Programmer's Tool Kit

Volume I

First VICTOR printing February, 1983.

ISBN 0-88182-018-0 Printed in U.S.A.

CONTENTS Programmer's Tool Kit, Volume I

SUPERSEDED BY
Applications Programmer's
* Tool Kit II Volume 1
# Tool Kit II Volume 2

CONTENTS Programmer's Tool Kit, Volume II

OVERVIEW

This kit describes the following utilities:

► FABS/86	A Fast Access B-tree System used to organize data for minimum retrieval time; designed to be called from high-level languages.
► AUTOSORT/86	A comprehensive sort utility that can be used stand-alone or called from application programs.
► PMATE-86	A full-screen, expandable editing system that allows you to create and maintain text files.
► EFONT	A font editor used to define or modify the characteristics of individual keys on the keyboard.
► KEYGEN	A keyboard generator used to define the characteristics of individual keys on the keyboard.
► MODCON	A console modification utility that allows you to set and save keyboard tables and character sets.

IMPORTANT SOFTWARE DISKETTE INFORMATION

For your own protection, do not use this product until you have made a backup copy of your software diskette(s). The backup procedure is described in the user's guide for your computer.

Please read the DISKID file on your new software diskette. DISKID contains important information including:

- The product name and revision number.
- The part number of the product.
- The date of the DISKID file.
- A list of the files on the diskette, with a description and revision number for each one.
- Configuration information (when applicable).
- Release notes giving special instructions for using the product.
- Information not contained in the current manual, including updates, additions, and deletions.

To read the DISKID file onscreen, follow these steps:

1. Load the operating system.
2. Remove your system diskette and insert your new software diskette.
3. Enter —

   TYPE DISKID

   and press Return.
4. The contents of the DISKID file is displayed on the screen. If the file is large (more than 24 lines), the screen display will scroll. Type ALT-S to freeze the screen display; type ALT-S again to continue scrolling.

FABS/86

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

First VICTOR printing February, 1983.

ISBN 0-88182-011-3 Printed in U.S.A.

CONTENTS

CHAPTERS

1

AN OVERVIEW OF FABS/86

GENERAL INFORMATION 1.1

One of the biggest problems when you compile data with a computer is how to retrieve that data once you've compiled it. How can you locate the information you want in the least amount of time using the simplest search procedure? The FABS/86 (Fast Access Btree Structure) program can help you solve this problem.

FABS/86 is an assembly language program module that uses key files for fast data retrieval with large data files. FABS/86 assigns input data to key files, and arranges those files in balanced trees (Btrees) to speed data retrieval. When you need to access a record in your data file, FABS/86 searches down the tree to locate that record. If necessary, FABS/86 rearranges the key files on that tree for easier access to your data base.

You should have at least a basic knowledge of programming in a high-level language (such as Pascal or BASIC) before you start using FABS/86. You should also have a rudimentary knowledge of how files are maintained in a computer system. If you are a beginning programmer, you should find FABS/86 easy to understand once you have mastered the data file concepts in your programming language.

DATA STORAGE AND RETRIEVAL 1.2

Whether you are using a computer or a file cabinet, the first step in the process of storing data is about the same. You give each piece of data a key name (an account number, subject, and so on) that distinguishes it from other pieces of data in the same filing system. The difference between computers and file cabinets starts when you actually put the new piece of data into the system. With a file cabinet, you shift all of the other records in the cabinet to make room for the new data. When you're storing data

on a diskette or hard disk, however, it takes too long for the computer to move all of the other records. This problem is generally overcome with one of the following techniques:

1. The new data is kept separate from the existing records until the insertion process is complete. After you've entered all of the new data, it is sorted into the existing data file.

2. An additional file (a key file) is created. This key file contains a field (key) into which the data is sequenced. It also contains an associated record pointer (the record number) of the data record that contains the key. Using this system, only the smaller key file has to be shuffled around each time you enter new data. Any data you enter is easily retrievable: Once the key is found, the data record pointer is used to access the data you want.

Both of these techniques have their problems, however. If you're using the first method, for example, your data is normally retrieved by using a "binary search." A binary search divides the data file in half, and determines whether the desired field (key) is in the upper or lower half of the file. Then, the search determines if the key is in the upper or lower half of the half chosen in the first step and so on, until the key is found or the file is exhausted. With large data files, a binary search can take a long time.

A binary search also causes problems during the insertion process, when you need to make sure that none of the inserted records are already in your file. In this case, all unsorted data inserts are searched to ensure that they do not already exist in your file.

Another method maintains keys in a sorted key file and uses an overflow file for inserted keys not yet sorted into the key file. The binary search technique is used to find the key and the associated data record pointer. If the key is not found in the sorted key file, the overflow file is searched. As the overflow file gets bigger, retrieval gets slower. Periodically, you must resort the key file to incorporate the keys in the overflow file.

THE FABS/86 DIFFERENCE

FABS/86 also maintains keys in a sequential key file — but that is where the similarity ends. Instead of storing data in a linear manner, the FABS/86 key file is a multi-path balanced tree. This design makes FABS/86 well suited for the maintenance and manipulation of very large data files (the most difficult).

With FABS/86, the data file and key file space is dynamically allocated — that is, the files grow as needed. Although FABS/86 does not actually read or write to your data file, it does provide you with the record number for all of your data file reads and writes.

Suppose you have a data file that you want to access by the name field. To insert a new record (with a given name) into the file, you must call FABS/86 with the Insert command and the key. In about two seconds, FABS/86 returns the number of the data record where you must write the data associated with that key. After another quarter-second or so, your key file and data file are in perfect order. (These times assume that you are using a floppy disk system. A hard disk is faster.)

If you have enough room on your diskette or hard disk (and you are not limited by your programming language), you can insert 50 thousand keys or more without much effect on the access time. Typically, this time is one second or less to search for the key and read the data record. Repeated accesses normally take about a quarter of a second each.

You should note that there are no overflow files associated with FABS/86. The estimated insertion times mentioned include reorganizing the key file (if necessary). Any portions of the key file that are changed during insertion are saved on diskette (or hard disk) so that the key file is always current. Sorting is not required at any time on either the data file or the key file.

USING FABS/86 TO RETRIEVE A RECORD

Since FABS/86 knows which data record has been attached to what key, finding a record is easy — all you have to do is tell FABS/86 to search for

a particular key. FABS/86 gives you the random data record number that you must read to get the desired record. Each time you delete a record, FABS/86 retains the record numbers for automatic re-use when you insert more records.

FABS/86 also lets you look for groups of records, rather than just a specific record. (This is called a "generic" search.) For example, you could do a generic search for every record with a name field that starts with the letter "M". After you give the proper instructions, FABS/86 finds the record number of the first occurrence of a name field starting with "M". A Search Next command (discussed in Chapter 2) continues the search sequentially through the name field.

A generic search is also useful in accessing data that is sequenced in several levels. For example, the data file you're searching might be in alphabetical order by state, then by city under each state, and by ZIP code under each city. First, you concatenate the state, city and zip code fields to form the insert key. The Search Generic command is then used along with the Search Next command to access only those records associated with a particular state, or a particular state and city.

With FABS/86, you can retrieve data sequenced on more than one key by using a single key file. When the key file is created, you specify the number of primary keys that you want for the data file. When inserting or deleting, you must specify all of the key values for the data record. When searching your file, you specify which primary key number you want.

KEYS

Duplicate Keys

FABS/86 lets you assign the same value to more than one key ("duplicate" keys). When you want to access a record assigned to a duplicate key, FABS/86 supplies the number of the first occurrence of that key in the index file. Use the Search Next command to find the next occurrence of the key. To access a series of duplicate keys, test the key of each data record you read against the original key to see if you are still in the block of duplicates. This

procedure is also used when you need to read through a block to determine which keys are to be deleted.

Multiple Keys

FABS/86 also supports multiple primary keys — that is, an area of the data file can be accessed in ascending or descending order by more than one key. The number of primary keys you can use is limited only by the length of the Insert and Delete command strings (255 bytes). Of course, the more keys you use, the longer it takes to insert or delete them. The search time should not be seriously affected, however.

When you create an index file, you must specify the number of primary keys. When inserting and deleting keys, all of the primary keys must be specified (except for the Replace command).

FABS/86 also lets you use duplicate multiple primary keys. You should be aware, though, that using this type of key presents some problems. If you want to delete a duplicate multiple key, you must decide which of the records is to be deleted. Then, you have to extract all of the primary keys from that record and use them to form the Delete command.

ASCII Keys

Keys are normally maintained in the key file as ASCII characters. (This mode is specified by entering an "A" for the KEYTYPE when you create the key file.) Any key having less than the maximum length is padded with zeros to bring it up to maximum length. When more than one primary key is specified, the maximum length applies to all keys. Each key occupies the maximum space.

Integer Keys

When creating a key file, you can specify the KEYTYPE as I (for Integer). With an integer file, FABS/86 converts the keys to a 2-byte integer numeric value. There are some limitations that must be observed when you use Integer keys: Any keys specified in the command string must be ASCII strings with a range of 0 to 65535. They cannot contain nulls, signs or other characters.

The maximum key length will be forced to 2 bytes. When a key file is specified as Integer, all the primary keys must be integer. Also, generic searches are not permitted with integer keys.

Maximum Number of Keys

With a balanced tree, it is impossible to predict the maximum number of keys you can use in a data file. This number depends on the length of the key, and on the sequence in which your keys are inserted. However, you can establish a "worst case" and a "best case" for each length; the average gives you an idea of how many keys of a particular length can probably be used.

FABS/86 has a constant node length of 512 bytes and can have up to 5 levels. The root node can have anywhere from one key to the maximum number of keys. All other nodes will be between half full and full depending upon the insertion sequence.

The following table gives you an idea of how many keys can be used with various key lengths. (N is the minimum number of keys per node — 512 bytes.)

Exhibit 1a: Maximum Keys

KEY LENGTH	N	WORST CASE	BEST CASE	AVERAGE
2	36	107,136	214,272	160,704
4	28	83,328	166,656	124,992
6	23	68,448	136,896	102,672
8	19	56,544	113,088	84,816
10	16	47,616	95,232	71,424
14	13	38,688	77,376	58,032
20	10	29,760	59,520	44,640
30	7	20,832	41,664	31,248
40	5	14,250	29,760	22,005
50	4	5,620	23,808	14,714

2

FABS/86 COMMANDS

The following commands can be executed with FABS/86:

Exhibit 2a: Commands

COMMAND	DESCRIPTION
B	Build key file
K	Close key file
C	Create key file
D	Delete key(s)
M	Get maximum key length
Q	Get next record number
U	Get number of open deletes
H	Get number of primary keys
T	Get number of records
I	Insert key
O	Open key file
R	Replace key
S	Search for key
F	Search for first key
G	Search for generic key
L	Search for last key
N	Search for next key
P	Search for previous key
W	Write page map

This chapter discusses the functions of these FABS/86 command strings. Each string is described in the following manner:

- First, you are given the format to use when you enter the command. Each element of the string is explained.
- Next, you are told the operation the command performs and when the string should be used. When appropriate, you are referred to other command strings that can be used with the one being explained.
- Last, any parameters returned by the string are explained. This tells you the meaning of each of the parameters that FABS/86 returns when you use the command string.

2.1 BUILD (B)

The command string is:

CMND$ = "B\FN\" + PK1$ + "\" + ... + "\" + PKn$

2

where:

B is the command.

FN is the file number (1 to 6).

PKn$ is the value of the nth primary key.

