot be combined. Private segments usually are addressed by setting a base register to the front. The 8086/8088 has "long call" and "long jump" instructions that set a base register while changing the execution address. Many MS-DOS compilers produce a private segment for the module code, and use long jumps and calls to set the segment register to the front of the called segment. This lets programs have more than 64K of code. At run-time, MS-DOS fixes up the segment addresses of the program by adding the paragraph address where the program was loaded. The PLINK overlay loader fixes the paragraph addresses contained in overlays as they are loaded.

2.2.3 GROUPS

A group is a collection of segments accessed by the same segment register. The address of the segment register is the address of the segment within a group having the lowest memory address. If the end of the highest segment is 64K or less from the start of the lowest one, any of the segments can be addressed with a 16-bit offset. Most compilers group the data segments of the program into a single group called

DGROUP. A segment register points to DGROUP when the program begins, and continues to point there while the program executes. Compilers that follow this procedure cannot create programs with more than 64K of data area.

PLINK does not move segments from one place to another to put the segments of a group within 64K of each other. Instead, PLINK gives warning messages if a segment cannot be addressed from the base of the group. If requested, PLINK can put all segments of a particular group into a given section of the program. Some examples of this feature are given later.

CLASSES 2.2.4

You can also refer to a collection of segments by a class name. The class name can be used to move all members of a class to a specified section without naming all the members of that class. Where possible, PLINK puts members of the same class into adjacent areas of memory and in the order the class names are given. Within a class, segments are ordered in the sequence the segment names are given.

Every segment is assigned to a class by the compiler or assembler. Public segments must have the same class name in order to be combined.

The 8086/8088 processor also uses an "overlay name" for each segment. PLINK ignores the overlay name, but provides other ways to specify the overlay structure of your program.

2.3.1 PLINK DESIGN

PLINK is a two-pass linkage editor; that is, each of the input files is read twice. During the first pass, PLINK determines the modules to be loaded and allocates segment addresses. The output file is created during the second pass. This makes PLINK slower than one-pass linkers, but ensures that full information about all program modules is available before the output file is created. This provides greater flexibility in assigning memory addresses.

PLINK uses all available memory to hold I/O buffers and the description of the program being linked. The program description is automatically read to disk if there is not enough free memory to hold it. PLINK can define as many as 35 thousand program description objects (symbols, segments, groups, and so on). Almost any size program can be linked, although it might take a while.

2.3.2 STORING A PROGRAM ON DISK

Linkage editors use two methods to store the sections of a program on disk. One way is to put each section into a separate file. This wastes space in the disk directory and requires a directory search whenever a section must be loaded into memory. If a program is heavily overlaid, you lose time searching for overlay files in the disk directory. Also, you can never be certain if the correct versions of each file are on the disk at the same time.

The separate-file approach does have an advantage, however. If your system does not have a hard disk, you can access other parts of a program by swapping diskettes. In this way, you can use a program larger than your available memory. PLINK lets you arbitrarily group program sections into any number of output files, so the tradeoffs just discussed can be handled as desired.

Some linkage editors let you link a program in pieces. This lets you avoid a relink of the program if only a single section needs to be changed. PLINK can't do this; the entire program must be linked each time, even if the program is being loaded into more than one output file.

Actually, piecemeal linking is not appropriate for the 8086/8088 because most compilers require the data areas to be combined. This means that you would have to relink everything anyway if you changed the data.

If the time spent linking the program becomes unacceptable, you should consider breaking up the program into several smaller ones that are linked separately and chained by facilities offered in the operating system.

OVERLAYS 2.3.3

Many linkage editors require you to code calls to the overlay manager in the source program. This approach hinders modular programming; you can't link the same module into overlay structures in different programs without changing the calls and recompiling. This requirement also scatters the description of the overlays all over the program, making it hard to visualize the final overlay structure.

PLINK does not require changes to the source program modules. Instead, it uses an easily-understood overlay description language that lets you specify the overlay structure in one place. You do not need to recompile when changing the overlay structure. With PLINK, you can use the same module in different programs, or quickly experiment with changes to the overlay structure of an existing program.

PLINK does this by intercepting calls to overlaid routines and substituting a call to a small piece of code called an "overlay vector." The overlay vector calls the overlay loader to ensure that the needed overlay is loaded, and then jumps to the desired routine. This automatic overlay call mechanism lets you set up complex overlay structures without worrying about when to load the overlays.

3

GETTING STARTED

This chapter shows how to install PLINK and to make sure you have a good copy of the program. The chapter also gives a simple example of how to use PLINK, and describes the files on the PLINK distribution disk.

INSTALLING PLINK 3.1

First, copy the files on your PLINK distribution disk onto another disk. Then, log onto the disk containing the PLINK files and validate PLINK by running the CHECKSUM program supplied on the disk. To run CHECKSUM, type:

CHECKSUM PLINK86(cr)

The message "Checksum OK" should appear on the screen. If this message does not appear, run CHECKSUM again. If the "Checksum OK" message still does not appear, you probably have a bad copy of PLINK. Get a replacement from your dealer.

After the CHECKSUM program completes successfully, try to execute PLINK by typing:

PLINK86 @PLTEST(cr)

PLINK should execute for a minute and then issue this message:

```
PLTEST (3K)
```

This means that the PLTEST program has linked successfully. See if PLINK runs by typing:

PLTEST(cr)

PLINK should respond with the following message:

```
PLTEST OK
```


If either of these tests fail, then some part of your operating environment is incompatible with PLINK. (Contact your dealer for assistance.) If the tests complete successfully, you are ready to use PLINK.

3.2 USING PLINK

Here is a simple example of how to use PLINK. (Everything in this section is described in more detail in Chapter 4.) Suppose you have a complete assembler program consisting of the source file TEST.ASM, which is assembled to produce TEST.OBJ. To produce the file TEST.EXE, type:

PLINK86 FI TEST(cr)

PLINK accepts input on the command line as it is executed. Every PLINK statement begins with a key word. The key word in the last example is FI, an abbreviation for FILE. Key words can be abbreviated by leaving off characters on the right, as long as the resulting abbreviation is unique. Statements can be put together on the same line in any column—this is called "free format" input.

Each program must contain at least one FILE statement giving the names of the files to be linked. All modules appearing in the files are included in the output program. If you do not specify an extension for a filename, PLINK assigns the default .OBJ extension.

After you enter a command statement, PLINK links the program and then displays a message like this:

```
TEST.EXE (5K)
```


This particular message tells you that the output program TEST.EXE has been created successfully, and that the program needs 5K bytes of memory when executed. This size can differ from the actual size of the program on disk, especially if the program uses overlays.

Unless you specify otherwise, PLINK gets the name of the output file by using the name of the first input file with extension .EXE. The OUTPUT statement could have been used to give the name explicitly by typing:

PLINK OUTPUT TEST.EXE FILE TEST.OBJ(cr)

PLINK also accepts input interactively from the keyboard. Type:

PLINK86(cr)

and PLINK prompts you for input statements. You can enter as many lines as you want. PLINK checks your input for syntax and stores it until you enter a semicolon to terminate the last line.

If an error occurs, PLINK displays an error message on the screen. The message is often accompanied by an error number. Look up this number in Appendix C to find advice on how to correct the problem.

With more complicated programs, store the necessary commands in a disk file to avoid retyping the commands each time you need them. The commands in these files have the same format as any other commands. Staying with our example program, you can create a file named TEST.LNK by entering:

```
OUTPUT TEST.EXE
FILE TEST.OBJ
LIB MATHLIB.LIB
MAP = TEST
```


Then link the program by putting an at-sign (@) in front of the filename:

```
PLINK86 @TEST(cr)
```

Unless you specify a different extension, PLINK assumes that the extension of an @ file is .LNK. These @ files can be used at almost any point during command input, and each can invoke other @ files up to three levels deep. This lets you create a complicated overlay structure for a library file and then use it within many different programs.

The linkage edit we've entered produces a memory map report on disk called TEST.MAP. There are a variety of formatted reports available. (See Section 4.3 for details.) The command file also names a library to be searched. Library files have the extension .LIB. Any modules needed by the TEST module are selected automatically.

When linking extremely large programs, PLINK may not have enough primary memory to store the description of the program. When this happens, the message:

```
Out of memory, opening work file
```

appears on the screen. PLINK then opens a work file on the logged disk, and stores a portion of the program description in that file. The logged disk should not be removed until the linkage edit is complete.

PLINK FILES 3.3

Your PLINK software includes several files not yet discussed. Some of these aren't needed to operate PLINK; they're included as a convenience to software developers.

The PLINK software consists of these files:

- PLINK86.EXE: The linkage editor program.
- CHECKSUM.EXE: A utility program that validates the checksum of any .EXE file. Use it if you suspect a program has been damaged.

  To run CHECKSUM, enter the name of the file followed by the name of the file to check, as in this example:

  ```
  CHECKSUM BIGFILE.EXE
  ```

- PLTEST.LNK, PLTEST.LIB: A PLINK test program with overlays.
- OVERLAY.LIB: An object file that contains run-time overlay loaders. OVERLAY.LIB must be on the logged disk when you link a program that contains overlays. PLINK automatically selects the appropriate overlay loader to include in the program when overlays are used.
- COMPARE.EXE: A utility that compares two files byte-by-byte. When differences are encountered, COMPARE lists the bytes from the first file with the corresponding bytes from the second file underneath. You can specify starting address and size to compare a portion of the files, as in this example:

  ```
  COMPARE FILE1 FILE2
  COMPARE F1 F2 START 21C5 SIZE 20
  ```

  Numbers are assumed to be hexadecimal unless followed by a period to indicate decimal. The key words START and SIZE can be abbreviated by leaving characters off the end.

- DUMP.EXE: A utility program that dumps a file in a readable form to the screen or a disk file. To get a listing on the screen, type:

```
DUMP filename
```

To put the dump into a disk file, type:

```
DUMP filename DISK
```

The name of the disk file is the name of the output file with the extension .LST.

The program selects a dump format according to the file extension.

.EXE

If the file has the .EXE extension, the header and all base fixups are printed. To prevent the fixups (there can be thousands of them in a large program with long calls) use the NOFIXUP option:

```
DUMP filename.EXE NOFIXUP
```

.OBJ or .LIB

With .OBJ and .LIB files, object files are dumped showing the Intel object format. If the file is a library, each module in the library is dumped. You can select a single module by giving the module name, or a public symbol defined in the module and present in the library index:

```
DUMP LC.LIB MODULE XIOS
DUMP LC.LIB SYMBOL PRINTF
```

Particular object file records are included in or excluded from the report by giving the Intel record names or hex record type-numbers:

```
DUMP FOO.OBJ INCLUDE THEADR
DUMP FOO.OBJ EXCLUDE 0A0, 9C
```

If PLINK does not recognize the file extension, the file is dumped in HEX, with the ASCII interpretation on the right-hand side of the line.

To force a HEX dump, or any other kind of dump, specify the type:

```
DUMP LC.LIB HEX DISK
DUMP TEST.FOO EXE NOFIX
```

When the format is HEX, the starting address and size can be specified in hex:

```
DUMP TEST.FOO START 1200 SIZE 80
```


When using any of these utility files, remember:

- The file type of the input file defaults to EXE.
- Numbers must begin with a digit 0–9.
- Numbers are assumed to be hexadecimal unless you end them with a period for base 10.
- Option names can be abbreviated by leaving off characters at the end.

4

PLINK COMMANDS

This chapter discusses the various PLINK commands. The commands are grouped according to their general functions. The groups are:

- Input format
- Output files
- Memory map
- Object files
- Defining program structure
- Overlays
- Miscellaneous commands

INPUT FORMAT 4.1

This section describes some basic input elements. Later sections show how these are combined to create full statements.

16-BIT VALUES 4.1.1

You can express a 16-bit value as a number in any of several bases, as shown in Table 4-1. An optional radix character immediately following a number indicates which number system is used.

Table 4-1: Bases for 16-Bit Values

BASE	RADIX	VALID DIGITS	VALID RANGE
Hex	H	0–9 , A–F	0–0FFFFH
Decimal	.	0–90	0–65535
Octal	O	0–70	0–1777770
Binary	B	0 and 1	16 digits

Use hexadecimal (H) representation if a number does not have a trailing radix character.

Here are some examples of valid 16-bit values:

1417O 0C1B5 55. 11B 11BH

Here are some invalid 16-bit values:

96O	contains an invalid octal digit.
C1C2	does not begin with a digit.
12345H	is too large.

4.1.2 IDENTIFIERS

An identifier is the name of an object, such as a module or segment. A simple identifier is a sequence of 20 characters or less that does not contain spaces or any of these special characters:

^ = ; < > / , \ ! ' # & * + - : @ DEL

Lowercase letters in identifiers are automatically translated into uppercase. The first character of an identifier cannot be a digit.

You can avoid these restrictions on valid identifier characters by using the escape character (^). The character immediately following the escape character is treated as a normal identifier character. If you want to include the escape character itself in an identifier, type two escape characters (^^).

Here are some valid identifiers:

ProgramI **SORT3** **ABC^@**

These are not valid identifiers:

34ABC	Begins with a number.
NIM A	Contains a space.
PROG%1	Starts a comment with a percent sign.