This command is identical to the Insert (I) command, except that Build does not write the map file to your diskette. All other FABS/86 commands update the diskette before returning to the calling program.

Because the Build command does not write to the map file (as the Insert command does), you save time if you use Build instead of Insert when building key files for large data files.

The Write Map File command must be executed after a series of Build commands to ensure that the correct map data has been entered on diskette. No other FABS/86 command should be executed during this procedure.

When in doubt use the Insert (I) command.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The date record number

ADRKEY = No significance

CLOSE (K) 2.2

The command string is:

CMND$ = "K\FN"

where:

K is the command.

FN is the file number (1 to 6).

The Close command closes a key file when you reach the end of the host language program. (This is the only time that you need to close a key file.) The index file on your diskette is updated after each FABS/86 operation (unless you used the Build command).

Parameters returned:

ERRF% = Error/Warning code

RECNO = No significance

ADRKEY = No significance

CREATE (C) 2.3

The command string is:

CMND$ = "C\[d:]filename[.ext]\MAXKL\NPK\KT\FN"

where:

C is the command.

MAXKL is the maximum key length (100 bytes max). The usual key length is 8 to 10 bytes.

NPK is the number of primary keys for this file.

KT is the key type (I = Integer, A = ASCII).

FN is the file number (1 to 6).

Use Create to create a key file (filename.ext) and a map file (filename.MAP) with the attributes specified in the command string. The key file is opened for access under the file number contained in the command string. If a file with the same name already exists, that file is deleted.

Parameters returned:

ERRF% = Error/Warning code

RECNO = No significance

ADRKEY = No significance

2.4 DELETE (D)

The command string is:

CMND$ = "D\RN\SBDFL\FN\" + PK1$ + "\" + ... + "\" + PKn$

where:

D is the command.

RN is the record number of the data record containing the keys.

SBDFL is the prompt "Search Before Delete Flag (Y/N)." If you answer yes, all primary keys are searched to ensure their presence before any are deleted. This process protects your key file against faulty programs.

FN is the file number (1 to 6).

PKn$ is the value of the nth primary key.

Use Delete to delete the specified keys from the key file and return the associated data record number in the data file. You should put "deleted" into some field of the data record if the key file is destroyed. Doing so lets you rebuild the key file, excluding the deleted data records.

FABS/86 maintains pointers to all deleted records and reclaims these records on future inserts on a last-in, first-out basis.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The deleted data record number

ADRKEY = No significance

GET MAXIMUM KEY LENGTH (M) 2.5

The command string is:

CMND$ = "M\FN"

where:

M is the command.

FN is the file number (1 to 6).

This command causes FABS/86 to return the maximum key length permitted in the key file. (The maximum key length was specified by a Create command.) Any attempt to insert a longer key causes a syntax error.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The maximum key length

ADRKEY = No significance

2.6 GET NEXT RECORD NUMBER (Q)

The command string is:

```
CMND$ = "Q\FN"
```

where:

2

Q is the command.

FN is the file number (1 to 6).

When you use the Get Next Record Number command, FABS/86 returns the record number to be assigned the next time you use the Insert command. (This assumes that no Delete command is used prior to the Insert command.) If there are no unreclaimed deleted data records, this command returns the next available data record in the file. If there are unreclaimed deletes, the record number of the last delete is returned.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The next data record number

ADRKEY = No significance

2.7 GET NUMBER OF DELETES (U)

The command string is:

```
CMND$ = "U\FN"
```

where:

U is the command.

FN is the file number (1 to 6).

This command returns the number of unreclaimed deleted data records. This number tells you how many records can be inserted before your data file expands.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The number of deleted records

ADRKEY = No significance

GET NUMBER OF PRIMARY KEYS (H) 2.8

The command string is:

```
CMND$ = "H\FN"
```

where:

H is the command.

FN is the file number (1 to 6).

This command returns the number of primary keys in the key file. (This number is specified in the Create command.)

Parameters returned:

ERRF% = Error/Warning code

RECNO = The number of primary keys

ADRKEY = No significance

2.9 GET NUMBER OF RECORDS (T)

The command string is:

```
CMND$ = "T\FN"
```

where:

2

T is the command.

FN is the file number (1 to 6).

Get Number of Records returns the total number of records in your data file, including the unreclaimed deleted records. To determine the number of active data records, subtract the value returned by Get Number of Deletes (U).

Parameters returned:

ERRF% = Error/Warning code

RECNO = The number of records

ADRKEY = No significance

2.10 INSERT (I)

The command string is:

```
CMND$ = "I\FN\" + PK1$ + "\" + ... + "\" + PKn$
```

where:

I is the command.

FN is the file number (1 to 6).

PKn$ is the nth primary key value.

Use this command to insert keys into the key file. The number of primary keys included in the command must equal the number you specified in the Create command. Duplicate keys are permitted. Variable length keys are also permitted, but are padded with zeros (not spaces) to bring them up to the maximum key length. If you want the keys padded with spaces, you must enter the spaces yourself. Each key in your key file has the maximum length specified in the Create command.

When control is returned to the calling program, you should write the entire data record to the data file at the record number specified by RECNO.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The data record number

ADRKEY = No significance

OPEN (O) 2.11

The command string is:

CMND$ = "O\[d:]filename[.ext]\FN"

where:

O is the command.

FN is the file number (1 to 6).

Use this command to open an existing key and map file for access. You can open up to six key files at a time.

Parameters returned:

ERRF% = Error/Warning code

RECNO = No significance

ADRKEY = No significance

2.12 REPLACE (R)

The command string is:

```
CMND$ = "R\PKN\RN\FN\" + OLDKEY$ + "\" + NEWKEY$
```


where:

R is the command.

PKN is the primary key number.

RN is the record number of OLDKEY$.

FN is the file number (1 to 6).

OLDKEY$ is the value of the key to be replaced.

NEWKEY$ is the new key value.

Use this command to replace a single key with another key having the same record number and primary key number. The returned record number is the same as the specified record number.

Parameters returned:

ERRF% = Error/Warning code

RECNO = Same as specified record number

ADRKEY = No significance

2.13 SEARCH (S)

The command string is:

```
CMND$ = "S\PKN\FN\" + KEY$
```

where:

S is the command.

PKN is the primary key number.

FN is the file number (1 to 6).

KEY$ is the value of the key.

This command returns the record number of the specified key string (KEY$). The record number associated with the first duplicate is returned if there are duplicate keys with the value KEY$. You can use the Search Next (N) command to access the others. (You should test each time to see if the key value is equal to KEY$.)

Parameters returned:

ERRF% = Error/Warning code

- If ERRF% = 0, KEY$ was found.
- If ERRF% = 12, KEY$ was not found and the value of KEY$ is between the first key and the last key.
- If ERRF% = 13, KEY$ was not found and the value of KEY$ is less than the value of all existing keys. The record number of the first key is returned.
- If ERRF% = 15, KEY$ was not found and the value of KEY$ is greater than the value of all existing keys. The record number of the last key is returned.
- If ERRF% = 16, there are no keys in the key file.

RECNO = The appropriate record number

ADRKEY = The FABS/86 memory address where the key can be found

2.14 SEARCH FIRST (F)

The command string is:

```
CMND$ = "F\PKN\FN"
```

where:

2

F is the command.

PKN is the primary key number.

FN is the file number (1 to 6).

Search First returns the number of the data record containing the smallest key value for the specified primary key.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The record number

ADRKEY = The FABS/86 memory address where the key can be found

2.15 SEARCH GENERIC (G)

The command string is:

```
CMND$ = "G\PKN\FN\" + KEY$
```

where:

G is the command.

PKN is the primary key number.

FN is the file number (1 to 6).

KEY$ is a left-justified partial key.

Search Generic returns the number of the first occurrence of the left-justified partial key. This number helps you find the start of a category of keys. The Search Next (N) command can then be used to access the remainder of the category of keys. Remember to test each time to see if the key in the data file is the same as KEY$.

The Search Generic command cannot be used with Integer keys.

See the discussion of the Search (S) command for the values of the error/warning code (ERRF%) returned if KEY$ is not found in the key file.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The record number

ADRKEY = The memory address (in FABS/86 segment) where the key can be found

SEARCH LAST (L) 2.16

The command string is:

```
CMND$ = "L\PKN\FN"
```

where:

L is the command.

PKN is the primary key number.

FN is the file number (1 to 6).

Search Last returns the number of the data record that contains the largest key value for the specified primary key.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The record number

ADRKEY = The FABS/86 memory address where the key can be found

2.17 SEARCH NEXT (N)

The command string is:

```
CMND$ = "N\FN"
```

where:

N is the command.

FN is the file number (1 to 6).

Search Next returns the number of the data record containing the next key in sequence. This command is reliable only if the last command for the same file number was one of the Search commands. The Search Next command does not cross over primary key boundaries. Error code 15 appears when Search Next reaches the end of the group of primary keys.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The record number

ADRKEY = The FABS/86 memory address where the key can be found

SEARCH PREVIOUS (P) 2.18

The command string is:

```
CMND$ = "P\FN"
```

where:

P is the command.

FN is the file number (1 to 6).

Search Previous returns the number of the data record containing the previous key in sequence. This command is reliable only if the last command for the same file number was one of the Search commands. The Search Previous command does not cross over primary key boundaries. Error code 13 appears when the bottom of the group of primary keys is reached.

Parameters returned:

ERRF% = Error/Warning code

RECNO = The record number

ADRKEY = The memory address where the key can be found

WRITE PAGE MAP (W) 2.19

The command string is:

```
CMND$ = "W\FN"
```

where:

W is the command.

FN is the file number (1 to 6).

Use this command to write the page map to your diskette (or hard disk) after a series of Build (B) commands. The Write Page Map command should be executed immediately after the Build commands; no other FABS/86 command should be used between the Build commands.

Parameters returned:

ERRF% = Error/Warning code

RECNO = No significance

ADRKEY = No significance

3

USING FABS/86 WITH PROGRAMMING LANGUAGES

FABS/86 AND THE MS-BASIC INTERPRETER 3.1

The FABS86M.COM module is loaded and fixed in memory (until the next restart) by executing it as you would any transient program:

FABS86M

This loads the module and displays the sign-on along with the "FSEG = &HXXXX" statement showing the segment where the module was loaded. This segment is different for different system configurations.

When you use this procedure, FABS86M.COM should always be the first program you load when your computer is powered up or restarted. By doing this, you ensure that the FABS86M.COM module is loaded in the same place.

WARNING: Don't load FABS86M.COM more than once between restarts. It will load higher each time you load it.

The FABS86M.OBJ file is a relocatable module used to link FABS/86 to compiled (.OBJ) files generated by the MS-BASIC Compiler, MS-COBOL, MS-Pascal and other compiled languages. Public declarations in the FABS86M.OBJ module provide linkages to the calling programs.

CALLING FABS/86 FROM MS-BASIC

Before you can begin the FABS/86 calling subroutine, your MS-BASIC program must have the FSEG statement declared. This statement is displayed when the FABS86M.COM module is loaded:

```
FSEG = &Hxxxx
```

where

xxxx is the load segment.

Your BASIC program must contain the following FABS/86 calling subroutine:

```
DEF SEG = FSEG
FABS86M = &H5
CALL FABS86M(CMND$,ERRF%,RECNO%,ADRKEY%)
DEF SEG
RECNO = RECNO%
IF RECNO<0 THEN RECNO = RECNO + 65536
RETURN
```


(Chapter 2 defines the CMND$, ERRF%, RECNO% and ADRKEY% parameters for each FABS/86 command string.)

The value of the key can be returned after a FIRST, LAST, NEXT, or PREVIOUS command by using the following subroutine:

```
RKEY$ = ""
ADRKEY = ADRKEY%
IF ADRKEY<0 THEN ADRKEY = ADRKEY + 65536!
FOR I = ADRKEY TO ADRKEY + MAXKLEN - 1
   DEF SEG = FSEG
   RCHAR = PEEK(I)
   DEF SEG
   RKEY$ = RKEY$ = CHR$(RCHAR)
NEXT I
```

where:

MAXKLEN is the maximum key length (specified in the Create command).

RKEY$ is the key value.

To execute FABS/86, you simply define a command string (CMND$) for the particular command you desire and then GOSUB to the calling subroutine which actually calls FABS/86.

TEST PROGRAMS

The following test programs are included on the FABS/86 program diskette to show the capabilities of FABS86M:

- FABS86M.COM: The FABS/86 module.
- FABSBLD.BAS: Builds test key and data files.
- FPRINT.BAS: Displays the data file.
- FABSTEST.BAS: Demonstrates the execution of FABS/86 commands.

The FABSBLD.BAS program constructs the FTEST.DAT, FTEST.KEY and FTEST.MAP files needed by the FABSTEST.BAS program. If they are not already present on your diskette, run FABSBLD.BAS to create them.

Follow these steps to run the test programs:

1. Copy the test programs from the FABS/86 program diskette to a diskette that contains the MS-DOS operating system. Save the original for backup.

2. Reboot your computer with the MS-DOS diskette in drive A. Load FABS/86 by typing FABS86M after the A> prompt.

3. Ensure that the FSEG statement displayed when FABS/86 is loaded is the same as the one at the beginning of the FABSBLD.BAS, FPRINT.BAS, and FABSTEST.BAS programs. If the FSEG statement is not the same, change the others to reflect the segment where FABS86M.COM was loaded.

4. Transfer your MSBASIC.COM (hereafter called MS-BASIC) to the MS-DOS diskette.

5. Enter:

   MSBASIC FABSBLD

   to build the test files. It takes about 20 minutes to insert the 1000 keys (500 records of 2 primary keys each). About half of this time is used to generate the keys randomly. The keys have a maximum length of 10 bytes. The data file contains the 2 primary keys and a 12-byte string.

6. Enter:

   MSBASIC FPRINT

   to run the FPRINT.BAS program. When prompted, select key sequence, primary key 1, and ascending order. The data record number is displayed on the right.

7. Run the FABSTEST.BAS program by entering:

   MSBASIC FABSTEST

   The list of FABS/86 commands is displayed.

   When prompted, select Generic search, primary key 1, and "R" for the key value. The data record for the first key beginning with "R" is returned. Use the Previous and Next commands to verify the key.

   You can insert and delete records at will. Remember that deleted records are reused on a last-deleted, first-reused basis.

CALLING FABS/86 FROM THE MS-BASIC COMPILER

The FABS86M.OBJ object file lets you link FABS/86 with compiled programs using the LINK.EXE program. The compiled BASIC program must contain the following FABS/86 calling subroutine:


```
CALL FABSMB(CMND$,ERRF%,RECNO%,ADRKEY%)
RECNO = RECNO%
IF RECNO<0 THEN RECNO = RECNO + 65536!
```

See Chapter 2 for an explanation of CMND$, ERRF%, RECNO%, and ADRKEY% for each command string.

The actual key value can be returned after any Search command by using the following subroutine:

```
        ADRKEY = ADRKEY%
        IF ADRKEY<0 THEN ADRKEY = ADRKEY + 65536!
        CALL GFSEG(FSEG%)
        FSEG = FSEG%
        IF FSEG<0 THEN FSEG = FSEG + 65536!
        RKEY$ = ""
        FOR I = ADRKEY TO ADRKEY + MAXKLEN - 1
            DEF SEG = FSEG
            RCHAR = PEEK(I)
            DEF SEG
            IF RCHAR = 0 THEN YYY
            RKEY$ = RKEY$ + CHR$(RCHAR)
        NEXT I
YYY     REM  RKEY$ = ACTUAL KEY VALUE
```

TEST PROGRAMS FOR THE BASIC COMPILER

The following example programs are included on the distribution diskette.

- FABS86M.OBJ: The FABS/86 relocatable object module.
- MCBUILD.BAS: Test build program.
- MCBUILD.OBJ: The object file.
- MCTEST.BAS: FABS/86 test program.
- MCTEXT.OBJ: The object file.
- MCPRINT.BAS: Prints the files.
- MCPRINT.OBJ: The object file.
- FTEST.KEY: The test key file.
- FTEST.MAP: The test map file.
- FTEST.DAT: The test data file.

Follow these steps to run a test program:

1. Copy the test programs from the distribution diskette to another that contains the MS-DOS operating system. Save the original for backup.

2. Using LINK.EXE, link the following .OBJ files to form the indicated RUN (.EXE) files. (See the *MS-LINK Section of the Programmer's Tool Kit, Volume II* for instructions on using LINK.EXE.)

RUN FILE	OBJECT FILES
MCPRINT.EXE	MCPRINT.OBJ + FABS86M.OBJ
MCTEST.EXE	MCTEST.OBJ + FABS86M.OBJ
MCBUILD.EXE	MCBUILD.OBJ + FABS86M.OBJ

3. Run the MCPRINT.EXE program by entering:

   MCPRINT

   Answer the prompts that follow by selecting key sequence, primary key

1, and ascending order. Observe the speed with which FABS/86 displays the data file in key sequential order. The data record is displayed on the right.

4. Run the MCTEST.EXE program by entering:

   MCTEST

   The key and data files are opened and the list of FABS/86 commands is displayed. When prompted, select Generic search, primary key 1, and "R" for the key value. The data record for the first key beginning with R is returned. Use the Previous and Next commands to verify the key. You can insert and delete records at will. Remember that deleted records are reused on a last-deleted, first-reused basis.

5. If you want to build a larger set of test files, change the FOR-NEXT loop in the MCBUILD.BAS program to reflect the number of records you desire. Then, use BASCOM.COM to compile it and obtain the MCBUILD.OBJ file for linking with the FABS86M.OBJ file to form MCBUILD.EXE. Then, run MCBUILD.EXE to build the new set of test programs.

USING FABS/86 WITH MS-PASCAL 3.3

The following is a typical calling program from Pascal:

```
program pastest;
type
     cmdstring = lstring(255);
var
     errf:              word;
     recno:             word;
     cmnd:              cmdstring;

function fbspas (vars cmd: lstring;
          vars err: word) : word; external;
begin
cmnd:= "C\FTEST.KEY\10\2\A\3";
recno:= fbspas (cmnd,errf);
end.
```

This program creates the FTEST.KEY and FTEST.MAP files. The key file is set for two primary keys with a maximum key length of ten bytes. Both keys are ASCII keys; the key file is opened as file number 3.

The returned parameter (recno) has no significance with a Create command. The error code (errf) should be tested for zero to ensure that there was no error.

An additional entry point (FBPAS1) is provided. It returns the key address (in the FABS segment) in addition to the preceding parameters. The function call is:


```
recno: = fbpas1 (cmnd,errf,keyadr);
```

To get the KEYADR segment, enter:

```
fseg: = gfseg1
```

You can access the key if you know the segment and the offset of the key.

FABS86M.OBJ is linked to your Pascal object file using LINK.EXE.

3.4 FABS/86 AND MS-FORTRAN

To call FABS/86 from MS-FORTRAN, you must define the command string (CMND) and execute an external function as follows:

```
C       TEST PROGRAM FOR CALLING FABS/86

        PROGRAM FBSTST
        CHARACTER *80 CMND
        INTEGER *2 ERRF, RECNO, KEYADR, FBSFOR

        CMND = "C\TEST.KEY\10\2\A\3"
        RECNO = FBSFOR(CMND,ERRF,KEYADR)

        WRITE(*,200) "ERCODE =" ERRF
200     FORMAT(1X,A8,I6)
        STOP
        END
```

This program creates empty FABS/86 key and map files (TEST.KEY and TEST.MAP) which are ready for keys to be inserted. The key file has two primary ASCII keys of ten bytes each; it is opened for access as file number 3.

With MS-FORTRAN, the command string must be terminated with an up-arrow (^) because no length byte is passed with the string. The returned parameter RECNO (normally record number) is returned in the AX register as the returned function. FABS/86 treats RECNO as a 16-bit positive number with a range of 0 to 65535. MS-FORTRAN may return a negative number if greater than 32767. In this case, you would add 65536 to RECNO to place it in the range of 0 to 66535.

The error code is returned in ERRF. This will be zero if there is no error or warning.

The pointer to the actual key in memory after any Search command is returned in KEYADR. This is the temporary address of the key in the FABS segment. You can get the FABS segment by using the following external function:

```
FSEG = GFSEG1
```

where:

FSEG is the FABS segment.

The FABS86M.OBJ module is linked to your FORTRAN object file using LINK.EXE.

3.5 FABS/86 AND MS-COBOL

To call FABS/86 from MS-COBOL, you must define the command string (CMND) and call an external procedure as shown by the following sample program.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBTST.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77    CMND PIC X(80) VALUE IS 'C\TEST.KEY\10\2\A\1'
77    ERRF     PIC  99999  COMP-0  VALUE 0.
77    RECNO    PIC  99999  COMP-0  VALUE 0.
77    KEYADR   PIC  99999  COMP-0  VALUE 0.
77    ERC      PIC  99999.
77    RN       PIC  99999.
77    KA       PIC  99999.
PROCEDURE DIVISION.
MAIN.
      CALL "FBSCOB" USING CMND, ERRF, RECNO, KEYADR
MOVE ERRF TO ERC.
MOVE RECNO TO RN.
MOVE KEYADR TO KA.
DISPLAY 'ERRF =' ERC.
DISPLAY 'RECNO =' RN.
DISPLAY 'KEYADR =' KA.
STOP RUN
```

This program creates a TEST.KEY and TEST.MAP file on the disk and opens the key file for access as file number 1. The key file has two ASCII primary keys of ten bytes each.

The command string (CMND) must be terminated with an up-arrow (^) since MS-COBOL does not provide a length byte.

The KEYADR pointer gives the offset of the actual key in the FABS segment. It is important for all Search commands. The FABS segment is returned by the following external procedure:

```
CALL 'GFSEG' USING FSEG
```

where:

FSEG is the FABS segment.

The FABS86M.OBJ module is linked to your COBOL Object file using LINK.EXE.

4

FABS/86 ERROR AND WARNING CODES

Exhibit 4a describes the error and warning codes displayed by FABS/86. Except for 0, any error code numbered 12 or less is normally a fatal error. The same is true of error codes numbered 16 or higher. All error and warning codes are returned from the calling subroutine in the variable ERRF%.

Exhibit 4a: FABS/86 Error and Warning Codes

WARNING NUMBER	DESCRIPTION
0	The FABS/86 operation was successful.
4	Improper key for Integer key file.
5	Attempted Generic search on Integer key.
6	Key not found on Delete.
7	Incorrect number of primary keys.
8	Syntax error in command string.
9	No more key space.
10	Input key larger than maximum.
11	Tried to access unopened file.
12	Key specified was not found, but smaller and larger keys were found.
13	Key not found. Key smaller than all keys.
15	Key not found. Key larger than all keys.
16	Key not found. No keys in key file.
22	File not present when opened.
23	Out of directory space.
24	Diskette full.
25	Write error.
26	File not present when closed.
27	Read error. End of file.

A

FABS/86 PUBLIC INTERFACES AND ABSOLUTE OFFSET ENTRY POINTS

PUBLIC INTERFACES A.1

The following are the FABS/86 PUBLIC entry and access locations:

Exhibit Aa: FABS/86 PUBLIC Entry and Access

PUBLIC SYMBOL	NORMALLY USED BY
FABSMB	MS-BASIC Compiler
FBSPAS	MS-Pascal (doesn't return key address)
FBPAS1	MS-Pascal (returns key address)
FBSFOR	MS-FORTRAN
GFSEG	General
GFSEG1	General
KEYADR	General

Each is discussed in the following sections.

FABSMB ENTRY POINT

On entering at FABSMB, nonsegmented pointers representing the addresses of CMND, ERROR CODE, RECNO, and KEYADR (in that order) must have been pushed on the stack.

All pointers are assumed to be offsets to the Data Segment register (DS) at entry. CMND must be four bytes: the first two bytes contain the length of the command string; the next two bytes contain the offset of the command string relative to the DS register at entry.

At exit from FABS/86, the four pointers (8 bytes) are removed from the stack and the returned parameters are placed in the specified addresses. The segment associated with the key address can be obtained by calling either GFSEG or GFSEG1.

FBSPAS ENTRY POINT

On entering at FBSPAS, segmented pointers (8 bytes) representing the addresses of CMND and ERROR CODE (in that order) must have been pushed on the stack. The first byte of CMND must contain the length followed by the actual string of characters.

At exit from FABS/86, the two pointers (8 bytes) are removed from the stack and the error code is placed in the specified offset and segment. The returned parameter (RECNO) is placed in the AX register as the returned function.

If the key address is needed, use the FBPAS1 entry point.

FBPAS1 ENTRY POINT

On entering at FBPAS1, segmented pointers representing the addresses of CMND, ERROR CODE and KEYADR (in that order) must have been pushed on the stack.

The first byte of CMND must contain the length of the command string followed by the actual string of characters.

At exit from FABS/86, the three pointers (12 bytes) are removed from the stack, and the error code and key address are placed in the specified segment and offset. You must use either GFSEG or GFSEG1 to get the FABS segment (same as the KEYADR segment) if you want to access the key. The returned parameter (RECNO) is returned in the AX register.

FBSFOR ENTRY POINT

On entering at FBSFOR, segmented pointers representing the addresses of CMND, ERROR and KEYADR (in that order) must have been pushed on the stack. The CMND pointer must point to the first actual byte of the command string. The command string must be terminated with an up-arrow (^), so that FABS/86 can determine the string length.

At exit, the three segmented pointers (12 bytes) are removed from the stack, and ERROR CODE and KEYADR are placed in the specified addresses. You must use either GFSEG or GFSEG1 to get the KEYADR segment (same as the FABS segment) to access the key value. The returned parameter (RECNO) is returned in the AX register.

FBSCOB ENTRY POINT

On entering at FBSCOB, nonsegmented pointers representing the addresses of CMND, ERROR CODE, RECNO, and KEYADR (in that order) must have been pushed on the stack. All pointers are assumed to be offsets to the Data Segment register (DS) at entry. The command string (CMND) must be terminated with an up-arrow (^) so that FABS/86 can determine the string's length.

At exit the parameters ERROR CODE, RECNO, and KEYADR are placed in the specified addresses. You must use either GFSEG or GFSEG1 to get the KEYADR segment (same as FABS segment) to access the actual value of the key.

GFSEG ENTRY POINT

This procedure returns the FABS segment in the specified variable as follows:

```
CALL    GFSEG(FSEG)
```

On entering at GFSEG, a nonsegmented pointer must have been pushed on the stack. The FABS segment is placed in the specified address relative to the Data Segment register (DS) at entry.

At exit, the pointer (2 bytes) is removed from the stack.

GFSEG1 ENTRY POINT

This is used as an external function to return the FABS segment as follows:

```
FSEG = GFSEG1
```

When this function is called, FABS/86 returns the FABS segment in the AX register.

KEYADR PUBLIC VARIABLE

The FABS/86 internal variable KEYADR contains the temporary address of the actual key value (in the FABS segment) after any normal Search command.

KEYADR resides at offset 28 (hex) in the FABS segment. It consists of a 2-byte offset pointer followed by a 2-byte variable containing the KEYADR segment (same as the FABS segment).

A.2 ABSOLUTE OFFSET ENTRY POINTS

Absolute entry points let compiled languages call an absolute offset in the FABS/86 segment, instead of a PUBLIC entry point. The FABS.COM

module can then be loaded and fixed in memory, as is normally done with the MS-BASIC Interpreter. By loading this module, you eliminate the need to link the FABS.REL file into all programs that use large amounts of disk space.

The FABS.COM file is loaded by typing:

FABS86M

following the A> prompt.

The FABS segment is displayed when the FABS module is loaded. The absolute entry points are offsets in the FABS segment. Exhibit Ab shows the absolute offsets to call for the various languages.

Exhibit Ab: FABS/86 Absolute Offsets

LANGUAGE	EQUIVALENT PUBLIC SYMBOL	OFFSET (DECIMAL)
MS-BASIC Interpreter	None	5
MS-BASIC Compiler	FABSMB	8
MS-Pascal	FBSPAS	11
Any	GFSEG	14
MS-FORTRAN	FBSFOR	17
MS-COBOL	FBSCOB	20
MS-Pascal	FBPAS1	23
Any	GFSEG1	26

A

If you are using the MS-BASIC Compiler, for example, you load the FABS module and note the load segment:

FSEG = &Hxxxx

where:

xxxx is the start segment of the beginning of the FABS module.

The following statement must be included in your BASIC program just prior to calling FABS/86:

```
DEF SEG = FSEG
```

Then execute an absolute call with an offset of 8. You return to the BASIC segment: DEF SEG.

AUTOSORT/86

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

First VICTOR printing February, 1983.

ISBN 0-88182-010-5 Printed in U.S.A.

CONTENTS

EXHIBITS

CHAPTERS

1

GENERAL INFORMATION

AUTOSORT/86 is a sort/merge/select utility designed for use with very large files that have fixed-length fields within fixed-length records. It is compatible with Microsoft's MS-DOS operating system, and it supports string fields and the MS-BASIC Integer, Single Precision, and Double Precision fields.

AUTOSORT/86 FEATURES 1.1

Nine modes of sort/merge/select are available. The mode is specified in the command string (CMND$).

- Mode 0: Full record sort/select using an existing parameter file.
- Mode 1: Merges two sorted files into one sorted file using an existing parameter file to specify the sort keys.
- Mode 2: Full record sort/select using the parameters specified in the command string and does not write the parameter file to the disk.
- Mode 3: Full record sort/select using the parameters specified in the command string and writes the parameter file to the disk.
- Mode 4: Sort/select using a parameter file. Output file contains only the Data Record pointer and sort keys.
- Mode 5: Sort/select using a command string. The Output file contains only the Data Record pointer and sort keys.
- Mode 6: Sort/select using a parameter file. Output file contains only the 2-byte Data Record pointer.
- Mode 7: Sort/select using a command string. Output file contains only the 2-byte Data Record pointer.
- Mode 8: Creates a parameter file on the disk from the command string. No sort/select is done.

AUTOSORT/86 can be used as a stand-alone sort routine, or it can be called from the MS-BASIC Interpreter, the MS-BASIC Compiler, MS-Pascal, MS-FORTRAN or MS-COBOL. Multiple users can sort simultaneously as long as each user's program specifies a different user number in the command string. AUTOSORT/86 creates unique temporary files for each user, eliminating the possibility of conflicting temporary file names.

Record lengths can be 5000 bytes or more if the specified sort buffer is at least 40K. File size is determined by your operating system and the available disk work file space. The logical record counter overflows at 65536.

In a 128K system, a typical default buffer size is about 60K if the sort module is loaded low enough to permit a 60K buffer. AUTOSORT/86 creates up to 30 work files, with each work file a little smaller than the sort buffer size. When you still have more to sort after these 30 work files are filled, AUTOSORT/86 temporarily merges the work files into a single file and creates up to 29 more work files. This process continues until either the input file or work space is depleted. The worst-case requirement for work space is about twice the size of the input file, if temporary merges are required. The temporary merge occurs when there are less than 30 work files and the specified buffer size is too small to allow 30 work files to be merged. A maximum of 10 sort keys are permitted, either ascending or descending (independently) on each key.

AUTOSORT/86 deletes or retains records by comparing up to four independent select keys with any fields in the record (fixed, variable, string, or numeric); AUTOSORT/86 checks whether the select key is "less than", "equal to" or "greater than" the selected fields. A select OR function (if activated) lets records be retained if any one of several select keys matches the fields.

An alpha option translates all lowercase alpha characters to uppercase for sorting. This causes "a" and "A" to sort as the same character.

The disk change capability is directed by the sort parameters. This capability lets you change the work file disk or the output disk during the sort process. (A screen prompt appears when you must change a diskette.) If the output file is given the same name as the input file, the input file is deleted after

the work files are created. This saves disk space; however, a power loss or malfunction during the final merge can cause you to lose the data file. For this reason, you should always back up your data files.

GETTING STARTED 1.2

Before using AUTOSORT/86, you should read the manual and then run the stand-alone test programs in Appendix B. Then, the chapter on high-level languages explains how to call the sort as a subroutine.

2

AUTOSORT/86 COMMAND STRINGS

Mode and sort parameters are passed to the AUTOSORT/86 module in a single command string (CMND$) with each parameter separated by a backslash (\).

For simplicity, the command string (CMND$) is shown as two strings separated by a backslash. The "\" indicates a backslash is necessary between the adjacent parameters when the two strings are concatenated.

The AUTOSORT/86 command string format is:

```
CMND$ = AMODE$ + "\" + PARM$
```

where:

AMODE$ contains parameters that must be defined during run time, and are associated with the sort mode.

PARM$ represents the sort parameters defined during run time or specified as parameters in a parameter file that has been previously stored on disk.

The sort parameter string (PARM$) begins with the input file name. String elements are in the same order as they were entered into the sort parameter file created by the parameter file generator program, PFG86M.COM. (See the section, "Parameter String Description," for a description of these parameters.) To call AUTOSORT/86, you create the command string (CMND$) for the desired mode. Then, call the AUTOSORT/86 module according to the procedure specified for your higher-level language.

2.1 AUTOSORT/86 MODES

MODE PARAMETERS

The first parameter in the mode parameter string (AMODE$) defines the mode of operation of the sort module. (An explanation of the various modes follows later in this section.)

The second parameter is the user number — a single character inserted into all temporary files to make them unique. Any normal file name character is permitted. The user number lets several users sort simultaneously on the same disk.

The third parameter sets the drive where the sort buffer memory area is saved during the sort process. Drives A through Z can be specified. (Use "0" for the default drive.)

The fourth parameter sets the sort buffer size (in bytes). You should make this size as large as possible; a zero defaults to the maximum size (approximately 60K for a 128K system). The sort buffer is above the sort module in memory. Its contents are written to disk before the sort and restored to disk after the sort. If you have limited disk space, you might want to set the sort buffer size to some smaller value. If you have very large records, be sure to leave enough room for the parallel merge. Work files, if needed, require additional disk space.

Some modes may have additional parameters in the mode parameter portion (AMODE$) of the command string (CMND$).

Sort parameters can be defined dynamically during run time as a sort parameter string (PARM$); or they can be defined external to the program by using the PFG86M.COM program. Also, certain modes let you define the sort parameters dynamically and write them to the disk as a sort parameter file for later use.

The parameter file generator (PFG86M.COM) creates the parameter files for modes 0, 1, 4, and 6. It is also useful when creating the parameter string

(PARM$) for modes 2, 3, 5, 7, and 8. PFG86M.COM requests the identical parameters (starting with the input file name) in the same order as are needed to form the sort parameter string (PARM$). Remember that the parameters in PARM$ must be separated by backslashes. To create a parameter file using PFG86M.COM, enter:

```
PFG86M
```

and then answer the questions. (See Chapter 3, "Sort Parameters," for a detailed description of PFG86M.)

MODE DESCRIPTION

Mode 0

The command string format is:

```
AMODE$ = "0\Un\Dm\Bufsize"
PARM$ = "D:PFNAME"
CMND$ = AMODE$ + "\" + PARM$
```

where:

0 is the mode.

Un is the user number. For a single user, enter 1.

Dm is the temporary memory storage drive. Drives A through Z can be specified. (Use zero for default drive.)

Bufsize is the number of bytes in the buffer. 0 defaults to maximum.

D is the drive containing PFNAME (optional).

PFNAME is the name of the parameter file. The extension must be .SRT.

In mode 0, a full record sort/select is performed using sort parameters from a parameter file previously created by PFG86M.COM or by using modes 3 or 8.

See the section, "Parameter String Description."

Mode 1

The command string format is:

```
AMODE$ = "1\Un\Dm\Bufsize"
PARM$ = "D:PFNAME\D:INPUT1\D:INPUT2\D:OUTPUT"
CMND$ = AMODE$ + "\" PARM$
```


where:

1 is the mode.

Un is the user number. Use 1 for single user.

Dm is the temporary memory storage drive. (0 is the default.)

Bufsize is the buffer size. 0 defaults to the maximum.

D is the appropriate drives (optional).

PFNAME is the parameter file name with extension .SRT.

INPUT1 is the name of the first input file.

INPUT2 is the name of the second input file.

OUTPUT is the name of the output file.

This mode merges two sorted files into one sorted file. A previously created parameter file must exist on disk to provide the sort/select key information. If the parameter file is set to skip records, the records in the first input file (INPUT1) are skipped.

The input and output file names specified in the parameter file on the disk are ignored since they are already specified in the command string. For a properly ordered merge, the sort keys must specify the same order as the sorted input files; otherwise, the sorted results are unpredictable.

If select keys are specified in the parameter file, the appropriate records are selected.

Mode 2

The command string format is:

```
AMODE$ = "2\Un\Dm\Bufsize"
PARM$ = The sort parameters
CMND$ = AMODE$ + "\" + PARM$
```

where:

2 is the sort/select mode.

Un is the user number. Use 1 for single user.

Dm is the temporary memory storage drive. (0 is the default.)

Bufsize is the buffer size. 0 defaults to maximum.

PARM$ is the sort parameter string beginning with the input file name.

This mode does a full record sort/select using the parameters specified in the command string. No parameter file is written to disk.

See the section, "Parameter String Description."

Mode 3

The command string format is:

```
AMODE$ = "3\Un\Dm\Bufsize\D:PFNAME"
PARM$ = The sort parameters
CMND$ = AMODE$ + "\" + PARM$
```

where:

3 is the mode.

Un is the user number. Use 1 for single user.

Dm is the temporary memory storage drive. (0 is the default.)

Bufsize is the buffer size. 0 defaults to maximum.

D is the drive on which to put the parameter file.

PFNAME is the name to give the parameter file.

PARM$ is the sort parameter string beginning with the input file name.

This mode writes a parameter file to the disk for later use and then does a full record sort using the parameters specified. The name you give the parameter file is inserted as parameter number 5 in AMODE$.

See the section, "Parameter String Description."

Mode 4

The command string format is:

```
AMODE$ = "4\Un\Dm\Bufsize"
PARM$ = "D:PFNAME"
CMND$ = AMODE$ + "\" + PARM$
```

where:

4 is the mode.

Un is the user number. Use 1 for single user.

Dm is the temporary memory storage drive. (0 is the default.)

Bufsize is the buffer size. 0 defaults to the maximum.

D is the drive containing the sort parameter file to use.

PFNAME is the sort parameter file name. (The extension is assumed to be .SRT.)

This mode uses a previously created sort parameter file. It creates an output file with records containing only the data record number (2 bytes) and the sorted fields in the order specified:

```
output record = [rec. no.][field #1]..[field #n]
```

The output record length equals 2 plus the sum of the sort key lengths.

Mode 5

The command string format is:

```
AMODE$ = "5\Un\Dm\Bufsize"
PARM$ = The sort parameters
CMND$ = AMODE$ + "\" + PARM$
```

where:

5 is the mode.

Un is the user number. Use 1 for a single user.

Dm is the temporary memory storage drive. (0 is the default.)

Bufsize is the buffer size. 0 defaults to the maximum.

PARM$ is the sort parameter string beginning with the input file name.

This mode uses a sort parameter string (PARM$). Mode 5 creates an output file which contains only the data record pointer (2 bytes) and the sort keys in the order specified:

```
output record = [rec. no.][field #1]..[field #n]
```

Mode 6

The command string format is:

```
output record = [rec. no. (2 bytes)]
AMODE$ = "6\Un\Dm\Bufsize"
PARM$ = "D:PFNAME"
CMND$ = AMODE$ + "\" + PARM$
```

where:

6 is the mode.

Un is the user number. Use 1 for a single user.

Dm is the temporary memory storage drive. (0 is the default.)

Bufsize is the buffer size. 0 defaults to the maximum.

D is the drive containing the sort parameter file to use.

PFNAME is the sort parameter file name. (The file extension is assumed to be .SRT.)

This mode uses a sort parameter file to do the sort/select. Then, it produces an output file consisting of only the input data record pointers (2 bytes per data record).

Mode 7

The command string format is:

```
output record = [rec. no. (2 bytes)]
AMODE$ = "7\Un\Dm\Bufsize"
PARM$ = The sort parameters
CMND$ = AMODE$ + "\" + PARM$
```

where:

7 is the mode.

Un is the user number. Use 1 for a single user.

Dm is the temporary memory storage drive. (0 is the default.)

Bufsize is the buffer size. 0 defaults to the maximum.

PARM$ is the sort parameters beginning with the input file name.

This mode uses a sort parameter string (PARM$) to do the sort/select, and produces an output file containing only the input data record pointers (two bytes per record).

Mode 8

The command string format is:

```
AMODE$ = "8\Un\Dm\Bufsize\D:PFNAME"
PARM$ = The sort parameters
CMND$ = AMODE$ + "\" + PARM$
```


where:

8 is the mode.

Un is the user number. Use 1 for a single user.

Dm is the temporary memory storage drive. (0 is the default.)

Bufsize is the buffer size. 0 defaults to the maximum.

D is the drive on which to put the sort parameter file.

PFNAME is the name to give the parameter file. (The extension is assumed to be .SRT.)

PARM$ is the sort parameter string beginning with the input file name.

NOTE: Although Un, Dm and Bufsize are not used, each must still be set to an acceptable value.

This mode only creates a sort parameter file on a disk. No sort/select is done.

PARAMETER STRING DESCRIPTION

The following table shows the order of the parameters in the parameter string (PARM$) for modes 2, 3, 5, 7 and 8. This is the same order in which the parameters are requested when building a parameter file on disk.

Exhibit 2a: Order of Parameters in Parameter String

PARAMETER	INPUT	MAX. CHARS.
INPUT FILE	d:filename.ext	14
OUTPUT FILE	d:filename.ext	14
NUMBER RECS TO SKIP	nnn	3
LOGICAL RECORD LENGTH	nnnn	4
CHANGE WORK FILE DISK	Y/N	1
CHANGE OUTPUT FILE DISK	Y/N	1
WORK FILE DRIVE	A-Z or 0	1
Sort Key 1		
FIELD STARTING POSITION	0 or nnnn (0 to stop key input)	4
FIELD LENGTH	nnn	3
ASCEND OR DESCEND FLAG	A or D	1
Sort Key 1		
ALPHA/HEX/INTEGER/SGL/DBL	A/H/I/S/D (Repeats the last 4 for next key. Enter 0 to stop key input, except when all 10 keys are used; then go to the next input.)	1
ACTIVATE SEL "OR" FUNCTION	Y/N	1
Select Key 1		
DELETE OR RETAIN	0 or D or R (0 to stop key input)	1
FIELD STARTING POSITION	nnnn	4
FIELD LENGTH	nnn	3
ALPHA/HEX/INTEGER/SGL/DBL	A/H/I/S/D	1
LESS-EQUAL-GREATER	L or E or G	
DELETE-RETAIN KEY (Sum of all keys must not exceed 135.)	(Repeats the last 6 for next key. Enter 0 to stop key input if all 4 keys are not used; otherwise input is complete.)	135

2.2 SAMPLE COMMAND STRING

The following command string (CMND$) sorts the input file TEST.DAT. TEST.DAT contains 128 byte records in ascending order on the field starting at byte 4; each byte record has a length of 8 bytes in ascending order. The

command string deletes all records that have "DELETED" at byte position 4. Mode 2 is used for a full record sort, so the parameter file is not written to the disk. The output file is named SORTED.

See Chapter 3, "Sort Parameters," for the definition of each parameter.