You can make these invalid identifiers valid by including the escape character:

^34ABC **NIM^ A** **PROG^%1**

Identifiers that appear in object files are truncated (cut off on the right) to 50 characters for symbol names, or 12 characters for other names (such as groups, classes, and segments). Truncation both saves memory and makes it easier for PLINK to compare these names with other identifiers in your program. Identifiers may be truncated again when included in memory map reports.

DISK FILENAMES 4.1.3

PLINK uses the filename format of the operating system under which it is executing. A filename is terminated when you enter a character that cannot be legally included in a filename. The escape character can be used to put any character into a filename.

This manual uses MS-DOS filenames when discussing how to use PLINK. These filenames have the form:

< d > : < filename > . < ext >

where:

< d > is a one-letter drive or device specifier.

< ext > is a three-letter extension that specifies the file type.

The device specifier and the extension are optional.

Here are some examples:

```
PROG1.EXE
B:CHESS.OBJ
SCANNER
```

If a device is not given, PLINK assumes that you want to use the logged disk.

4.1.4 STARTING PLINK

You can use PLINK interactively, or you can give input as it is executed. All input uses this format:

PLINK86 < statements > (cr)

where (cr) means you must press the Return key.

To use PLINK in the interactive mode, type:

PLINK86(cr)

PLINK then reads statements from the keyboard, prompting you with an asterisk (*) each time more input is needed. All input is stored uninspected until you type a carriage return. Standard line-editing features supplied by the operating system are available.

A disk file that contains all or part of a command can be inserted into the input at any point. Do this by entering an @ followed by the name of the command file, as in this example:

```
PLINK @COMFILE(cr)
```

PLINK assigns the default extension .LNK unless you specify otherwise. A disk file can contain up to three additional levels of @ specifications.

This feature is most often used to prepare a file containing a complete command. Usually, .LNK files are prepared once for a given program and then used over and over, simplifying the whole process.

PLINK reads an entire command, checking for syntax only, before processing files.

COMMAND FORMAT 4.1.5

All PLINK input is free format. Blank lines are ignored, and a command can use any number of lines. You can include comments by preceding them with a percent sign. When PLINK encounters the percent sign, all remaining characters on the same line are ignored.

Input takes the form of a list of statements. Each statement begins with a key word, and many statements are followed by arguments separated by commas. In this statement:

```
FILE A,B,C
```

FILE is the key word, and A, B, and C are the arguments.

Key words can be shortened by omitting trailing characters, as long as the resulting abbreviation is unique. For instance, the previous statement can be entered as:

```
FI A,B,C
```

For other types of errors, PLINK displays an error message as an error code. Appendix C explains the PLINK error messages and error codes.

If a fatal error occurs, PLINK terminates after displaying the error message. Re-run PLINK after you correct the error.

4.2 OUTPUT FILE

The OUTPUT command specifies the name of the file that will hold the linked program. The extension must be .EXE. If you do not specify an extension, PLINK assigns .EXE by default. The output file is written over any existing file with the same name.

Here are some files specified with the OUTPUT command:

```
OUTPUT PROG1
OUTPUT PROG2.EXE
OUTPUT PROG3.EXE
```

4.3 MEMORY MAP

You can use the MAP statement to obtain various reports describing the output of the linkage edit. You can choose reports that show the memory addresses assigned by PLINK to the sections, segments and symbols in the linked program, or that describe the modules that were included.

The format of the MAP statement is:

MAP [= <filename>] <flag1>,... ,<flagn>

where:

<flag1>,... ,<flagn> select the report you want. These can be:

G Global symbols. This report lists all public symbols of all loaded modules. The symbols are listed in alphabetical order with their assigned addresses.

S Sections. All program sections are listed in input order. The assigned disk and memory addresses, size, and other information is given for each section.

A All. This report lists the program sections in input order. The segments of each section are listed in order of ascending memory address.

M Modules. This is the largest report. Each module is listed in input order, along with the segments it contains, the segment addresses, and their sizes. The symbols contained in each segment are also listed. Common blocks are listed separately because they are not really a part of any single module.

If you do not specify a flag, PLINK uses MAP G by default.

Memory map reports usually appear on the screen. However, they can be written to any file by typing an equal sign and then a filename. In this example:

MAP = BIGFILE

a report listing all global symbols is written to BIGFILE.MAP. (The default extension is .MAP.)

Public segments with the same name located within the same section appear as a single combined entry in the S and A reports. The size of the entry is the sum of the individual segment sizes plus amounts needed for alignment.

In the map reports, segment names are given as the segment name followed by the class name, with a period separating the two. This name is truncated if necessary to fit the allotted number of columns.

4.3.1 WIDTH

This statement sets the page width of the memory map reports. For example:

```
WIDTH 132
```

sets the page width to 132 characters. The report generators change the page width by changing the number of columns per line.

4.3.2 HEIGHT

This statement sets the page height of the memory map reports. For example:

```
HEIGHT 80
```

sets the page height to 80 characters. The report generators change the page height by changing the number of lines per page.

4.4 OBJECT FILES

PLINK uses a single object file format—the same format used by the MS-FORTRAN and MS-Pascal compilers. This is the Intel 8086/8088 relocatable object module format as modified by Microsoft for faster library searches. Several other compilers also use this format.

FILE, LIBRARY, AND SEARCH 4.4.1

The FILE, LIBRARY, and SEARCH statements define the object files and libraries to be used as input to the linkage edit. Each statement is followed by a list of filenames, separated by commas:

```
FILE MAIN, PASS1.OBJ, PASS2
LIBRARY LC.LIB, APPLIB
SEARCH PASLIB.LIB
```

The default extension for these files is .OBJ.

With the FILE statement, all modules in the listed files are included in the output program.

If you use the LIBRARY statement, the only modules selected are those that contain the definitions of symbols needed to define external symbols in modules that are already linked. This selection process is known as a "library search," and is commonly used for the run-time support libraries supplied with most compilers. A library search reduces the size of a linked program because only those parts of the run-time support actually needed are loaded.

The SEARCH statement is the same as the LIBRARY statement, except that PLINK makes multiple passes through the file if undefined symbols remain after all specified files are read. SEARCH is rarely used, but is useful when you can't pull all the needed modules out of the library in one pass. This can occur when you are searching two libraries that can contain symbols defined in the other library.

You can use the same filename twice in a FILE, LIBRARY, or SEARCH statement, but a duplicate symbol error will probably occur if the same module is loaded twice.

If PLINK can't find an object file you've asked for, it looks on drive A. If the file is not on this drive, PLINK asks for the name of another drive to search. You can change diskettes at this time if necessary. Make sure that the diskettes you remove do not contain open files. (During pass 2 of the linkage edit, for instance, the output file is open.

Also, if PLINK runs out of memory, a work file is opened on the default disk. You must not remove that disk.)

PLINK accepts a PATH name as part of an object filename. If an object file can't be found, PLINK looks for a string named OBJ in the environment and puts that string's value onto the front of the file name, after stripping any drive ID. For example, suppose that you enter:

```
SET OBJ = \OBJECT
```

and then run PLINK using this command:

```
FILE  B:TEST.OBJ
```


If TEST.OBJ doesn't exist on drive B, PLINK strips the B: from the name and tries \OBJECT\TEST.OBJ to obtain the requested file.

Usually you set up a directory containing libraries and other commonly used object files. These libraries and files can then be automatically linked into any program in the system.

4.5 DEFINING PROGRAM STRUCTURE

This section describes PLINK statements that define the organization of module segments into sections.

Normally, segment classes are allocated memory in the same order that they appear in the input files. Within each class, segments are allocated memory in the order that the segment names are encountered.

For instance, suppose the first module linked defines the following segments:

Class	Name
PROG	BASE
DATA	PROG
PROG	PROG
DATA	STACK

The segments will be assigned memory in the order BASE, PROG, DATA, and STACK. The two segments in class PROG are first, followed by those in class DATA. Whether or not a segment is in a group has no effect on this canonic definition of segment ordering.

Because of the canonic ordering, many compilers supply a "header file". This file has to be linked first; it defines all segments the compiled program will use. The order they are given in thus defines the memory allocation order for all segments in the program.

Other compilers require the user to link the main module first, and place the necessary segment definitions within it. Still others output the required segment definitions with each module compiled. These could be linked in any order.

Plink86 follows the canonic segment ordering unless it is deliberately altered with the CLASS command. When the program contains more than one section, the segments within each section are ordered canonically (rather than trying to use the canonic order over the entire program). For example, if two sections contained segments in class PROG and DATA, the PROG segments would be first in each section, followed by the DATA segments.

4.5.1 SECTION

The SECTION command creates a new program section. Segments from all files specified after the SECTION command are loaded into the new program section by default.

The format for the SECTION command is:

SECTION [= <section name>]

where:

<section name> is an optional element used to refer to the section in subsequent commands.

Segment classes are normally allocated memory in the order in which they appear in the input files. Regular files are processed first, followed by library files. A simple, none overlaid program usually has only one section containing all the classes in order.

Here are some examples:

```
SECTION =Global
SECTION
```

Plink86 will automatically create a new section when necessary. If, for example, you write a FILE or LIBRARY statement without a preceding SECTION statement, Plink86 assumes the SECTION statement and, if one is not already available, creates one.

All program overlays are sections. SECTION, then, must also be used to create an overlay, or any other portion of the program that is loaded from disk as a separate unit. For more information, see Section 4.6.

The CLASS command moves some or all of the segments in a class to the currently defined section. There are two formats.

The first format is:

CLASS < **classname** >

All segments in the specified class are moved to the current section by default.

The second format is:

CLASS < **classname** > **(seg1, seg2, ..., segn)**

where:

seg1, seg2, ..., segn are the segments to be moved to the current section.

Nothing happens to the other members of the class. (The class name is used only to identify the segments to be moved.) This form of the CLASS statement overrides all other specifications.

Here are some examples of the CLASS command:

```
CLASS MEMORY
CLASS DATA (OV1DATA, OV2DATA, COMM5)
```

The classes and segments specified in CLASS statements become part of the canonic segment ordering of the program. These classes and segments appear in the ordering before those defined by program modules. This happens because Plink86 processes all input statements before any object files are read. Therefore, even if the class statement is used only to move something from one section to another, one must be careful to restore the canonic order to what it should be.

4.5.3 MODULE

The MODULE command assigns all segments from the given modules to the current section. It is overridden only by the CLASS statement.

The format for the MODULE command is:

MODULE <**module1**>, <**module2**>, ... <**modulen**>

where:

<module1>, <module2>, ... <modulen> are the modules to be included in the current section, separated by commas.

MODULE is most often used to select modules from within a library. A typical library manager program gives a module the same name as the object filename when the object file is inserted into a library.

You can find the names of the modules in an object file by using the DUMP utility. For example, this statement:

```
DUMP BIGFILE INC THEADR
```

tells you the names of the modules in BIGFILE.

Determining module names can be tricky. FORTRAN compilers typically give modules whichever name you specify. Other compilers, however, often give each module the same name, making it impossible to use the MODULE command. With these compilers, you should use the FILE command.

GROUP 4.5.4

The GROUP command is used to move all segments within the specified group to the current section. GROUP overrides the section assignment given by the FILE statement, but does not override the MODULE or CLASS statements.

The format of the GROUP command is:

GROUP <**group1**>, <**group2**>, ... <**groupn**>

where:

<group1>, <group2>, ... <groupn> are the names of the groups to be allocated in the current section.

Here is an example of the GROUP command:

```
GROUP DGROUP, COMGRP
```

You can make addressing easier by creating a section which contains only members of a particular group. The command would look like this:

```
SECTION GROUP DGROUP
```

COMMAND PRECEDENCE 4.5.5

As the preceding discussion has shown, some commands can override other commands. Here is a list of the preceding commands in order of precedence (highest precedence first):

1. CLASS (with segments)
2. CLASS (without segments)

3. MODULE

4. GROUP

5. FILE, LIBRARY, SEARCH

FILE, LIBRARY, SEARCH, and SECTION statement are generally used to structure the program segment into sections. These statements are then overriden by GROUP, MODULE, and CLASS statements to place the appropriate entities into overlays, to ensure that groups can be addressed properly, and to restore the cononic segment ordering.

4.6 OVERLAYS

Overlays reduce a program's memory requirement. Each overlay is a program section that uses the same memory area as another section. Two overlays do not generally use the same memory area concurrently; an overlay normally destroys any earlier overlay in its memory area.

When you use overlays, PLINK adds an overlay loader module to the program. The loader automatically reads overlays from disk into memory as required by the executing program. Memory requirements are reduced because parts of the program share the same memory space while the program is running. Overlays increase execution time, however, because of the extra time needed to load the overlays from disk.

No explicit calls to load overlays are required in the source code of the application program; the overlays are loaded automatically by an overlay loader supplied with PLINK. The application program can also call the overlay loader and pass the number of the overlay to be loaded.

If you have unusual requirements, see Appendix A for instructions on writing custom overlay loaders.

BEGINAREA AND ENDAREA 4.6.1

BEGINAREA and ENDAREA (abbreviated BEGIN and END) define overlay structures. You can usually create complex overlay structures without modifying the program modules as long as you obey some rules concerning calling sequences and accessing data. The overlay structure should be organized to minimize the number of times overlays have to be loaded. If it takes .05 second to load an overlay, a program loop that switches overlays each of the 100,000 times the program executes will run for almost three hours. Be careful.

An overlay area is a group of overlays (sections) that share the same memory address. The BEGIN statement is used without arguments to create an overlay area beginning at the current memory allocation address. Each overlay area must be ended by an END statement. Any sections between the BEGIN/END pair automatically become overlays.

This example:

```
OUTPUT TEST
FILE F1
BEGIN SECTION FILE F2
      SECTION FILE F3
      SECTION FILE F4 END
```

creates a program named TEST.EXE. When TEST.EXE is executed, the code in file F1 is loaded. This code is the root section of the program that is resident in memory at all times. Then, execution begins. Suppose a call to some code is made from file F2. The overlay loader is automatically invoked to read F2 from the disk, and then a branch is made to the called routine. If a call is then made to F3, F3 is loaded, overwriting F2 in the process. If F2 is called again before F3 or F4, no disk I/O occurs because the required overlay is still in memory.

An overlay is not written back to disk when replaced by another overlay. A fresh copy is loaded the next time the overlay is needed; the disk file is never modified.

PLINK arranges for overlays to be loaded automatically by replacing the address that points to an overlaid routine with a call to a small piece of code that calls the overlay loader and then jumps to the overlaid routine. (This piece of code is called the "overlay vector.") This ensures that the correct overlay is in memory. Calls to an overlaid routine can be long or short. The overlay loader uses the program's stack area, but the stack and all machine registers are returned to their previous state before returning to the program.

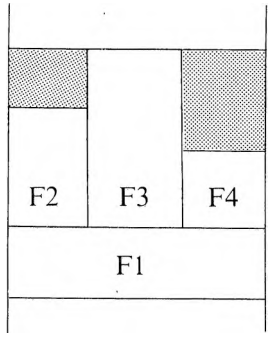
A program can either call or jump to an overlaid routine. However, you must make sure that the caller is still in memory when you make a return to the caller. Overlays are not loaded automatically when a routine returns to its caller. Using the previous example, if F2 calls F3, F2 is replaced by F3 before the return to F2 is made. F2 can jump to F3 with no problem, however.

PLINK cannot tell if an address in an overlay represents code or data. If F2 references F3, PLINK assume the reference is a call or jump. Data can't be accessed from one overlay to another in the same area.

Figures 4-1, 4-2, and 4-3 are memory diagrams used in the following discussion of overlay structures. In these diagrams, the vertical dimension represents memory addresses, with lower addresses at the bottom. The horizontal dimension is used to indicate where memory locations are shared.

In Figure 4-1, F1 reaches all the way from left to right because it does not share its memory (F1 is resident). F2, F3, and F4 share the same memory space within the overlay area. Since F3 is larger than F2 or F4, some memory is unused when these overlays are in memory. These wasted areas are shaded.

Figure 4-1: Simple Overlay

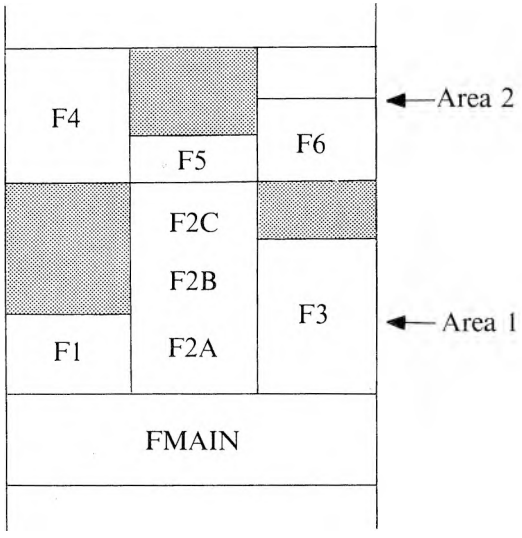


More than one overlay area can be created. Each additional one is allocated to the next available space in memory, as shown by this command:

```
OUTPUT TEST2
FILE FMAIN
BEGIN SECTION FILE F1
      SECTION FILE F2A, F2B, F2C
      SECTION FILE F3 END
BEGIN SECTION FILE F4
      SECTION FILE F5
      SECTION FILE F6 END
```

This command creates two independent overlay areas. One overlay from the first and one from the second can be in memory simultaneously, as shown in Figure 4-2.

Figure 4-2: Two Independent Overlays

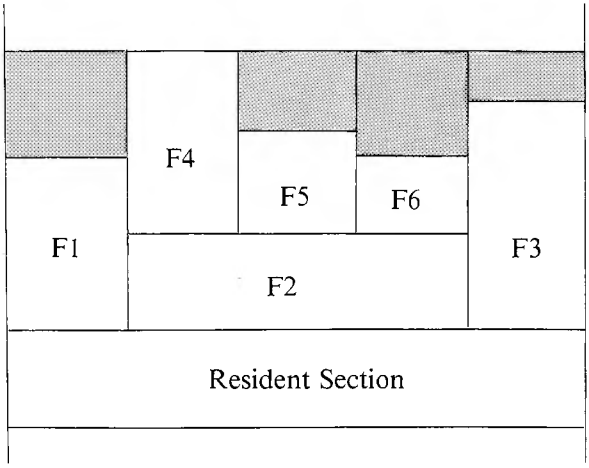


You can also create overlay structures by nesting BEGIN and END statements up to 32 levels deep, as shown in this example:

```
OUTPUT TEST3
FILE FMAIN
LIB FLIBR
BEG SECT FILE F1
    SECT FILE F2 BEG SECT FILE F4
                     SECT FILE F5
                     SECT FILE F6
                END
    SECT FILE F3
END
```

Each section of a program is assigned a "level number." This number tells you how many levels of overlay exist at a particular section. A resident section always has a level number of 0. Whenever a BEGIN command is entered, the level number of a section increases by one. The level number decreases by one at an END statement. In Figure 4-1, for example, F1 is level 0, while F2, F3, and F4 are level 1. In Figure 4-3, F1 through F3 are level 1, and F4 through F5 are at level 2.

Figure 4-3: Overlay Structure with Nested BEGIN/END Pairs



Each overlay has at least one "ancestor." An ancestor is the last section defined before the beginning of the overlay area. The ancestor also has a lower level number than the overlay area. In Figure 4-1, F1 is the ancestor of F2, F3, and F4. In Figure 4-2, the global section is the ancestor of all the overlays. In Figure 4-3, FMAIN is the ancestor of F1 through F3, while F2 is the ancestor of F4 through F6.

The ancestors of an overlay consist of the overlay's ancestor, the ancestor of that overlay, and so on until a resident section is reached. The number

of ancestors an overlay has is equal to its level number. In Figure 4-3, the ancestors of F5 (which is at level 2) are F2 and FMAIN.

A section can also have "descendants." Descendants are all those sections that share the same section as an ancestor. In Figure 4-3, the descendants of F2 are F4, F5, and F6.

Now that the terminology is established, here is the rule for accessing data within overlays:

- Data within an overlay can be accessed only from within the overlay, or from one of the overlay's descendants.

For instance, data within F2 can be accessed only from within F2, or from F4, F5, or F6. The data within F1 can be accessed only within F1, since the section has no descendants.

PLINK and the overlay loader guarantee the data access rule by obeying these additional rules:

- When an overlay is in memory, all of its ancestors must be in memory as well. In Figure 4-3, if F6 were called without a call to F2, F2 is automatically loaded anyway.
- Overlay vectors are never used when an overlay accesses an ancestor. Instructions that access data in this direction are not modified. Procedure calls execute at full speed as well.

The simplest way to avoid complications when accessing data is to put all data in a separate section. Most available compilers use a single 64K physical segment for data. If the data is scattered throughout the overlay structure it can be impossible to stay within this 64K restriction. Many compilers also require the data segment to be at the end of the program so the free memory above the program can be used as heap space.

Nested overlay structures are useful when a set of routines is used only by certain overlays. In Figure 4-3, for instance, F2 would typically contain routines used by F4 through F6, but not by F1 or F3.

SEPARATE OVERLAY FILES 4.6.2

Resident portions of a program and all overlays are normally written to the same output file. Because the overlays appear at the end of the file, the operating system is unaware of their presence when loading the program. There are two advantages to this:

- It decreases the execution time of your program. No directory search is needed to find an overlay, because the program file is opened only once at run time.
- It makes it hard to confuse overlay files from different versions of a program.

Sometimes, however, it is useful to put overlays into separate disk files. If a program is linked into a single file, for example, it might not fit on a single disk.

Add the INTO option to the SECTION statement if you want to put an overlay into a different file. INTO is followed by the name of the file where the overlay should be placed. If you do not specify an extension, PLINK uses the default .OVL.

If we add the INTO option to the program discussed with Figure 4-1, this statement:

```
OUTPUT TEST
FILE F1
BEGIN SECTION INTO TEST1 FILE F2
      SECTION INTO TEST2 FILE F3
      SECTION FILE F4 END
```

writes the program into three disk files. Files F1 and F4 are in TEST.EXE. File F2 is in TEST1.OVL, and F3 is in TEST2.OVL.

If you do not use the INTO option, a section is written to the file of its ancestor, or to the main output file if the section is at level 0. The same overlay filename can be mentioned more than once if needed.

4.6.3 SELECTING AN OVERLAY LOADER

The overlay loader library must reside on disk in a file named OVERLAY.LIB. PLINK searches this file whenever necessary, so you should not include this filename in a FILE, LIBRARY, or SEARCH statement. PLINK looks in the directory specified in the OBJ environment string if OVERLAY.LIB is not found in the current directory.

OVERLAY.LIB contains several overlay loaders. PLINK selects an appropriate one based on your operating system and other factors. You can select a debugging version of each loader by using the DEBUG statement. The DEBUG statement displays a screen message whenever an overlay is loaded at run-time. If you already have a .LNK file of an overlaid program, for example, you could type:


```
PLINK86 @TEST.LNK DEBUG
```

to get a version with debugging messages.

4.6.4 ALWAYS AND NEVER

Sometimes PLINK's strategy for determining when an overlay vector should be used for symbol access is incorrect. In these cases, you can override the default choice.

Suppose, for example, that you're linking a program in which the address of a function is passed as an argument to a function in another overlay which calls the original function. PLINK doesn't know that the original function is being called from another overlay and, as a result, might not use the vector. If you use the ALWAYS command, PLINK is forced to use the vector for all references to the symbol.

The format for the ALWAYS command is:

ALWAYS <**sym1**>, <**sym2**>, ... , <**sym n**>

Problems also occur when a data area is to be accessed from an ancestor of the overlay that contains the data area. By default, PLINK assumes that a function is being accessed and uses the overlay vector. Use the NEVER command to make all access to the symbol use the real address.

The format for the NEVER command is:

NEVER <**sym1**>, <**sym2**>, ... , <**sym n**>

FINDING THE OVERLAYS 4.6.5

None of the operating systems under which PLINK-created programs can run offer a way for an executing program to find out which disk file it was executed from. Because the program overlays are usually in the same file, the overlay loader must somehow find this file and open it. If the overlays are in one or more separate files, the overlay loader must find these files as well. With PLINK, this process is usually automatic, but occasionally you may be asked to tell the program where the overlays are (see Section 4.6.6).

PLINK supplies the overlay loader with a common block containing the names of all files making up the program, as well as all other overlay information (see Appendix A for details). These filenames consist of name and type fields only; the drive ID (and directory name) is stripped.

When looking for an overlay file, the overlay loader first checks the logged drive (within the current directory) for the required file. If this fails, the overlay loader tries the internally-set "overlay drive." The overlay drive is initially drive A, but it can be changed as described later. If the overlay file is not on the overlay drive, the PATH currently set for executable files is tried.

If all else fails, PLINK asks you to enter the drive ID where the overlay can be found (see Appendix B for exact messages). You can change diskettes at this time if necessary, making sure that none of the diskettes removed have open files on them. The drive ID you enter becomes the new overlay drive and is used to find overlays needed later.

The easiest approach to use when disk space is adequate is to put the program file and all overlay files into the directory specified by the PATH string in the environment. This is probably where you keep most other programs anyway; using the specified directory lets PLINK load the overlays without bothering you.

Another approach to finding an overlay is to have the application program invoke the overlay loader and handle the "no overlay" situation directly. This process is described next.

4.6.6 DIRECT OVERLAY LOAD

Sometimes you need to call the overlay loader directly from the application program. If you have used NEVER to override the automatic overlay loading associated with a symbol in an overlay, the associated overlay must be loaded manually before calling the symbol. A manual load also lets the application program handle the "overlay file not found" situation itself. This can be done by having the program display a message suggesting ways to correct the problem.