```
AMODE$ = "2\1\A\0" (mode parameter string)
PARM1$ = "TEST.DAT\SORTED\0\128\N\N\B"
PARM2$ = "4\8\A\A\0" (sort keys)
PARM3$ = "N\D\4\7\A\E\DELETED\0" (sel keys)
PARM$ = PARM1$ + "\" + PARM2$ + "\" + PARM3$
   (sort parameter string)
CMND$ = AMODE$ + "\" + PARM$ (final command string)
```

Exhibit 2b explains the parameters contained in the sample command string.

Exhibit 2b: Sample Command String Parameter Description

PARAMETERS	DESCRIPTION
(AMODE$)	
2	Mode
1	User number
A	Memory storage drive is "A"
0	Default to maximum buffer size
(PARM1$)	
TEST.DAT	Input file (default drive)
SORTED	Output file (default drive)
0	Skip 0 header records
128	Record length = 128 bytes
N	Do not change work diskette
N	Do not change output diskette
B	Work file drive is "B"
(PARM2$) SORT KEYS	
4	Sort key starts at byte 4
8	Sort key is 8 bytes long
A	Ascending order
A	Alpha (make upper/lowercase same)
0	End of sort keys (repeat the last 4 if more sort keys)
(PARM3$)	
N	Do not use select "OR" function
SELECT KEYS	
D	Delete option (0 if no select keys)
4	Select key starts at byte 4
7	Select key is 7 bytes long
A	Alpha (upper/lowercase same)
E	Select on "EQUAL"
DELETE	Select key value = "DELETED"
0	No more select keys (repeat last 6 if more select keys)

3

SORT PARAMETERS

PARAMETER FILE OVERVIEW 3.1

The sort parameter files contain three types of parameters: file, key and control.

The file parameters specify the drive, file name, and file type of the output and input files. The allowable drives are A through Z.

The sort and select key parameters are specified by the byte position in the record and the number of bytes in the key. Up to ten ascending or descending keys are allowed. The keys are sorted on the hex value of each byte if the Hex (H) control character is selected. If the Alpha (A) control character is selected, upper- and lowercase characters sort as the same value. The Integer (I), Single Precision (S), and Double Precision (D) control characters expect to find fields that are compatible with Microsoft BASIC integer, single precision, and double precision fields. Negative numbers are considered smaller than positive; a larger negative number sorts smaller than a lesser negative number.

If the disk change options are used, a screen prompt asks you to insert either the work file diskette or the output diskette, and tells you the drive in which to place it.

SORT PARAMETER DEFINITION 3.2

Generally, parameter files are created dynamically (if needed) during run-time; however, they can also be created by the PFG86M.COM program.

When the command string (CMND$) is used to specify the sort parameters, the actual sort parameters begin with the input file name contained in

the command string. The first four parameters specify the mode, the user number, the memory storage drive, the sort buffer size, and possibly additional parameters.

The parameter file generator program is executed by entering "PFG86M" after the system prompt. Then the following prompts appear. (The action you need to take is described following each prompt.)

ENTER PARAMETER FILE TO CREATE (D:FILENAME), DO NOT ENTER FILE EXTENSION (ASSUMED .SRT).

Enter the desired name.

ENTER "INPUT" FILE (D:FILENAME.EXT)

Enter the input file.

ENTER "OUTPUT" FILE (D:FILENAME.EXT)

Enter the output file.

ENTER NUMBER OF RECORDS TO SKIP

Enter the number of header records to skip.

ENTER LOGICAL RECORD LENGTH

Enter the record length.

CHANGE WORK FILE DISKETTE?

Enter Y or N.

CHANGE OUTPUT FILE DISKETTE?

Enter Y or N.

ENTER WORK FILE DRIVE

Enter drive names from A to Z. Enter 0 to use default drive.

ENTER KEY #1 STARTING POSITION (0 TO STOP)

Enter starting byte (0 if no more keys). Enter 0 for the first key if there are no select keys.

ENTER KEY #1 LENGTH

Enter the length (1 to 255).

ENTER ASCEND/DESCEND FLAG

Enter A or D.

ENTER ALPHA/HEX/INTEGER/SGL/DBL FLAG

Enter A to sort upper/lowercase as the same value. Enter H to sort on the hex value. Enter I to sort integer fields. Enter S to sort single precision fields. Enter D to sort double precision fields.

The sort keys repeat until a 0 is entered for the FIELD #/STARTING POSITION, or until 10 keys are used.

ACTIVATE THE SELECT KEY "OR" FUNCTION?

Enter N to use multiple select keys as an AND function and Y to use multiple keys as an OR function. For single select keys use N. If there are no select keys, use either.

DELETE OR RETAIN RECORD?

Enter D to delete or R to retain the records whose fields match the select key. Enter 0 if no more select keys. Entering 0 for the first select key results in no select function being applied.

ENTER SELECT FIELD STARTING POSITION

Enter the starting byte number.

ENTER SELECT FIELD LENGTH

Enter the length of the select field in the data record.

ALPHA/HEX/INTEGER/SGL/DBL FLAG

Enter A to treat upper/lowercase same. Enter H to use actual hex value. Enter I to sort integer fields. Enter S to sort single precision fields. Enter D to sort double precision fields

DELETE ON "LT, EQ, GT" THE SELECT KEY?

Enter L for less than. Enter E for equal. Enter G for greater than.

ENTER D/R (DELETE/RETAIN) KEY
D/R KEY:

Enter the actual select key value.

The select keys repeat until a 0 is entered for the DELETE or RETAIN parameter, or until 4 select keys are used. The parameter file is created on the specified drive.

4

RECORD SELECT FEATURES

AUTOSORT/86 can create an output file that contains only selected records from the input file. The select can occur during the sort process, or it may occur independent of a sort (if no sort keys are specified).

Four independent select keys can be specified. This lets you delete or retain records if the specified fields are less than, equal to, or greater than the DELETE/RETAIN select keys.

If Y is entered for the select key OR function and more than one select key is used, records that match any of the keys can be deleted or retained.

If N is entered for the select key OR function and more than one select key is used, all fields must match for the record to be retained (AND function).

The select keys are totally independent of the sort keys. Selects can be performed on the same keys or on different keys than the sorts.

For additional flexibility, the characters <, =, > can be inserted at any place in the select key, with the following results:

- When a < is encountered in the select key, the select key character is considered "less than" the corresponding character in the data field.
- When a > is encountered in the select key, the select key character is considered "greater than" the corresponding character in the data field.
- When an = is encountered in the select key, the select key character is considered "equal to" the corresponding character in the data field.

If the select key is longer than the data field, it is truncated to the length of the data field. If the select key is shorter than the data field, the data field is compared only for the length of the select key. The select key is considered a "match" if all characters in the select key match the data field for the length of the select key.

The select characters can only be used with string fields. They are not permitted with integer, single precision, or double precision fields.

5

USING AUTOSORT/86 WITH PROGRAMMING LANGUAGES

Read Appendix B and run the stand-alone tests before continuing with this chapter.

AUTOSORT/86 AND MS-BASIC 5.1

To call AUTOSORT/86 from an MS-BASIC program, the AS86M.COM module must be resident at a known segment in memory. To load the AS86M.COM module, type:

AS86M

after the A> prompt. The BASIC Interpreter and programs are loaded above the AS86M.COM module by MS-DOS. When the AS86M.COM module is loaded, the load segment is displayed as:

ASSEG = &Hnnnn

where:

nnnn is the segment where the AS86M.COM module is loaded.

This statement must be placed at the beginning of all MS-BASIC programs that call the AUTOSORT/86 program (AS86M.COM). Also, you need the following subroutine in your MS-BASIC program:

```
DEF SEG = ASSEG
ASORT = &H3
CALL ASORT(CMND$,SORTERR%,RECCNT%)
DEF SEG
RECCNT = RECCNT%
IF RECCNT<0 THEN RECCNT = RECCNT + 65536!
RETURN
```

To call a sort, simply declare a command string (CMND$) and GOSUB to the preceding subroutine. After a Return, you should test the returned variable SORTERR%. If it is not zero, there was an error. The variable RECCNT is equal to the number of logical records in the output file.

Follow these steps to run the test programs:

1. Copy the following programs from your AUTOSORT/86 disk to a new, formatted disk:
   - ► AS86M.COM
   - ► TEST.DAT
   - ► TEST.BAS
   - ► PRINT.BAS
2. Copy your MSBASIC.COM file to the new diskette.
3. Place the new diskette in drive A and type AS86M so that the AS86M.COM module is loaded and fixed in memory.
4. Record the ASSEG statement displayed when AS86M.COM was loaded.
5. Edit the TEST.BAS and PRINT.BAS programs so that the ASSEG statement is replaced with the one displayed for your system configuration.
6. Run the test program by entering:

   MSBASIC TEST

   after the A> prompt.

   A mode 2 (full record) sort is done on 500 25-byte records. See Appendix B for a description of the parameters used in this sort.
7. Run the PRINT.BAS program to display the TEST.DAT (input) file or the SORTED (output) file.
8. List the TEST.BAS program and use it as an example of how this particular command (CMND$) was formed.

You can change the mode from 2 to 5 or 7 to do different kinds of sorts. If the mode is changed to 5, the output records consist of a 2-byte integer field (the data record pointer) and an 8-byte string field (10-byte record length). If you set the mode to 7, the output records consist of a 2-byte integer field for a total record length of 2 bytes. The PRINT.BAS program must be modified to print the output file for mode 5 or 7 because it expects a 25-byte record and the same fields as in the input file.

AUTOSORT/86 AND THE MS-BASIC COMPILER 5.2

Follow these steps to combine AUTOSORT/86 with a compiled MS-BASIC program:

1. Put this subroutine in the MS-BASIC program:

```
CALL SORTMC(CMND$,SORTERR%,RECCNT%)
RECCNT = RECCNT%
IF RECCNT<0 THEN RECCNT = RECCNT + 65536!
RETURN
```

2. To call a sort, you simply define a command string (CMND$) and GOSUB to the preceding subroutine. The command string provides the sort attributes.

3. Compile your MS-BASIC program using the BASCOM program. (See your programmer's guide for the BASIC Compiler.) You now have an .OBJ file for your program.

4. Using LINK.EXE, link your .OBJ file with the AS86M.OBJ file to form the executable program. (See the MS-LINK Section of the *Programmer's Tool Kit, Volume II.*)

   NOTE: When linking the AS86M.OBJ module to your programs, the AS86M.OBJ module MUST NOT be loaded first.

5. Execute your program.

Follow these steps to run the MS-BASIC Compiler test program:

1. Copy the following programs from your AUTOSORT/86 disk to a new, formatted disk:

   - AS86M.OBJ The sort module
   - TEST.DAT Test data file
   - TESTBC.BAS BASIC test program
   - TESTBC.OBJ Object file for test program
   - PRINT.BAS Program for printing files
   - PRINT.OBJ Object file for PRINT.BAS

2. Copy your BASRUN.EXE file to the new diskette.

3. Link the following object files to form the indicated executable files.

Executable File	Object Files
TESTBC.EXE	TESTBC.OBJ + AS86M.OBJ
PRINT.EXE	PRINT.OBJ

4. Run the test program by entering:

   TESTBC

   after the system prompt.

   A mode 2 (full record) sort is done on 500 25-byte records. The sort field starts at byte 4 and is 8 bytes long. See Appendix B for a description of the parameters used in this sort.

5. Run the PRINT.EXE program to display the TEST.DAT (input) file or SORTED (output) file.

6. List the TESTBC.BAS program and use it as an example of how this particular command (CMND$) was formed.

You can change the mode to 5 or 7 for different kinds of sorts. If the mode is 5, the output records consist of a 2-byte integer field (the data record pointer) and an 8-byte string field (10-byte record length). If the mode is 7, the output records consist only of a 2-byte integer field for a total record length of 2 bytes. The PRINT.BAS program must be modified to print the output file for mode 5 or 7 because the program expects a 25-byte record and the same fields as in the input file.

AUTOSORT/86 AND MS-PASCAL 5.3

The following sample Pascal program does a mode 0 (zero) sort (full record sort using a parameter file from the disk), saves the buffer area on the default drive, and uses the maximum buffer space.