You must use assembly language to call the overlay loader directly. Assembly language gets around the fact that different compilers use incompatible methods to pass parameters to procedures and return function results. This is the code you need for MS-DOS:

```
extrn $LOAD$:far
  ...
  ...
mov AH,<parameter 1>
mov AL,<parameter 2>
mov CX,<# of overlay to load>
call $LOAD$
jc <error routine>
```

The $LOAD$ routine returns with the carry flag set if the load was unsuccessful.

Parameter 1 (if nonzero) is a flag that tells the overlay loader to return to the caller if an overlay can't be found. In this case, the carry flag is set; otherwise, an error message appears and the program terminates.

Parameter 2 (if nonzero) is a flag that tells the overlay loader to ask for the name of the drive on which an overlay is to be found. If parameter 2 is zero, the loader searches the usual place and then aborts the program or returns to the application, depending on parameter 1. If parameter 2 is nonzero, you can enter the drive name, or you can enter a period to abort. If you decide to abort, and parameter 1 is set, the loader returns to the application program with the carry flag set.

The overlay number is determined by the order in which overlays are defined in the input to PLINK, regardless of the level of the overlay. The first overlay is level 1. If you are unsure how the numbers are assigned, use the MAP command and the S (or A) report, as described in Section 4.3.

If the overlay being loaded has ancestors, the ancestors are loaded as well. After the load is finished, routines in the overlays are called in the normal fashion. The carry flag is set if the overlay loader is unable to load one of the ancestors.

If you want the overlay loader to check a different disk or PATH for an overlay, change them programmatically before calling $LOAD$.

4.7 MISCELLANEOUS COMMANDS

4.7.1 DEFINE

This statement can be used to give values to symbols not defined by any module in the program. These symbols are then used to resolve EXTERNAL symbol references made by the program modules. The symbols can be given absolute values (with or without a base address) or they can be defined as a plus- or minus-offset to some other symbol. For example:


```
DEFINE VERSION = 3, FLAGS = 10110011B,
S1 = 100:10B, S2 = S1, S3 = S2 + 5
```

When an absolute symbol is defined (like S1), use a colon to separate the base and offset components of the address, with the base appearing first. If only one number is given (like VERSION), PLINK assumes that number is the offset and sets the base address to zero.

4.7.2 VERBOSE

The VERBOSE command causes PLINK to maintain a status line at the bottom of the screen that tells you what operation is being carried out. This is helpful during a long linkage edit.

The format of VERBOSE is:

```
VERBOSE
```

5

PLINK EXAMPLES

This chapter describes more complex uses of PLINK by showing how to create programs under the MS-DOS operating system. An MS-DOS program is specified by using a filename with the .EXE extension in the the OUTPUT statement:

```
OUTPUT  TEST.EXE
```

The .EXE extension is the default.

PLINK COMMANDS FOR MS-DOS FILES 5.1

Two commands are used only on .EXE files. After these are discussed, we'll see how to use PLINK with three MS-DOS compilers.

DSALLOC 5.1.1

DSALLOC changes the address of DGROUP to be the size of the group minus 10000H. All items referenced relative to DGROUP have offsets as though DGROUP were moved to the high end of the physical segment used to hold the data area. DSALLOC does not take arguments.

DSALLOC is used when linking MS-FORTRAN and MS-Pascal programs. At execution, the run-time systems of the MS-FORTRAN and MS-Pascal compilers physically move the DGROUP segments to high memory before any references to them are made.

5.1.2 HIGH

HIGH sets the "high switch" in the .EXE file header to zero. (Normally this field contains FFFFH.) This causes the program to be loaded as high as possible in memory when executed. You do not usually need to use the HIGH statement—the run-time support libraries of most compilers use free memory above the program as heap space.

5.2 USING PLINK WITH COMMON MS-DOS COMPILERS

5.2.1 LATTICE C

Under MS-DOS, the Lattice C compiler requires that header file C.OBJ be linked first, and that library LC.LIB be searched. The STACK segment in class DATA must appear at the end of the program because the heap begins there.

This example shows how to link a non-overlaid Lattice C program:

```
OUTPUT TEST.EXE
MAP = TEST.LST G
FILE C, F1, F2, F3, F4
LIB LC.LIB
```

The example links the program and produces a memory map on disk that contains an alphabetic list of all symbols and their addresses. The STACK segment automatically appears last because it is defined last in the C.OBJ module.

When overlays are used, you must explicitly move the STACK segment to the end of the program. The resident portion of the program comes first, and the overlays are in-between. Here is an example:

```
OUTPUT TEST.EXE
MAP = TEST S
FILE C, F1, F2, F3
LIB LC
BEGIN  SECTION FILE F4, F5
       SECTION FILE F6, F7, F8
       SECTION FILE F9
END
SECTION = DATA GROUP DGROUP
```

Use BEGIN and END to define an overlay area where all three sections share the same memory space. If you have not created a necessary section, PLINK does so automatically. (This is why there is no SECTION command at the front of the input.) The overlay sections in the example do not have names, but the DATA section does. The GROUP statement moves everything in DGROUP to the DATA section at the end of the program's assigned memory area. The STACK segment is located at the end of this.

The DATA section must be less than 64K in length. If it is longer, PLINK gives warning messages about offsets being too large. (Lattice C uses 16-bit offsets to access the data area.)

When the program is executed, MS-DOS loads only the first section. The DATA section is marked to be preloaded by the overlay loader, and is loaded as execution of the program begins. This is apparent in the memory map, which is simply a list of all sections in this example.

In the previous example, the overlays contain only code; all the data is in the DATA segment. To overlay the data as well, the last line of the command can be changed to:

```
SECTION = DATA CLASS DATA (STACK)
```

or:

```
CLASS DATA (STACK)
```

This puts only the STACK segment at the end of the program; the other data segments are put with the code segments, allowing the data to be overlaid. The distance from the lowest-addressed data segment in the program to the end of the highest must still be 64K or less, however. The same is true for the code segments.

5.2.2 MS-FORTRAN AND MS-PASCAL

With these compilers, the FORTRAN.LIB or PASCAL.LIB library must be searched. In addition, the data segments must be allocated to high memory (for an explanation of this concept, see the DSALLOC command in Section 5.1.1).

The following examples are for MS-FORTRAN, but the statements for MS-Pascal are identical. Here is an example of a simple link:

```
OUTPUT TEST.EXE
MAP = TEST S
DSALLOC
FILE F1, F2, F3, F4
LIB FORTRAN
```

The program is linked and a memory map listing all modules in the program and the segments in each is produced. Common block segments appear in a separate report.

When overlays are used, all data segments (including some not in DGROUP) must be moved to the end of the program in a specified order. The main program appears first, then the overlays, then the data segments, as shown in this example:

```
OUTPUT PROGRAM.EXE
MAP = TEST A
DSALLOC
FILE F1, F2, F3
LIB FORTRAN
BEG     SECT FILE F4, F5
        SECT FILE F6, F7, F8
        SECT FILE F9
END
CLASS   MEMORY,STACK,DATA,COMADS,
        CONST,COMMON,HIMEM
```

When the program is executed, MS-DOS loads only the first section. The data section at the end is marked to be preloaded by the overlay loader, and is loaded as execution of the program begins. This is apparent in the memory map, which is a list of all sections in the program and the segments contained in each.

Use the CLASS command to move all the data segments to the end of the program in the order required by the MS-FORTRAN run-time system. MS-FORTRAN does quite a bit of processing before execution of the application begins, making it difficult to use other structures for an overlaid MS-FORTRAN program.

PL/M-86 AND ASM86 5.2.3

Under MS-DOS, the PL/M-86 compiler and ASM86 assembler require that you link the file EXEBASE.OBJ with your object modules and libraries to create an .EXE file. The CONST segment should appear before the DATA segment.

This example shows how to link a non-overlaid PL/M-86 program:

```
OUTPUT FOO
MAP = FOO.MAP M, A, G, S
FILE EXEBASE, FOO1, FOO2, FOO3
LIB FOO.LIB
CLASS CODE, CONST, DATA, STACK
```

You can also choose to use the Intel linker, LINK86, to produce .LNK files, and then use PLINK to turn these files into an .EXE file, as shown below:

```
OUTPUT TESTPROG
FILE EXEBASE, TESTPROG.LNK
CLASS CODE, CONST, DATA, STACK
```

A

OVERLAY LOADER

This appendix contains technical information about the overlay loader for programmers who want to write their own overlay loader (not recommended unless absolutely necessary) and for those who just want to know more about how PLINK works. None of the information here is needed to use the overlay loader in the usual way.

ENTRY POINTS A.1

At link time, the overlay loader is selected from a library of overlay loaders for MS-DOS. The specific overlay loader selected depends on the main entry point called by the symbol vectors. The name of the entry point is $OVLY??; the last two characters indicate one of these overlay loaders:

- $OVLYM: Standard overlay loader for MS-DOS.
- $OVLYMD: MS-DOS overlay loader with debugging messages.

In addition to one of these entry points, the loader contains the following symbols:

- $OVIN$: The initialization entry point. This is always the starting address of the program when the overlay loader is present. It initializes the loader and then jumps to $STRT$, the starting address of the application. This is a long jump.
- $LOAD$: A routine that loads the overlay whose number is in CX, callable from the application program. If AH is nonzero, the loader returns to the application if the overlay isn't found on disk. Otherwise, program execution terminates and an error message appears. If AL is nonzero, the overlay loader may ask you to enter the drive containing the overlay file. Otherwise, it doesn't ask, and an error is returned to the application program. An error is indicated when the

carry flag is set.

- $OVEX$: The exit point from $OVLY??, used for debugging. Breakpoints can't be set in overlays until after they are loaded. Sometimes it is helpful to break at the address of this symbol. It points to the return back to the overlay vector jump instruction.

To construct a library of the two overlay loaders, assemble each loader separately into the .OBJ files OVLYM, OVLYMD. Then, use these PLIB commands to build the OVERLAY.LIB file:

```
BU  OVERLAY.LIB
FI  OVLYM,OVLYMD
NOI  $LOAD$,$OVEX$,$OVIN$
```

The symbols that would be duplicated in the library index are prevented from going to the index. (See PLIB for more information.)

A

A.2 SEGMENTATION

The overlay loader addresses three segments:

- Code
- Data
- $OVTB$

In MS-DOS programs, the two data segments are accessed by putting a segment register at the front.

OVERLAY TABLES A.3

The linkage editor creates the common block $OVTB$ that provides all necessary information about overlays to the overlay loader. $OVTB$ contains this information:

- Base Offset (2 bytes): Contains the paragraph offset of the overlay loader from the front of the program. By subtracting the base offset from the current CS register at execution time, the overlay loader determines the paragraph address at which the program was loaded.
- Overlay Flags (2 bytes): Contains information about the first overlay. Bit 15 of the word is set to zero. This bit is used by the overlay loader to keep track of which overlays are in memory. Bit 14 is the preload bit. When this bit is set, the overlay loader loads the overlay immediately upon execution of the program; a request to load it probably won't be generated by the application program. Bits 13 through 0 contain the overlay number of the overlay's ancestor. Whenever an overlay is in memory, its ancestor must be in memory as well.
- Fixup Count (2 bytes): Contains the number of base fixup entries (4 bytes each) appearing at the front of the overlay on disk. The fixup count is nonzero for MS-DOS programs. This table is rounded up to a page boundary; the overlay contents begin after the table.
- Starting Memory Address (2 bytes): Contains the memory address where the overlay should be loaded. The starting memory address is a paragraph address for MS-DOS.
- Ending Memory Address (2 bytes): Contains the address of the end of the overlay in memory. The loader must not load anything beyond this address. The ending memory address uses the same format as the starting memory address.
- Filename Offset (2 bytes): Contains the offset (from the front of $OVTB$) to the name of the file containing this overlay. These filenames consist of a name and type, separated by a period and followed by a NULL. There are no drive IDs or path names.

- Disk Address (4 bytes): Contains the address of the overlay within the selected disk file. For MS-DOS, this is a paragraph offset from the front of the file. The first portion of the overlay consists of a fixup table.
- Disk Length (2 bytes): Contains the size of the overlay on disk in paragraphs.

Except for the base offset, all the previous items are repeated for each overlay in the program. This subtable is terminated by a −1 (FFH) in the position that is normally occupied by the overlay flags field.

The next part of $OVTB$ contains the overlay filenames, each an ASCII string terminated by a NULL. These filenames are pointed to by the overlay entries described earlier. There is only one name unless PLINK was told to use separate files for overlays.

The final portion of $OVTB$ is made up of the symbol vectors. These consist of a call to the $OVLY?? routine, a word giving the number of the overlay to load, and a jump to the true address of the routine supported by the vector. The call and jump are long for MS-DOS programs. The overlay loader should never have to look at this portion of $OVTB$—any calls to this portion are automatically inserted by PLINK as the program is linked.

B

WARNING MESSAGES

Occasionally PLINK detects a situation that may cause a problem when the linked program is executed. When this occurs, PLINK issues a warning message but continues to link the program. These warning messages are listed here, along with an explanation of what has happened.

< group > is larger than 64K

You've used a group that's too big to fit in 64K of memory. Each group is a collection of segments that must reside within a 64K memory space. This lets 16-bit addresses be used to access objects within the group. If the group is too large, part of the group can't be accessed. You must reduce the size of the group. You can use overlays to decrease the memory requirement of a code group.

With MS-FORTRAN, an overflow of DGROUP can be handled by moving some items to a common block. Common blocks are limited to 64K, but you can use many of them.

This situation often occurs when you mix modules in assembly language with modules in a high level language. The class and segment names you use in the assembly language should match those used by the high level language. If they don't, your assembler segments probably will be assigned memory locations at one end of the section, this section might be too far from the other members of the group.

If you use an undefined segment in a group statement, the Microsoft assembler creates a dummy segment with no class name. For example, if you use the Microsoft assembler to link an MS-FORTRAN program containing a MASM statement (like "DGROUP GROUP DATA") whose DATA segment is undefined, you will get a dummy DATA segment with an omitted class statement. This segment won't combine with the other data segments and might cause this warning message. Check your memory map for segments with no class names.

In MS-FORTRAN, an overflow of DGROUP can be handled by moving some things to a common block. Commons are also limited to 64K, but you can have many of them. Note: Overlays may be used to decrease the memory requirement of a code group.

Offset out of range in reference to XXXX

You tried to use a near reference to the named object, but the offset was larger than 64K. The object may be in a group that is larger than 64K. The offset you used will be wrapped around to a number within range. As a result, the program probably won't address the object correctly.

No stack segment defined, 0000:0000 used

The stack location of your program is not defined in the .EXE file header (MS-DOS). This location is used to set the SS and SP registers when the program is executed. PLINK looks for a stack segment marked as such by the compiler or assembler being used.

- If found, the stack segment's eventual address is put into the header.
- If not found, zeros are used. In this case, the program probably won't function correctly unless it sets the SS and SP registers itself.

No starting address given, 0000:0000 used

PLINK cannot find a module marked with the starting address of your program. Under MS-DOS, the starting address of a program is kept in the .EXE file header. However, PLINK uses this location for a jump to the true starting address. If PLINK can't find a module marked with a starting address by the compiler or assembler, zeros are used for the starting address. The program begins execution at the first location.

Can't find module XXXX

The named module is used in the MODULE command, but PLINK can't find that module in the input files. The module name usually isn't the same as the filename. Some compilers use the same name for every module compiled. Consequently, the library manager usually replaces the module name with the filename when a module is inserted into a library. Use the DUMP utility to find a file's module name by typing:

```
DUMP  <filename>  INC  THEADR
```

This selects and displays all module name records from the file.

Frame = XXXX, target = XXXX
Fixup offset 99999 out of range in <module>

The named module contains a reference to the given target object. That reference assumed that the segment register to be used pointed to the given frame object. However, the target is more than 64K bytes from the frame and cannot be accessed. One of your groups is probably larger than 64K. Reduce its size so this access can be made correctly.

Unknown record type 99 in <module>

The named module contains a record type unfamiliar to PLINK. The whole record is skipped. This message appears the first few times the problem arises, and then is inhibited.

Checksum error in <module>

PLINK has found a checksum error. Each record in an object file contains a check field at the end for validation purposes. This message tells you that the checksum in that field is bad, but that linking continued anyway. These messages are inhibited after the first few times the problem occurs.

Check to see if the object file was patched on disk before the link. (Typically, people who patch object files don't bother changing the checksum.) If the file is badly damaged, a fatal error usually occurs soon after this message appears.

Record size error in < module >

PLINK reached the end of a record and found the number of bytes processed differs from the specified size. Each record in an object file is preceded by a byte containing the record size. (This size can be revealed by the DUMP utility.) When this warning appears, the object file is probably damaged. PLINK tries to continue processing it anyway.

Duplicate symbol XXXXX in < module >
Duplicate symbol XXXXX in DEFINE command

PLINK has found two definitions of the named global (PUBLIC) symbol. The second definition was found in another module, or you created it with the DEFINE command. There can be only one definition for each global symbol in the program being linked. PLINK stays with the first definition, ignores the duplicate, and continues linking.

C

ERROR MESSAGES

Most common error conditions display a self-explanatory screen message. The more uncommon or obscure errors are accompanied by a number you can look up in this appendix.

COMMAND SYNTAX ERRORS C.1

These errors are caused by mistakes in the input to PLINK. The input line causing the problem is displayed on the screen, with two question marks inserted at the point where the error was detected. Re-run PLINK after correcting the problem.

1 @ files are nested too deeply. Only three levels of @ files can be active at any given time. Do you have a loop in your @ file references?

2 Disk error encountered while reading @ file. Try rebuilding the file.

5 The item given for input at this point is too large. The maximum size allowed is 64 characters.

6 Invalid digit in number. The legal digits depend on the radix used. The default is hex for addresses; decimal for everything else.

10 Invalid filename. Use a legal filename.

11 Expecting a statement. You've omitted a key word that begins a statement.

14 Expecting identifier. You must enter a section, module, segment, or symbol name at this point.

15 Expecting = . An equal sign is missing.

16 Expecting a value. You must enter an expression or 16-bit quantity here.

17 No files were given to link. Use the FILE statement to specify at least one input file.

18 Expecting the) at the end of the CLASS statement. The list of segment names must be enclosed in parentheses.

C.2 WORK FILE ERRORS

When PLINK runs out of memory, it opens a disk work file named PLINK86.WRK to hold the description of the program. These error codes indicate problems with processing the work file.

30 The work file can't be created. The disk directory is probably full.

31 An I/O error occurred while writing the work file.

32 An I/O error occurred while reading the work file.

33 An I/O error occurred while positioning the work file.

34 This program contains too many program description objects. About 35,000 symbols, segments, groups, and so on can be defined. This program is too large for PLINK to handle.

C

C.3 INPUT OBJECT FILE ERRORS

These errors involve the object files that PLINK is to link. Usually, the errors occur when a file has been corrupted. Try recompiling to get a new copy of the object file. If the file causing the problem is a library supplied with the compiler, try to get a new copy.

41 Premature end of input object file. The end of the indicated file was reached unexpectedly. The file may have been truncated by copying it with a program that assumes an ALT-Z (1AH) is end-of-file.

42 Fatal read error in object input file.

43 Fatal file position error in object input file. This can occur when a library file is truncated.

OUTPUT FILE ERRORS C.4

These errors are caused by a problem in creating the output code file or memory map file (when written to disk). They are often caused by a full disk or disk directory, a disk that is write-protected, or a disk-related hardware problem.

45 Can't create output disk file. The disk directory may be full, or the disk is write-protected.

46 Invalid output file type. If given, the file type must be .EXE.

47 Fatal disk write error in output file. The disk is full or write-protected, or a hardware error has occurred.

48 Fatal disk read error in output file. An irrecoverable hardware error has probably occurred.

49 Can't close output file. The disk may be write-protected, or a hardware error has occurred.

50 Can't create the memory map disk file. Possibly the disk directory is full, or the disk is write-protected.

C.5 MISCELLANEOUS ERRORS

51 Undefined symbols exist. The listed symbols are external to one or more modules, but you never defined them. Use the DEFINE statement to create the listed symbols, or define them as internals of a module.

52 Symbol is self-defined. The given symbol was defined relative to a another symbol, and that symbol defined relative to yet another symbol, and so on until the original symbol was reached again. You've created a circular chain without defining any of the symbols.

53 Duplicate input file. Each file used in the FILE, SEARCH, or LIBRARY statements must have a unique name.

54 The program is too large. The program will not fit into a 20-bit (1 megabyte) address space. Try using overlays to decrease the memory requirement of the program.

55 Incorrect starting address in module. The starting address must be a location inside one of the segments defined in the module.

56 An absolute segment cannot be initialized at link time. These segments are supported for the purpose of defining addresses only.

57 There is a problem with the OVERLAY.LIB file. Be sure you are using the most recent version. If you created your own OVERLAY.LIB, it is incorrectly formatted.

C.6 OBJECT FILE FORMAT ERRORS

These errors are caused by a problem with the format of the input object files. These files should be in the correct format for Intel 8086/8088 compiler output.

Sometimes an error occurs because the object file is using a feature that PLINK does not support. You are probably trying to use a compiler that is not compatible with PLINK. See your dealer. It's also pos-

sible that the relocatable file has been corrupted in a subtle way. Try recompiling.

61 An LTL segment appeared in an Intel module. PLINK does not accept these as input.

65 An absolute starting address was specified in the given file. The starting address must be given relative to some segment.

66 Unsupported group element type in module. Only segments can be included in groups. (The group component descriptor code is FFH.)

69 Invalid record type in Intel module.

70 Invalid location specified for fixup. The LOC field is greater than 4 for a segment-relative fixup.

71 Invalid location specified for fixup. The LOC field is greater than 1 for a self-relative fixup.

73 Bad Frame specification. A frame of type 6 or 7 was given. These are not supported by PLINK.

PROGRAM STRUCTURE ERRORS C.7

These errors involve a problem with the overlay structure that was specified. Correct the problem and relink.

80 Overlays are nested too deeply. There are too many levels in the overlay structure.

81 There are too many END statements. The number exceeds the number of BEGIN statements that need to be closed.

82 The BEGIN and END statements are unbalanced. There should be the same number of each.

C.8 PLINK BUGS

These errors indicate a bug in PLINK. It's likely there's nothing you can do to correct them. If one of these errors occurs, try running PLINK again. If the error persists, contact your dealer.

200 Missing segment (SegFnd).

201 Expandable array bug.

202 Public segment base missing (FIXUPP).

203 Unrecognizable Object (Eaddr).

204 Unrecognizable Object (Daddr).

205 SEEK errors while writing output file. An attempt to SEEK past end-of-file.

206 Can't find symbol during pass 2 (ExtDef).

207 Unrecognizable Object (Addr32).

208 No I/O buffers available.

209 Segment not assigned to any section.

219 Bad object block (GetBlock).

220 Requested record size too large (Newrec).

221 Invalid object key (Q).

222 Invalid object key (QM).

225 No object (Daddr).

226 No ouput file (Daddr).

D

SUPPORTED COMPILERS

This appendix lists the assemblers that PLINK has been checked out with. If you are using a more recent version of a compiler listed here, the chances are good that PLINK can handle it. Your dealer may know for sure.

This manual contains examples using some of these compilers (these compilers are marked with an asterisk). Some testing was done with the others, but no test programs of significant size were linked. With these semi-tested compilers, PLINK links the program correctly, but you must make sure that the segment classes are ordered correctly (especially if you used overlays).

If you want to use a compiler that's not on this list, you are on your own. Compilers from the same source usually work in the same way. A useful guideline is that PLINK can usually link anything that MS-LINK can handle.

MS-DOS compilers include:

- Lattice C
- MS-FORTRAN *
- MS-Pascal *
- MS-BASIC
- MS-COBOL
- MACRO-86 Assembler
- PL/M-86
- ASM86

E

DEBUGGING HINTS

This appendix offers some debugging hints, and explains how to use Plink86 memory map reports.

ADDRESSING E.1

Addresses in the memory map are generally given as 20 bit offsets from the start of the program. Symbols, however, are given a two-part Intel format address of the form "paragraph:offset". When the program is executed it will typically be loaded at a non-zero address. Therefore, when you use the debugger, you must add the program load address to addresses obtained from the map.

Suppose, for example, that the program was loaded at memory address 8100. A segment (or other 20 bit address) appearing in the map as 525 would correspond to a memory address of 8100 + 525, or 8625. A symbol appearing as 630:255 would be in memory at (810 + 630):255 = E40:255, or E655 in long form.

Locations accessed via short (16 bit) addresses are typically addressed relative to a particular group or segment. If there is currently a segment register pointing to that group or segment, you can use the offset as it is given in the map. For example, a symbol in DGROUP addressed as 630:255 can be referred in the debugger as DS:255 (assuming DS is assigned to that location).

Most MS-DOS programs move the DS register to its assigned location some time after execution begins.

The address at which MS-DOS loads the program into is not always easy to determine. You can look at the contents of the CS register for simple un-overlaid programs using short addresses. These programs usually are set up with the code area in front; the operating system sets

CS to this location. If a program uses far calls, however, CS may not be set to the front of the program, because MS-DOS sets CS:IP to the starting address of the program at load time.

The following method works for all un-overlaid MS-DOS programs: Look up a symbol named $STRT$ in the "G" or "M" memory map report. This is a dummy symbol created by Plink86 which gives the address where program execution is to begin. Then subtract the paragraph address given for this symbol from the contents of the CS register before the program begins execution. This yields the paragraph address where the program was loaded.

For example, if CS is 1A60 and $STRT$ has an address 1250:267, the program was loaded at paragraph 1A60 − 1250 = 810, or 8100 in long form.

If a program does have overlays, use symbol $OVIN$ instead of $STRT$. This symbol is the initialization entry point for the overlay loader, and is always executed first.

E.2 DEBUGGING WITH OVERLAYS

If a program has overlays, the overlay loader initialization routine is always executed before the user program. You can go directly to the user program by setting a break-point at symbol $STRT$ (discussed in Section E.1).

A break-point can't be set in an overlay that is not currently in memory. A break-point can, however, always be set at symbol $OVEX$. This symbol is in the overlay loader at the point where the requested overlay has been loaded and the loader is about to return to the user program.