```
program pasprog(output);
  type
    cmdstring = lstring(255);
  var
    ercode:         word;
    reccnt:         word;
    command:        cmdstring;
  function sortps(vars cmd: lstring;
           vars err: word) : word; external;

  PROCEDURE errtn(code:word);
  BEGIN
    writeln(output,"Fatal error number ",code)
  END;

  PROCEDURE rcnt(nrecs:word);
  BEGIN
    writeln(output,"Number of records is ",nrecs)
  END;

  begin
    command:= "0\1\0\20000\TEST";
    reccnt:= sortps(command,ercode);
    rcnt(reccnt);
    if ercode <> 0 then errtn(ercode)
  end.
```

To call a sort, simply define a command string and execute the external procedure (sortps). Remember to use an lstring for the command string and segmented variables (vars) where indicated.

To link AUTOSORT/86 to the Pascal program, you must first compile your Pascal program using the directions given in your *MS-Pascal Reference Manual*. This will produce an object version of your program (YOURPROG.OBJ).

Then, you must link your program with the AS86M.OBJ module using the instructions given for linking modules using LINK.EXE. (See the MS-LINK Section of the *Programmer's Tool Kit, Volume II.*)

NOTE: The AS86M.OBJ module must not be the first module specified.

When requested by the linker, enter the object modules as shown:

```
Object Modules:  YOURPROG,AS86M
```


AUTOSORT/86 assigns buffer space above its location in memory. The contents of this buffer area are written to the specified diskette during the sort procedure and are replaced before returning to the Pascal program. This lets you sort large files (several megabytes) from within the Pascal program in a reasonable time without the overhead of dedicated buffer space.

A sample Pascal program (PASTEST.PAS and PASTEST.OBJ) is provided on the distribution diskette. (The sample Pascal program on the disk is not the same as that in the last example.) Follow these steps to run the sample program:

1. Copy the following programs to a new, formatted diskette:

   - PASTEST.PAS: The sample source file.
   - PASTEST.OBJ: The sample object file.
   - TEST.DAT: The test data file.

- AS86M.OBJ: The AUTOSORT/86 module.
- PRINT.BAS: A BASIC file used to display the TEST.DAT and SORTED files.

2. Link the PASTEST.OBJ and the AS86.OBJ file to create PASTEXT.EXE. (See your *MS-Pascal Reference Manual*.)

3. Run the PASTEST.EXE program.

   A mode 2 (full record) sort is done on 500 25-byte records. The sort field starts at byte 4 and is 8 bytes long. See Appendix B for a description of the parameters used in this sort.

4. If you have the MS-BASIC Interpreter, run the PRINT.BAS program to display the TEST.DAT (input) file or the SORTED (output) file. If you do not have the interpreter, use the PRINT.BAS program as a guide to constructing a Pascal program that displays the files.

5. List the PASTEST.PAS program and use it as a guide to incorporate AUTOSORT/86 into your Pascal program.

You can change the mode (the first parameter) in the command string to produce different types of output files. If the mode is 5, the output records consist of a 2-byte integer field, the data record pointer, and an 8-byte string field (for 10 byte total record length). If the mode is 7, the output records consist only of a 2-byte integer field (for a total record length of 2 bytes). The PRINT.BAS program must be modified to print the output file (SORTED) for mode 5 or 7 because the program expects a 25-byte record and the same fields as the input file (TEST.DAT).

5.4 AUTOSORT/86 AND MS-FORTRAN

The following sample FORTRAN program does a mode 0 sort (full record sort using a parameter file from the disk), saves the buffer area on the default drive, and uses the maximum buffer space.

```
C     TEST PROGRAM FOR AUTOSORT AND FORTRAN

      PROGRAM FORTEST
      CHARACTER *80 CMND
      INTERGER *2 RECCNT, ERCODE, SORTFO

      CMND = '0\1\0\0\TEST^'
      RECCNT = SORTFO(CMND, ERCODE)

      WRITE (*,100) ' ERCODE = ', ERCODE
      WRITE (*,200) ' RECCNT = ', RECCNT
100   FORMAT (1X,A8,I6)
200   FORMAT (1X,A8,I6)
      STOP
      END
```


The CMND character string must be terminated with an up-arrow. This is because no length byte is passed with a FORTRAN string. See the description of mode 0 in Chapter 2 for an explanation of this command string.

To link AUTOSORT/86 with your FORTRAN program, first compile your program to produce the Object file. Using LINK.EXE, link your program with AS86M.COM when requested by the linker:

```
Object Modules: YOURPROG + AS86M
```

Do not load the AS86M.OBJ module first.

AUTOSORT/86 assigns buffer space above its location in memory. The contents of this buffer area are written to the specified diskette during the sort procedure and are replaced before returning to the FORTRAN program. This lets you sort large files (several megabytes) from within the FORTRAN program in a reasonable time without the overhead of dedicated buffer space.

The following sample COBOL program does a mode 0 sort (full record sort using a parameter file from the disk), saves the buffer area on the default drive, and uses the maximum buffer space.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBTEST.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 CMND PIC X(80) VALUE IS "0\1\0\0\TEST^"
77 ERCODE PIC 99999 COMP-0 VALUE 0.
77 RECCNT PIC 99999 COMP-0 VALUE 0.
77 ERC     PIC 99999.
77 RCNT    PIC 99999.
PROCEDURE DIVISION.
MAIN.
   CALL "SORTCO" USING CMND, ERCODE, RECCNT
   MOVE ERCODE TO ERC.
   MOVE RECCNT TO RCNT.
   DISPLAY "ERCODE = " ERC.
   DISPLAY "RECCNT = " RCNT.
   STOP RUN
```


The CMND character string must be terminated with an up-arrow so that AUTOSORT/86 can find the length of the string. See the discussion of mode 0 in chapter 2 for an explanation of this command string.

To link AUTOSORT/86 with your COBOL program, first compile your program to produce the Object file. Using LINK.EXE, link your program with AS86M.OBJ when requested by the linker:

```
Object Modules: YOURPROG + AS86M
```

Do not load the AS86M.OBJ module first.

AUTOSORT/86 assigns buffer space above its location in memory. The contents of this buffer area are written to the specified diskette during the sort procedure, and are replaced before returning to the COBOL program. This lets you sort large files (several megabytes) from within the COBOL program in a reasonable time without the overhead of dedicated buffer space.

6

ERROR INDICATIONS

When an error is detected, the error number is returned to the calling program. If the returned error number is 0, no error occurred.

The following error numbers are generated by AUTOSORT/86. Each is followed by an explanation and suggestions for correcting the error condition.

1 READ PAST END OF FILE

This error should not occur. It probably indicates a system or diskette malfunction.

2 READ ERROR OR BAD FILE

Same as error 1.

3 FILE NOT PRESENT WHEN OPENED

A file was not present on a specified drive. Check the directory.

4 OUT OF DIRECTORY SPACE

Too many directory entries. Delete unnecessary files.

5 NOT USED

6 NOT USED

7 FILE NOT PRESENT WHEN CLOSED

Same as error 1.

8 INSUFFICIENT DISK SPACE

Delete unnecessary files.

9 SORT BUFFER SPACE TOO SMALL

Set larger buffer size in the command string.

11 NOT USED

13 SYNTAX ERROR IN THE COMMAND STRING

Check the command string carefully.

14 SELECT KEY LENGTH IS ZERO

Select key length cannot be zero.

15 SYNTAX ERROR IN THE COMMAND STRING

Check the command string carefully.

A

AUTOSORT/86 PUBLIC INTERFACES

Exhibit Aa shows the PUBLIC entry and access locations used by AUTOSORT/86:

Exhibit Aa: PUBLIC Entry and Access Locations

PUBLIC SYMBOL	NORMALLY USED BY
SORTMC	MS-BASIC Compiler
SORTPS	MS-Pascal
SORTFO	MS-FORTRAN
SORTCO	MS-COBOL

SORTMC ENTRY POINT A.1

On entering at SORTMC, three nonsegmented pointers must be pushed on the stack. These represent the addresses of the command string descriptor, sort error code, and record count, in that order. All pointers are assumed to be offsets to the Data Segment register (DS) at entry.

The command string descriptor must be 4 bytes. The first 2 bytes contain the length of the command string; the next 2 bytes contain the offset of the command string relative to the DS register at entry.

At exit from AUTOSORT/86, the three pointers (6 bytes) are removed from the stack and the returned parameters are placed in the specified addresses.

A.2 SORTPS ENTRY POINT

On entering at SORTPS, two segmented pointers (8 bytes) must be pushed on the stack. These represent the addresses of the command string descriptor and error code, in that order. The segment is pushed on the stack first, followed by the offset. The first byte of the command descriptor must contain the length, followed by the actual string of characters.

At exit from AUTOSORT/86, the two pointers (8 bytes) are removed from the stack and the error code is placed in the specified offset and segment. The returned parameter (RECCNT) is placed in the AX register as the returned function.

A.3 SORTFO ENTRY POINT

On entering at SORTFO, two segmented pointers must have been pushed on the stack. These represent the addresses of the command string descriptor and error code, in that order.

The command string segmented pointer must point to the first actual byte of the command string. The command string must be terminated with an up-arrow so that AUTOSORT/86 can determine the length.

At exit, the two segmented pointers (8 bytes) are removed from the stack and the error code is placed in the specified address. The returned parameter (RECCNT) is returned in the AX register.

A.4 SORTCO ENTRY POINT

On entry at SORTCO, three nonsegmented pointers must be pushed on the stack. These represent the addresses of the command string descriptor, error code, and RECCNT, in that order.

All pointers are assumed to be offsets to the Data Segment register (DS) at entry. The command string must be terminated with an up-arrow, so that AUTOSORT/86 can determine the length.

At exit, the parameters ERROR CODE and RECCNT are placed in the specified addresses.

B

STAND-ALONE SORTING

To do a stand-alone sort/select, the following files must be available on one of the drives:

- SORTM.COM: This program requests some information and then loads and calls the sort module.
- AS86M.COM: The sort module.
- A previously created sort parameter file.

Follow these steps to run a sample stand-alone sort:

1. Copy SORTM.COM, AS86M.COM, TEST.DAT and TEST.SRT from the distribution disk to a freshly formatted disk in drive A.

2. Execute the sort caller by entering "SORTM" after the system prompt. A series of prompts appears.

3. Enter:

   A:

   when the sort program drive is requested.

4. Enter:

   0

   when the sort mode is requested.

5. Enter:

   1

   when the user number is requested.

6. When the parameter file name is requested, enter:

TEST

The sort begins and will complete in 10 to 15 seconds.

The TEST.DAT file consists of 500 records of 25 bytes each. Exhibit Ba shows the fields contained in TEST.DAT.

Exhibit Ba: TEST.DAT Field Description

START	LENGTH	DESCRIPTION
1	1	Constant "R"
2	2	2-character string
4	8	8-character string
12	2	Integer
14	4	Single precision
18	8	Double precision

The parameter file (TEST.SRT) specifies a sort on the string field, starting at byte position 4. The field is 8 bytes long. Mode 0 is selected for a full record sort. The output file is named SORTED.

IMPORTANT NOTE: Since the record is less than 128 bytes long and the file was created using Microsoft's MS-BASIC, you may have enough null bytes at the end of the file for them to appear as additional null records. In an ascending sort, these null records migrate to the beginning of the file. To avoid this problem, a single byte field with a constant "R" is placed at the beginning of each record. Then, a select key that retains only those fields having an "R" at position 1 is defined in the parameter file.

If you have MS-BASIC, run the PRINT.BAS program to display the TEST.DAT and SORTED files. Here is how the parameter file TEST.SRT is created using PFG86M.COM.

First, "PFG86M" is entered after the system prompt. Then, the following answers are given in response to prompts:

```
PARAMETER FILE NAME:                    TEST
INPUT FILE NAME:                        TEST.DAT
OUTPUT FILE NAME:                       SORTED
NO. OF RECS TO SKIP:                    0
LOGICAL REC LENGTH:                     25
CHANGE WORK DISK?                       N
CHANGE OUTPUT DISK?                     N
WORK FILE DRIVE:                        0
KEY #1 START POSITION:                  4
KEY #1 LENGTH:                          8
KEY #1 ASCEND/DESCEND                   A
KEY #1 ALPHA/HEX/INTEGER/SGL/DBL:       A

KEY #2 START POSITION:                  0

SEL KEY "OR" FUNCTION:                  N

SELECT KEY #1

DELETE OR RETAIN:                       R
SEL FIELD START:                        1
SEL FIELD LENGTH:                       1

ALPHA/HEX/INTEGER/SGL/DBL:              A
LT,GT,EQUAL:                            E
ACTUAL SELECT KEY:                      R

SELECT KEY #2

DELETE OR RETAIN:                       0
```

When all of these prompts are answered, AUTOSORT/86 creates parameter file TEST.SRT.