Another method is to find the call to the overlaid routine. The address called will be the overlay vector for the symbol (unless the NEVER statement was used). Set a break-point at the jump following the call to the overlay loader (see Appendix A).

INDEX

Pfix86™

STILL FIXING BUGS THE HARD WAY?

Advanced Dynamic and Symbolic Program Debuggers

Both Pfix86 and Pfix86 Plus™ give you the power of a multiple-window debugger with advanced breakpoint capability and an in-line assembler, so you can make program corrections directly in assembly language. Pfix86 Plus gives you the added advantage of symbolic debugging to handle overlayed programs developed with Phoenix's Plink86™ overlay linkage editor.

In addition, both debuggers provide standard debugging tools including memory and register examination/modification and program execution with breakpoints. However, they extend these common tools considerably with user-controlled data formatting, powerful expression evaluation for user-entered data, and extremely flexible trap capabilities with tracing.

Phoenix

Pfix86™

Symbolic Debugging – Symbolic debugging eliminates the struggle microcomputer programmers traditionally have with linker memory maps. When a program linked by Phoenix's Plink86 is being debugged, Pfix86 Plus displays the names of public symbols as their addresses are encountered. Pfix86 Plus will accept symbol names wherever you normally would need to type in an address. This greatly simplifies tasks such as setting a breakpoint at the start of a procedure. It even works on overlayed programs.

Multiple Window Display – View program code and data, breakpoint settings, current machine register and stack contents all at the same time. Make changes by simply positioning the cursor within a window and entering a new value. Enter instructions in the code window and bytes, words, addresses, long integers, or text strings into the data window. Or review text files in the file window. You can even adjust window size to match your needs. Pfix86 will preserve the program screen if you are using one terminal. Or, you can have your program screen on a monochromatic display and your debugger screen on a color display. Or, vice versa.

Advanced Breakpoint Capabilities – Want to run a program at full speed until a loop has been performed 100 times, or have the program automatically jump to a temporary patch area? You can with the powerful breakpoint features Pfix86/Pfix86 Plus offer. Set up breakpoints that are marked in both the breakpoint and code windows. Then disable them so that they remain in the table in an inactive state. Temporary breakpoints can be set right in the code window with a single keystroke. When reached in execution, the breakpoints are eliminated.

Trace Facility – With a single keystroke, you can trace an instruction and the action will be immediately reflected in code, data, stack, and register windows.

Easy-To-Use Menus – An easily accessible menu makes the power of Pfix86 instantly available to the new user, and the menu won't inhibit the practiced user: often-used functions are assigned to the IBM PC function keys, and menu selections can be made with just a few keystrokes.

I/O – With both Pfix86 and Pfix86 Plus you can read and write absolute disk sectors and access your computer's I/O ports.

System Requirements – Both debuggers will run on the IBM PC (or compatibles) because of their extensive use of the keyboard and display. A custom version is available for the TI Professional, Wang Professional, DEC Rainbow, and other machines. No other special hardware is needed. At least 50K of free memory is required, and the MS-DOS™ (or PC DOS) operating system (version 2.0 or above) must be used.

Pfix86/Pfix86 Plus. One in a series of software development tools by Phoenix. It's the right tool for the job.

Phoenix

Phoenix Computer Products Corporation
1416 Providence Highway, Suite 220
Norwood, MA 02062
(800) 344-7200
In Massachusetts (617) 769-7020

PLIB

VICTOR Publications
380 El Pueblo Road
Scotts Valley, California 95066
(408) 438-6680

First VICTOR printing November, 1983.

ISBN 0-88182-039-3 Printed in U.S.A.

CONTENTS

Appendix A: Error Messages

CHAPTERS

1

INTRODUCTION

PLIB is a program that manipulates libraries of object files. It supplements the PLINK linkage editor, and handles object files and libraries that conform to the format generated by Microsoft compilers for the Intel 8086/8088. (This is the standard Intel format with an enhanced library index.)

This manual assumes that you have some programming experience. It is designed to be read from front to back—each chapter assumes you understand the information in all previous chapters.

Chapter 2 explains the "object library" concept and the capabilities of PLIB. Start here if you are unfamiliar with library managers. This chapter also discusses object files and linkage editors.

Chapter 3 shows how you can use PLIB to handle several common object library situations. It also gives an informal explanation of what the PLIB commands do. You may want to go directly to this section if you are experienced with linkage editors and library managers; this section gives enough information to use PLIB.

Chapter 4 is a detailed list of PLIB commands and features. Use this section when you need more information than is provided by the examples in Chapter 3.

An appendix describes the error messages that can appear when you use PLIB.

2

LIBRARY MANAGER CONCEPTS

This chapter introduces the basic concepts of libraries and modular programming, and gives an idea of how PLIB can be used to create and manipulate libraries of object modules. It also contains a short glossary of terms used in this manual.

WHAT DOES A LIBRARY MANAGER DO? 2.1

Often it's convenient to divide a large programming job into smaller pieces called "modules" that can be edited and compiled separately. Sometimes you have to use modules, because microcomputer compilers can handle only a limited number of program lines at once. In either case, modular programming is a way to organize your program into manageable pieces that are easier to understand and work with. After you create and compile your program modules, you must link them together with a linkage editor to produce an executable program.

You may find that some modules in your program are useful in other programs. With a bit of work, you can make these modules more general in function and use them in several programs. In this way, you gradually build up a "library" of useful routines that can be linked into programs by the linkage editor whenever needed.

Almost all compiler packages are sold with a library because there are many useful functions (such as arithmetic on real numbers) that are not supported by hardware, and that have to be implemented as procedure calls. Compiler libraries also contain modules that support high-level features of a language, such as formatted output with FORTRAN. This kind of library is usually called the "run-time support," since its modules are required while executing the program.

Because of the importance of libraries, linkage editors typically have special library-handling facilities. To save memory space, only the library modules actually required by a program are linked in. Sometimes a library is a group of object modules that the linkage editor searches sequentially to find the needed modules. Some more sophisticated systems use a "library index" that contains a list of the public symbols offered by each library module, and the location of the module that defines each symbol. A library index helps the linkage editor to find the required modules quickly. Microsoft compilers have libraries that use the indexed structure.

The PLIB library manager creates and manipulates object module libraries, and is a useful aid to the linkage editor, PLINK. PLIB has commands that create libraries from individual object modules, and that extract a selected module from a library. PLIB can also merge libraries and replicate the library search process used by the linkage editor while creating a program. (In other words, you can create a library that consists only of the modules that the linkage editor would include in a particular program.)

PLIB also has a powerful cross-reference function. With this function, you can generate a listing that shows each public symbol, the module that defines the public symbol, and a list of other modules that refer to it. You can use this listing to cross-reference one or more libraries. When used with the library search feature already described, the cross-reference function can generate a cross reference of a program to be created by the linkage editor.

2.2 TERMINOLOGY

This section defines some terms commonly used when discussing library managers.

OBJECT FILE 2.2.1

The compiler (or assembler) produces the object file as output after compiling a program. The object file usually contains one module. The PLINK linkage editor takes the object file as input.

RELOCATABLE FILE 2.2.2

This is another term for object file. This term is often shortened to "REL FILE."

MODULE 2.2.3

This is the smallest unit of code that can be compiled at one time. The relocatable file created by a compiler typically contains one module. PLIB combines modules into libraries for easy access by PLINK.

You usually create a module for each major function within a program; however, a compiler usually lets you put several procedures or functions into a single module. Although the functions and procedures in a module can be called separately, the linkage editor treats the module as a single entity and the procedures can no longer be separated.

2.2.4 LIBRARY

A library is a relocatable file that contains more than one module. A library often has an index to its modules; this helps the linkage editor quickly find the modules it needs. This kind of library is called an "indexed library."

2.2.5 LIBRARY SEARCH

This term refers to the way that libraries are processed by the PLINK linkage editor. When PLINK encounters an undefined external symbol, it looks up the symbol in the library index. If a module in the library defines the symbol (i.e., specifies the symbol as public), that module is included in the program. If the library does not have an index, the linkage editor usually scans the entire library one or more times to find the needed module. A module loaded from the library can contain undefined external symbols of its own; the linkage editor keeps searching to find those as well.

See PLINK in the *Applications Programmer's Tool Kit II, Volume II*, for more information.

3

USING PLIB

This chapter tells how to use the PLIB functions introduced in Chapter 2. These include:

- Creating and merging libraries
- Searching libraries
- Updating a library
- Extracting modules from a library
- Making a cross-reference listing

Before you can use any of these functions, however, you must load PLIB into your operating system. First, boot your operating system. After the system prompt appears, type:

```
PLIB86(cr)
```

The program loads and you can go on to work with library modules.

CREATING AND MERGING LIBRARIES 3.1

To create a new library, use the BUILD and FILE commands. For example, loading PLIB and typing:

```
BUILD DB.LIB(cr)
FILE BTREE, SORT, REPGEN, FIRSTLIB.LIB;(cr)
```

after the prompt creates a library called DB.LIB that contains the files listed after the FILE command. These files can be individual object modules or complete libraries. All of the files are merged into a single library. Normally you can type commands on as many lines as you want. End the last line with a semicolon to begin processing.

Each PLIB statement begins with a key word like BUILD or FILE and is followed by arguments, possibly separated by commas. Key words can be abbreviated by leaving off characters at the end. For example, you can type BU and FI instead of BUILD and FILE. PLIB displays an error message if your abbreviation could be confused with another command. Command input is free format; blank lines are ignored.

You can also give PLIB commands while the program is executing. The library defined in the last example can also be created by entering on a single line:


```
PLIB86 BU DB FI BTREE, SORT, REPGEN, FIRSTLIB(cr)
```

Since you did not specify an extension for your library file, PLIB automatically assigns the default extension .LIB.

3.2 LIBRARY SEARCH

Suppose you want to create a library that consists of several modules, plus the portions of another library that are referenced by those modules. To do this, you must use the LIBRARY command to search the second library for the needed modules. Using the library defined in the last example, enter:

```
BU DB FI BTREE, SORT, REPGEN LIB FIRSTLIB.LIB
```

Only the portions of FIRSTLIB.LIB referenced by the three other files in the command are put into the DB library.

UPDATING A LIBRARY 3.3

To update a library, you must copy the old library to an output file, delete the module to be updated, and include the new module. To replace the module COSINE in library MATHLIB, for example, rename the current library file MATHLIB.LIB to MATHLIB.OLD. Then, enter:

```
BU MATHLIB FI COSINE, MATHLIB.OLD EXCLUDE COSINE
```

The EXCLUDE statement applies to the old filename; it causes the COSINE module in MATHLIB to be ignored.

EXTRACTING A MODULE 3.4

The EXTRACT statement causes a single object module file to be created. Unless you specify a particular module, EXTRACT uses the first object module found in the input files. For example, typing:

```
EXT OLDCOS FI MATHLIB.LIB INCLUDE COSINE
```

creates the file OLDCOS.OBJ that contains the object module COSINE. The INCLUDE statement is the counterpart of EXCLUDE: it applies to the previous input file and causes only those modules named to be considered for processing.

EXTRACT cannot be used at the same time as BUILD.

3.5 CROSS-REFERENCE LISTING

Use the LIST command to create a cross-reference listing. Using the same input file commands given in previous examples, you can enter:

```
LIST = DB S FI BTREE, SORT, REPGEN, FIRSTLIB.LIB
```

This creates a cross-reference listing file named DB.LST. This file describes the modules in all of the files listed in the command. The S selects the type of cross-reference report. (For a list of other report types available, see the LIST command description.) The equal sign (=) specifies that the report is to be put into a disk file. If the equal sign is not used, the report appears on the screen.

4

PLIB COMMANDS

This chapter describes the basic elements you use to create a PLIB statement. It also lists the PLIB commands and gives examples of their use.

INPUT ELEMENTS 4.1

PLIB commands and statements contain numbers and identifiers. Numbers are normally decimal.

An identifier is the name of an object such as a module or symbol. An identifier is a sequence of no more than 30 characters. It cannot contain any spaces or any of the following characters:

^ = ; < > \ , / ! ' # & * + - : @ DEL

All lowercase letters are automatically converted to uppercase. The first character of an identifier cannot be a digit 0–9.

These restrictions on valid identifier characters are avoided if you use the escape character (^). The character immediately following the escape character is treated as a normal identifier character. Enter the escape character twice to include the escape character itself in an identifier.

Here are some examples of valid identifiers:

ProgramI
SORT3
ABC^@

Here are some invalid identifiers:

34ABC	begins with a number.
NIM A	contains a space.
PROG%1	starts a comment with %.

The previous invalid identifiers can be made valid using the escape character (^):

^34ABC
NIM^ A
PROG^%1

Identifiers that appear in object files are truncated to 50 characters to compare them with other identifiers in the program. Identifiers may be truncated again for inclusion in report listings.

4.2 DISK FILENAMES

PLIB uses the filename format of your operating system. The first illegal filename character terminates the name. Use the escape character to include an illegal character in a filename.

The PLIB filename can include a device specification, the filename itself, and an extension, as shown here:

< device > : < filename > . < ext >

If you do not specify a device, PLIB uses the logged drive.

INPUT MODES 4.3

PLIB can be used interactively, or input can be given while the program executes. Enter input using this format:

PLIB86 < statements > (cr)

To use PLIB in the interactive mode, enter:

PLIB86(cr)

PLIB reads all statements you type at the keyboard, prompting you with an asterisk (*). All input is stored until you press the carriage return. All standard line editing features supplied by your operating system are available.

You can use a disk file containing all or part of a command at any point in your input by preceding the disk filename with @. If you do not specify an extension, PLIB automatically uses the default extension .LNK. These disk files can contain further @ specifications, up to three levels deep. The most common use of this feature is to prepare a file containing a complete command. Typing:

PLIB86 @NEWFILE(cr)

creates the library NEWFILE.LIB.

These .LNK files can be prepared once for a given library and then used over and over, greatly simplifying the process of creating libraries.

PLIB reads an entire command, checking only for syntax, before any file processing occurs.

4.4 COMMANDS

This section gives the format used for PLIB commands and describes each command in detail.

4.4.1 COMMAND FORMAT

All PLIB input is free format. Blank lines are ignored, and a command can extend over any number of lines. You can include comments by prefacing them with a percent sign (%). When PLIB encounters a percent sign, it ignores all subsequent characters on the same line.

Input takes the form of a list of statements. Each statement begins with a key word and many are followed by arguments separated by commas. In this statement, for example:

```
FILE A,B,C
```

FILE is the key word, and A, B, and C are the arguments. Key words can be abbreviated by omitting trailing characters, as long as the resulting abbreviation is unique. For example, the previous statement can also be entered as:

```
FI A,B,C
```

If PLIB finds a syntax error, the current input line appears on the screen with two question marks (??) appearing at the point where the error was detected. This is followed by an error message. (See the appendix.) You must re-enter the command before proceeding. If some other error occurs, PLIB terminates with an error message.

You must tell PLIB which object files and libraries to use for input and which modules to take from them. Usually you use the FILE command to process all modules in the given files. For example:

```
FILE COSINE, SIN, ARCTAN
```

The LIBRARY and SEARCH commands have a similar function but are used only on libraries. These commands select only those modules that define a public symbol needed by an already processed module. This selection process is called a "library search," and is a process carried out by most linkage editors. Library searches ensure that only those library modules actually needed are included in the program.

Here are statements that use the LIBRARY and SEARCH commands:

```
LIBRARY MATHLIB
SEARCH FORTRAN
```

The LIBRARY command causes the specified libraries to be searched once. The SEARCH command causes the libraries to be searched until there are no remaining undefined symbols. SEARCH is needed only if you need to search two or more libraries that contain references to symbols defined in each other.

When you request an object file, PLIB searches drive A or the logged drive (if you are logged onto a drive other than drive A). If the requested object file is not found, PLIB asks you to enter a drive name. Diskettes can be changed at this time if necessary. (Be sure that the removed diskette does not contain open files such as the BUILD and EXTRACT files.)

If PLIB runs out of memory, a work file is opened on the default disk. Do not remove that disk while PLIB is running.

PLIB accepts a PATH name as part of an object filename. If an object file can't be found, PLIB looks for a string named OBJ in the operating system environment and appends its value to the front of the filename, after stripping the drive ID.

Suppose you enter:

```
SET OBJ = \OBJECT
```

and then run PLIB. Also, suppose one of the commands to PLIB is:

```
FILE B:TEST.OBJ
```

and that TEST.OBJ does not exist on drive B. PLIB then strips the B: designation from the filename and tries \OBJECT\TEST.OBJ to obtain the requested file.

If an object file (but not a library) is being processed, the module it contains is usually given the same name as the filename when the module is copied to the output file. This is done because some compilers do not supply a unique module name. Use the AS statement if you do not want this automatic renaming to occur. When you use AS, the module is given the filename specified in the most recent FILE statement. If you type:

```
FILE MATH1 AS COSINE
```

the module in MATH1 is named COSINE instead of MATH1.

INCLUDE and EXCLUDE Statements

You can further restrict the modules selected from a library with FILE, LIBRARY, and SEARCH by using the INCLUDE and EXCLUDE statements. These statements are followed by a list of module names, as shown here:

```
FILE MATHLIB INCLUDE SIN, COSINE LIB MATHLIB, DB EXCLUDE BTREE
```

With the INCLUDE statement, only the specified modules are considered for processing. With EXCLUDE, all modules except those specified are considered for processing.

INCLUDE and EXCLUDE apply to the immediately preceding FILE, LIBRARY, or SEARCH file. In the second example, the EXCLUDE BTREE applies only to the DB library, not to MATHLIB.

BUILD Command

The BUILD command creates a library from the modules selected from the input files. Follow BUILD by the name of the library file to be created. If you do not specify an extension, PLIB uses the default extension .LIB.

Here are example statements using BUILD:

```
BUILD DB.LIB
BUILD D:MATHLIB
```

After all modules are written out, the library index is created.

Be sure that the output file does not have the same name as any of the input files. For instance, entering:

```
BUILD MATHLIB FI COSINE, ARCTAN, MATHLIB
```

doesn't work because MATHLIB is erased before it is read.

The BUILD command cannot be used at the same time as the EXTRACT command. If no output is requested (i.e., there is no BUILD, EXTRACT, or LIST command), then PLIB simply reads the input modules and reports any errors it finds.

EXTRACT Command

The EXTRACT command extracts a single object module from a library file and puts the module into a separate disk file. The EXTRACT command is followed by the name of the file to be created, as in these examples:

```
EXTRACT COSINE.OBJ
EXTRACT ARCTAN
```

If you do not specify an extension, PLIB assigns the default .OBJ.

Unless you specify otherwise, the EXTRACT command extracts the first module in the given input files. Because of this, it's a good idea to use the INCLUDE statement to specify the particular library module you want. For example:


```
EXTRACT COSINE FI MATHLIB
```

extracts the very first module in MATHLIB, even if that module is not named COSINE. To get the correct module, enter:

```
EXTRACT COSINE FI MATHLIB INC COSINE
```

LIST Command

The LIST command generates report listings about the object files being processed. You can specify two types of listings.

- When followed by M, LIST produces an alphabetical listing of all modules.
- When followed by an S, LIST produces an alphabetical listing of all public and external symbols. Each symbol is followed by parentheses containing the name of the module in which the symbol is defined. (For external symbols, this part of the listing is blank.) Then comes a cross-reference listing—an alphabetical list of all modules that access the symbol.

If you enter an equal sign and a filename after the LIST command, the listing you specify is sent to the named disk file or device.

These statements use the LIST command:

```
LIST M
LIST = DB.LST M, S
LIST = XREF.LST S
```

WIDTH and HEIGHT Commands

WIDTH and HEIGHT are used to reconfigure the listing generator for different paper sizes. The default values are 80 columns and 66 lines per page.

Use the WIDTH command to change the number of columns. This command, for example, produces a page 132 columns wide:

```
WIDTH 132
```

Use the HEIGHT command to change the number of lines. This command produces a page 88 lines long:

```
HEIGHT 88
```

NOINDEX Command

Normally, all public symbols from all modules are inserted into the library index. PLINK continues to create a library if it finds a duplicate public symbol; however, a warning message is displayed, and the index entry for that symbol selects the first module that defines the symbol.

Use the NOINDEX command to exclude certain symbols from the library index. For example:

```
NOINDEX SYM1, SYM2, SYM3
```

excludes SYM1, SYM2, and SYM3 from the index.

Suppose you want to create a library that contains several versions of the same module, such as a driver for a hardware device. The entry point symbols are the same for each version of the module, but the code in the module differs according to the device. If you put all the modules into the library, you will get duplicate symbol warnings, and at link time the linkage editor won't select the desired module.

You can solve this problem by using NOINDEX on the module entry points. This excludes these symbols from the library index. To get the linkage editor to select the correct modules, insert a unique, unused dummy symbol into each module. At linkage edit time, one of these symbols will be accessed to create a need for the desired module. The linkage editor will then select it when the library is searched. For example, you could use the statement "DEFINE F00 = DRIVER1" to select the module containing driver 1.

Alternatively, you can use the existing dummy index entry to select the module. The name of each module is in the library index, followed by an exclamation point. For example, if the library contains a module named DRIVER1, the dummy index entry will be DRIVER1!. You can use this entry instead of creating your own dummy module entry point.

A

ERROR MESSAGES

Most error conditions detected by PLIB cause an error message to appear on the screen. Usually the message is fairly easy to interpret. For more uncommon or obscure errors, a number appears next to the message. These error codes can be checked in the following listing.

COMMON SYNTAX ERRORS A.1

These errors are caused if you make a mistake in your input to PLIB. The input line causing the problem is displayed, with a pair of question marks appearing at the point where the error was detected. Rerun PLIB after correcting the problem.

1 @ files are nested too deeply. Only three levels of @ files can be active at any given time. Check to see if you have a loop in your @ references.

2 Disk error encountered while reading @ file. Try rebuilding the file.

5 The item given for input is too large. The maximum allowable size is 64 characters.

6 Invalid digit in number (i.e., not 0 through 9).

10 Invalid filename. The filename is not legal for your operating system.

11 Expecting a statement. A key word is missing.

12 The INCLUDE and EXCLUDE statements cannot be used simultaneously on the same input file.

14 Expecting identifier. A section, module, segment, or symbol name must be entered at this point.

15 Expecting an equal sign.

16 Expecting a value. You need to supply an expression or 16-bit quantity.

17 No input file has been specified. You must use the file statement and specify at least one input file.

18 The BUILD and EXTRACT commands cannot be used at the same time. Run PLIB separately for each command.

A.2 WORK FILE ERRORS

When PLIB runs out of memory, it opens a work file on disk (called PLIB.WRK) to hold the description of the library. These error codes indicate a problem with processing the work file.

30 The work file can't be created. There is probably no space in the disk directory.

31 An I/O error occurred while writing the work file.

32 An I/O error occurred while reading the work file.

33 An I/O error occurred while positioning the work file.

34 There are too many module-description objects in the library. (About 50 thousand symbols, segments, groups, and so on can be defined.) The library is too large for PLIB to handle.

INPUT OBJECT FILE ERRORS A.3

These errors have to do with the object files given to PLIB for processing. They usually occur when a file has been corrupted in some way. Try recompiling to get a new copy of the object file. If the corrupted file is a library supplied with a compiler, try to get a fresh copy.

41 Premature end of input object file. The end of the indicated file was reached unexpectedly. The file may have been truncated by copying it with a program that assumes ALT-Z (1AH) is end-of-file.

42 Fatal read error in object input file.

43 Fatal file-position error in object input file. This can occur when a library file is truncated.

OUTPUT FILE ERRORS A.4

These errors are caused by problems in creating the output code file or memory map file when writing to disk. They are often caused by a full disk or disk directory, a disk that is write-protected, or a hardware problem with the disk.

45 Can't create output disk file. The disk directory may be full or the disk may be write-protected.

46 Output file too large. The given modules won't fit into the library. Break the library into two or more smaller libraries.

47 Fatal disk write error in output file. The disk is probably full or write-protected, or a hardware error may have occurred.

48 Fatal disk read error in output file. An irrecoverable hardware error has probably occurred.

49 Can't close output file. The disk is probably full or write-protected, or a hardware error has occurred.

50 Can't create the LIST output file. The disk directory may be full, or the disk is write-protected.

A.5 MISCELLANEOUS ERRORS

51 There are too many symbols to be placed into the library index. Break the library into several smaller ones.

52 No modules were selected (by the library search, or the INCLUDE or EXCLUDE statements) to be put into the output file (BUILD or EXTRACT).

54 Your computer does not have enough memory to run PLIB. This error should never occur.

A.6 BUGS

These errors indicate a bug in PLIB. It's unlikely you can do anything to correct them. If one of these errors occurs, run PLIB again. If the error persists, gather the relevant information and contact your dealer.

201 Expandable array bug.

205 SEEK errors while writing output file. (Attempt to SEEK past end-of-file.)

219 Bad object block (GetBlock).

221 Invalid object key (Q).

222 Invalid object key (QM).

INDEX

VICTOR®

PMATE
Quick Reference Guide

PMATE Quick Reference Guide

INSTANT COMMANDS

CURSOR MOVEMENT

ALT-B or ↓	Down one line
ALT-Y or ↑	Up one line
ALT-G or ←	Left one space
→	Right one space
ALT-U	Up six lines
ALT-J	Down six lines
ALT-P	Beginning of next word
ALT-O	Beginning of current or preceding word
ALT-FM	Beginning of line
ALT-A	Beginning/end of text

SCROLLING

ALT-FG	Scrolls display left one column
ALT-FH	Scrolls display right one column
ALT-FY	Scrolls display up one line
ALT-FB	Scrolls display down one line

DELETING AND RECOVERING TEXT

Backspace or ALT-H	Deletes character preceding cursor
ALT-D	Deletes character at cursor; moves text to right of cursor left one column
ALT-K	Deletes line beginning at cursor
ALT-W	Deletes text up to next word
ALT-Q	Deletes word preceding cursor
DEL	Deletes character at cursor; moves cursor to next character
ALT-R	Recovers last text deleted; puts it at cursor position

LOCKS OF TEXT

Tags beginning of block to be moved

Moves text between tag and cursor into special buffer

Inserts contents of special buffer at cursor

Toggles cursor between tag and current cursor position

Sets indentation at cursor if cursor position is tab stop

Indents text after indent is set

Increments indented text and cursor four columns

Decrements indented text and cursor four columns

MODES

Enters Command Mode

Enters Overtype Mode

Enters Insert Mode

NEOUS INSTANT COMMANDS

Inserts a line below cursor

Inserts a line; moves cursor to beginning of new line

Changes case of character at cursor; advances cursor one column

Toggles default case of text between upper- and lowercase

Clears command line or stops command execution

Inserts command line in text area

Redraws and reformats screen

Enters character x in text 4 times (ALT-SS x enters x in text 16 times and ALT-SSS x enters x in text 64 times)

Executes < command > n times

and >

COMMAND-LINE COMMANDS

CURSOR MOVEMENT

A	To beginning of text in memory
UA	To beginning of text file
Z	To end of text in memory
UZ	To end of text file
nM/ − nM	Forward/backward n columns
nL/ − nL	Up/down n lines
nP/ − nP	Up/down n paragraphs
nW/ − nW	Forward/backward n words

INSERTING TEXT

I < strng >	Inserts < strng > at cursor
nI	Inserts ASCII character n at cursor
IA@n	Inserts contents of buffer n at cursor
IAa$	Gets string a from command line; inserts string a in text at cursor
@n\I	Inserts value of variable n in text at cursor

REPLACING TEXT

R < strng >	Replaces text at cursor with < strng >
nR	Replaces text at cursor with ASCII character n
RA@n	Replaces text at cursor with contents of buffer n

SEARCHING FOR TEXT

(−)S < strng >	Searches text in memory forward (backward) for < strng >
(−)US < strng >	Searches entire text file forward (backward) for < strng >
(−)SA@n	Searches text in memory forward (backward) for match with contents of buffer n
(−)UAS@n	Searches entire text file forward (backward) for match with contents of buffer n

TEXT

$ <st2>	Changes first (previous) occurrence in memory of <st1> to <st2>
>$ <st2>	Changes first (previous) occurrence in text file of <st1> to <st2>
A@x	Changes first (previous) occurrence in memory of match with buffer n contents to match with buffer x contents
$A@x	Changes first (previous) occurrence in text file of match with buffer n contents to match with buffer x contents
>$ <s2>$]	Globally changes all occurrences of <s1> to <s2>

TEXT

	Prints entire text file
	Prints n lines of text beginning at cursor
Γ]	Prints n pages of text x lines each (beginning at cursor)

TEXT

	Deletes n characters forward/backward
	Deletes n lines forward/backward

G MODES

	Enters Command Mode
	Enters Overtype Mode
	Enters Insert Mode if n isn't 0 or 2
	Turns Format Mode on; specifies column n as right margin
	Turns Format Mode on/off

) SEPARATION AND EXECUTION

	Separates commands on command line
	Executes commands on command line

BUFFER COMMANDS

(−)nBC	Copies n lines from edit buffer into special buffer; deletes previous contents of special buffer
(−)nBxC	Copies n lines from edit buffer into buffer x; deletes previous contents of buffer x
(−)nBxD	Copies n lines from edit buffer into buffer x before cursor
(−)nBM	Moves n lines from edit buffer into special buffer; deletes previous contents of special buffer
(−)nBxM	Moves n lines from edit buffer into buffer x; deletes previous contents of buffer x
(−)nBxN	Moves n lines from edit buffer into buffer x just before cursor
BG	Inserts contents of special buffer at cursor
BxG	Inserts contents of buffer x at cursor
BxE	Displays buffer x for editing
BxK	Deletes contents of buffer x while you are editing another buffer
T	Tags beginning of text to be moved
.x	Executes command string or macro stored in buffer x

ATIONS

	Declares (new) output filename < txtfl > or (existing) input filename < txtfl >
< f2 >	Declares input filename < f1 > and output filename < f2 >
	Writes text to output file on disk; reopens that file as input file
>	Writes text to new output file < newfl > ; opens < newfl > as input file (original input file is not changed)
	Writes text to output file; clears buffer T
>	Writes text to renamed output file < newfl > ; clears buffer T
	Closes input and output files as they are (on disk); deletes contents of edit buffers
	Clears edit buffer without writing text to disk
	Exits to operating system
>	Saves copy of PMATE as < newfl >
	Reads copy of < txtfl > from disk into edit buffer just before cursor; < txtfl > on disk is unchanged and currently declared input and output files remain the same
>	Reads n lines of < txtfl > from disk into edit buffer just before cursor; < txtfl > on disk is unchanged and currently declared input and output filenames remain the same
>	Writes copy of edit buffer contents to new file < newfl > ; currently declared input and output filenames remain the same
fl >	Writes copy of n lines of edit buffer contents to new file < newfl > ; currently declared input and output filenames remain the same
	In Manual Mode, reads n pages of text from input file appending pages to text in edit buffer
	In Manual Mode, reads n pages of text from output file prepending pages to text in edit buffer
	In Manual Mode, writes n pages of text from beginning of edit buffer to output file
	In Manual Mode, writes n pages of text from end of edit buffer back to input file
	In Manual Mode, writes n pages of text to output file; reads n pages of text from input file to edit buffer

anual Mode, commands nXI and nXO read and write pages
nes.

DIRECTORY MAINTENANCE

XL	Lists all files in current working directory of default drive
XL < d > :*.*	Lists all files in current working directory of drive < d >
XL < pthnm > *.*	Lists all files in subdirectory < pthnm > of default drive
XL < d > : < pthnm > *.*	Lists all files in subdirectory < pthnm > of drive < d >
XX < txtfl >	Deletes file < txtfl > from current working directory of default drive
XX < pthnm > < txtfl >	Deletes file < txtfl > from subdirectory < pthnm > of default drive
XX < d > : < pthnm > < txtfl >	Deletes file < txtfl > from subdirectory < pthnm > of drive < d >
XS < d > :	Changes to the current working directory of drive < d > from any other drive; drive < d > becomes default drive
XP < pthnm >	Changes to subdirectory < pthnm > on default drive; < pthnm > becomes current working directory

SETTING TAB STOPS

YK	Deletes all tab stops; each space becomes a tab stop
nYS	Sets a tab stop at column n
nYD	Deletes the tab stop at column n
nYE	Deletes all old tab stops; sets new ones every nth column
nYF	Replaces all tabs in the next n lines with spaces
nYR	Replaces blocks of spaces in the next n lines with tabs (where possible)

INDENTING TEXT

nYI	Sets indentation to tab stop at column n (use Tab key to indent text)

SETTING VARIABLES

nVx	Sets the value of variable x to n
nVAx	Adds n to the value of variable x

EOUS MACRO COMMANDS

Suppresses error message

Displays < strng > to prompt for keyboard input and gets character from keyboard if n is nonzero or missing; displays < strng > on command line without processing keyboard input if n is 0

Turns Trace Mode on/off

Executes macro x stored in permanent macro area

Suppresses interpretation of characters until next carriage return

AND BRANCHING

Enclose iteration loop

Enclose iteration loop except when using next (^) and break (_) commands

Skips to next set of iteration brackets if n is nonzero (true) or missing

Exits iteration brackets if n is nonzero (true) or missing

Jumps to branchpoint x (labeled :x) if n is nonzero (true) or missing

Exits macro if n is nonzero (true) or missing

CONTROL CHARACTERS

Matches any character (SMAEE matches MALE, MADE, and MATE).

Matches character that follows (SMALEE matches only MAEE)

Matches anything but character that follows (SMANTE finds MALE or MADE, but not MATE)

Matches a space or a tab

Matches any character except a letter or number

ese wild-card characters in Search, Replace, Insert, and s.

IC OPERATORS

Multiplication	+	Addition
nteger division	−	Subtraction

LOGICAL OPERATORS

=	Equal
<	Less than
>	Greater than
&	And
!	Or
'	Logical complement (Not)

FUNCTIONS

@n	Returns value of variable n, where n is a digit 0–9
@n,	Puts value of variable n on number stack
@A	Returns value of numeric argument preceding last macro call
@B	Returns value of current edit buffer: 0 if buffer T, 1 if buffer 0, 2 if buffer 1,...,10 if buffer 9, 11 if buffer C (command line)
@C	Returns value of current cursor position in edit buffer (first position is 0)
@D	Returns number of lines scrolled by instant commands ALT-U and ALT-J
@E	Returns value of error flag
@F < flname >	Returns value − 1 if < flname > exists in working directory; returns value 0 if < flname > is not in working directory
@G	Returns length of string argument just referenced by an I, S, R, or C command
@H < strng > $	Returns value 0 if < strng > matches characters at cursor; returns value − 1 or 1 if no match
@I	Returns number of pages read from input file; pages are counted only if delimited by form-feed characters
@J	Returns number of lines available on screen for text display; top three lines are not counted
@K	Returns ASCII value of key entered after a G or QR command
@L	Returns line number of cursor's position in text file if Auto Buffer Mode is on; returns line number of cursor's position in memory if Auto Buffer Mode is off or if editing in buffer other than T
@M	Returns amount (in bytes) of working memory space available
@O	Returns number of pages written to output file; pages are counted only if delimited by form-feed characters
@P	Returns value of absolute memory address of cursor position

	Returns column number of previous tab stop
	Returns remainder of last arithmetic division performed
	Returns top number in number stack; pops number off stack
	Returns ASCII value of character at cursor
	Returns value −1 if Auto Buffer Mode is on; returns value 0 if Auto Buffer Mode is off
	Returns value of the current operating mode (0 if Command Mode, 1 if Insert Mode, 2 if Overtype Mode)
	Returns column number of right margin
	Returns column number of left margin
	Returns column number of next tab stop
	Returns value of byte in memory pointed to by value assigned to variable 9
	Returns column number of current indentation setting
	Returns ASCII value of x, where x is any character

NDS

	Indicates that n strings are stored on the command line
	Rings bell
	Sets control-shift character to that represented by ASCII value of n
	Delays for a time proportional to n
	Sets Type Out Mode where n is 0, 1, or 2
	Sets form-feed character to that represented by ASCII value of n
	Turns garbage stacking off if n is 0; turns garbage stacking on if n is nonzero or missing
	Inserts n spaces at cursor
	Sets current input radix to base n
	Shifts text display up or down n lines without changing cursor position
	Turns Backup Mode off if n is 0; turns Backup Mode on if n is nonzero or missing
	Sets number of lines that instant commands ALT-U and ALT-J scroll to n

Q COMMANDS

QMC	Sends permanent macro file to permanent macro area
QMG	Gets permanent macro file from permanent macro area
nQO	Sets current output radix to base n
nQP	Divides pages into n lines each
nQQ	Shifts text display left or right n columns without changing cursor position
nQR	Redraws screen if n is 0; checks keyboard without redrawing screen if n is −
nQS	Sets the uppercase/lowercase shift character to that represented by ASCII value n
nQT	Sends character represented by ASCII value n to the listing device
nQU	Turns Auto Buffer Mode off if n is 0; turns Auto Buffer Mode on if n is nonzero or missing
nQV	Enables tab fill if n is nonzero or missing; disables tab fill if n is 0
nQW	Turns the command-line error display off if n is nonzero; turns the command-line error display on if n is 0
nQX	Moves cursor to column n on current line; depending on state of free space flag (see QY) cursor might not be able to go past last character in line
nQY	Sets free space flag if n is 0; resets free space flag if n is nonzero
nQZ	Prevents cursor from moving past column n
Q#	Toggles cursor between tag and current cursor position
nQ−	Sets flag to display numbers as signed integers if n is not 0; resets flag (and displays numbers as positive only) if n is 0
nQ/	Sets indentation to column n; if n is missing, Q/ increments indentation by one column and −Q/ decrements it by one column
nQ<	Saves current edit buffer, Format Mode setting, and garbage stack if n is nonzero or missing (they are restored once, after the next error message); this feature is disabled if n is 0
nQ>	Gets character represented by ASCII value n as if you type the character from the keyboard
nQ!	Stores value of n in memory in location pointed to by variable 9
nQx	Sets value of user variable x (0−9) to n; use these 10 user variables with user-written I/O drivers.

L-LINE COMMANDS

Begins formatting control line

Separates control-line commands

Sets left margin to column n

Sets right margin to column n

Deletes all tab stops

Sets a tab stop at column n

Deletes the tab stop at column n

Deletes old tab stops and sets new tab stops every nth columns

Sets indentation to tab stop at column n

ANEOUS CONTROL CHARACTERS

Precedes macro definition in permanent macro file; ends permanent macro file

Precedes first macro in permanent macro file to execute macro when you enter PMATE

Precedes first macro in permanent macro file to execute macro when you enter PMATE; processes operating system commands

Form-feed character

End of file character

echnologies, Inc.
Road
CA 95066 USA

Printed in USA