PMATE-86

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

First VICTOR printing February, 1983.

ISBN 0-88182-009-1 Printed in U.S.A.

CONTENTS

CHAPTERS

1

INTRODUCTION

TEXT EDITING, WORD PROCESSING, OUTPUT PROCESSING 1.1

A "text editor" is a computer program that helps people create and modify text. Traditionally, text editors have been oriented toward the needs of programmers; they move, copy, modify or eliminate portions of computer programs. Very quickly, it became obvious that these capabilities would be useful in creating and editing documents written in plain English. So, word processing programs were developed that let people enter text into a computer's memory and then edit that text as needed.

Text editors and word processors each have advantages in the manipulation of text. Originally, a text editor program was meant for use on a teletype, or some other type of "slow" terminal. Because of this, it takes a lot of time for a text editor to print out a version of a document. You would not want to print a new copy of the text each time you made a change, so you wind up working "blind." Sections of text are typed out only when you request them. To make up for this inconvenience, text editors usually have powerful text editing commands that will let you totally rearrange all your text with a few keystrokes.

On the other hand, word processing programs usually display the current appearance of your text on a video screen. As you change the text, the display changes instantly. The penalty you pay for seeing your work is that a word processor usually allows you to make only simple changes in your text.

A "text output processor" tries to make a text editor compatible with the real world of pages, paragraphs, underlines, and so on. You use a text editor to enter text, including a set of control words that indicate whether it's time to start a new page, indent a margin 20 spaces, or perform other text manipulations. After you finish your document, you run the output of the text editor through a text output processor, which types it out on your printer, nicely formatted. Text output processors give you a great deal of control

over the form of your final document; however, they have the disadvantage that the required input — the part created and updated by your text editor — looks very little like the final printed document. PMATE-86 combines some of the best features of text editors, word processors, and text output processors.

1.2 HOW THIS MANUAL IS ORGANIZED

Writing a manual for PMATE-86 presents problems. Since PMATE-86 is such a versatile program, its features are useful to people with vastly differing computer experience. This manual is designed to help any user. It begins with a discussion of the more basic features of the PMATE-86 program, and works its way through the more sophisticated applications. Some ideas and commands are presented in several places, each time at a higher level.

- Chapter 2 is for people who have a minimum of experience with computers or word processors. It tells how PMATE-86 implements basic text editing functions.
- Chapter 3 explains more advanced concepts and commands.
- Chapter 4 describes the complete PMATE-86 command set. You can start here if you have had experience with other text editors and only need to know the correct command formats in order to get going.
- Chapter 5 discusses the use of macros. After you are well acquainted with PMATE-86, work through the examples in this chapter. They'll show you how to use macros to greatly expand PMATE-86's built-in capabilities.
- Appendix A is a summary of commands that you can refer to while using PMATE-86.
- Appendix B provides configuration information. Read this chapter to customize PMATE-86 for your personal requirements and preferences.

2

AN OVERVIEW OF PMATE-86

PMATE-86 AND THE OPERATING SYSTEM 2.1

PMATE-86 resides on disk as the command file PMATE.CMD or PMATE.COM. Call the command file by typing:

PMATE

In a few seconds, PMATE-86 appears on the screen in Command mode. Typing Alt-N puts PMATE-86 in Insert mode. You can now enter and edit text. If you call PMATE-86 by typing:

PMATE file

"file" is opened as your input and output file.

If you want to save your work on disk, you must define an "output file." First, return to Command mode with an Alt-X. Then enter XF followed by the name you want to give to the output file. Next, strike the ± key twice. This produces two escape characters (each displayed on the screen as a dollar sign [$]). A single escape character terminates or separates commands; two escape characters in a row cause the previous command string to be executed.

When you finish editing, use one of these:

XE End edit pass by writing entire text buffer to output file, and closing it.

XK End edit pass without writing anything on the disk.

Both commands erase your text buffer, but only XE saves those contents on disk. After these commands, you are still in PMATE-86. To return to the operating system, use:

XH Go to the operating system. This command gives an error message if there are files open. This prevents you from exiting before you save your work on disk. To end the error condition, you must enter an XK or XE.

PMATE-86 can be used to modify an already existing file. To do this, you must create an input file. Again use:

XFfile Edit "file" (this time, assume "file" already exists). The file is opened for input, and the text is read in.

You can now modify the text, and again finish up with an XE or an XK. An XK leaves the original file intact. An XE, however, updates the input file to include the changes you have made. A copy of the input file as it was before modification is kept under the same file name with the extension .BAK (the previous backup file is deleted).

2.2 BUFFERS, DISPLAY, AND CURSOR

PMATE-86 stores text in the computer memory as a continuous stream of characters placed into any of 11 edit buffers. If you insert a character into existing text, all following characters are moved to make room for the new character. If a character is deleted, all the following characters move to fill in the space opened by the deleted character. Text lines are separated by carriage returns.

A video display shows you the part of the edit buffer that you are working on. As you enter or modify text, the display immediately shows the results of your actions.

A cursor shows exactly where any modifications take place. (The character to which the cursor points is displayed in reverse video.) The cursor can be moved around the screen by using the appropriate keys on your keyboard. As the cursor moves through the text, the screen scrolls up or down (vertical scrolling). If a line is too long to fit on the screen, the whole text display shifts to keep the cursor from moving off the right edge of the screen (horizontal scrolling). Lines are up to 250 characters long.

MANIPULATING TEXT 2.3

There are several ways to enter or modify text. The simplest way is to use Overtype mode or Insert mode. In these modes, any characters you type appear on the screen at the cursor location. In Insert mode, characters at or beyond the cursor location are shifted to the right to make room for the new ones. If you make a mistake, the Backspace or Delete keys are used to eliminate the incorrect characters. In Overtype mode, any character you type replaces the one beneath the cursor. (However, neither Returns nor Tabs are overwritten.) The bottom line of the command display tells you whether you are in Overtype or Insert mode.

"Instant commands" are also used in the text editing process. These are key sequences that do not appear in the text; they let you move the cursor, delete characters or lines, shift from upper- to lowercase letters, or do other editing operations. Instant commands are useful for entering text, and making minor changes in existing text.

For more complex text editing, you can use command strings. For instance, you could use a command string to tell PMATE-86: Find the third occurrence of "George" and change it to "Harry". Then, delete all characters until you find an F. Then, insert the numbers from 240 to 1000 in base 5, one per line. Finally, tell me how much (3*46/(5 + (3*7))) is. Of course, you wouldn't use exactly that language, but such a command string is easily constructed.

PMATE-86 executes command strings while in Command mode (the mode that PMATE-86 enters immediately after being loaded). Command mode is indicated by the absence of an Overtype or Insert mode message in the command line (the last one in the command display located at the top of the screen).

In Command mode, your keystrokes don't immediately affect the text; instead, they are entered into the command buffer and appear on the command line. (An underline cursor indicates where the next character will be entered.) A single command is usually one or two characters, but commands can be strung together into longer command strings. As soon as a command is executed, the display shows the updated edit buffer.

2.4 LINE FORMATTING

A line is a string of characters that ends with a carriage return. When you enter a Return, the cursor moves down to the beginning of the next line. Returns always indicate the end of a line. They should be used whenever characters must appear on a new line.

PMATE-86 also has an automatic line formatting facility for entering and editing text. When using this facility, text automatically "wraps around" as you enter: a new line is started each time a line reaches a specified length. Words are not broken up, however — the complete word is moved down to the next line. This line formatting is preserved, even as portions of the text are edited. When operating in this mode, Returns are normally entered only at the end of a paragraph.

2.5 INSTANT COMMANDS

Instant commands are keystrokes that don't appear in the command or text buffers. Instead, they have an immediate effect on your work. Instant commands are used all the time; a complete listing and description of Instant command actions follows.

MODE SWITCHING

These Instant commands set the mode of the editor.

^X	Go into Command mode.
^N	Go into Insert mode.
^V	Go into Overtype mode.

CURSOR MOTION

These Instant commands move the cursor throughout the text buffer. As the cursor moves, the display updates in order to keep itself centered on the cursor. The cursor is never allowed to move outside of the text buffer.

^A	Move the cursor to the beginning of the text buffer. If the cursor is already at the beginning of the buffer, it moves to the end.
← or ^G	Move the cursor to the left.
→	Move the cursor to the right.
↓ or ^B	Move the cursor down one line.
↑ or ^Y	Move the cursor up one line.
Shift-scroll or ^U	Move the cursor up six lines.
Scroll or ^J	Move the cursor down six lines.
^P	Move the cursor to the beginning of the next word.
^O	Move the cursor to the beginning of the word. If it is already there, the cursor moves to the beginning of the preceding word.
^F^M	Move the cursor to the beginning of the line. (^M is a Return.)

SCROLLING

These commands scroll the screen, while leaving the cursor at the same character.

^F^G	Scroll left one column.
^F^H	Scroll right one column.
^F^Y	Scroll up one line.
^F^B	Scroll down one line.

DELETION

These commands let you delete characters from the text. The remaining characters move down to fill in the space vacated by the deleted characters.

^D	Delete the character under the cursor.
^K	Delete characters from the cursor position to end of line.
^W	Delete the next word, starting at the cursor.
^Q	Delete the word preceding the cursor.
Backspace	Delete the character just put into the buffer. In Command mode, this deletes the character just entered into the command. When in Insert mode, this deletes the character just before the cursor. (^D deletes the character at the cursor.)

MOVING TEXT

These commands provide easy ways to copy or move sections of text.

^T	Tag the cursor location. This marks one end of the text to be moved.
^E	Move the section of text between the tagged location and the cursor location to a special buffer. The text is deleted from the text buffer.
^Z	Insert contents of the special buffer at the cursor location.
^F^T	Exchange the tag and cursor. This lets you see both ends of the block you have defined — the block remains the same. This command is also useful for moving between two different sections of text.

To move a block of text, type Alt-T at the beginning of the block, then go to the end and type Alt-E. Move the cursor to the destination and type Alt-Z.

To copy a section of text without deleting it, type Alt-Z immediately after the Alt-E. Subsequent Alt-Zs produce copies of the text elsewhere.

AUTOMATIC INDENT

When entering text in Insert or Overtype mode, a Return normally brings the cursor back to column 0. However, it is possible to indent this margin. Indents work best in Overtype mode; they're useful when writing code in structured languages, when writing outlines, or when dealing with columns of data.

^F^I	Set the auto-indent level to cursor column.
^F^P	Increment the auto-indent level by four columns and move the cursor to the new indent.
^F^O	Decrement the auto-indent level by four columns and move the cursor to the new indent.

MISCELLANEOUS

^S	Repeat the next keystroke four times. If multiple Alt-Ss are struck, the repeat count is multiplied by four with each Alt-S. For example, ^S^S^Sa is equivalent to striking "a" 64 times. Alternately, if Alt-S is followed by a number, the next non-numeric keystroke repeats that number of times. ^S12^D deletes 12 characters.
^_	Edit command string. If you make a mistake when entering a long command string, this command lets you edit the string. When you type Alt-_, the old command buffer becomes the text buffer, and can be edited just like text. Striking Alt-_ again restores the old text. The edited command string returns to the command area, ready for execution.
^C	Abort. Clears the command area. If you type Alt-C while a command string is being executed, execution aborts at the earliest opportunity.
^L	Insert a line. Inserts a new line into the text and puts the cursor at the beginning of the new line.
^F^F	Redraw and reformat display. This is usually necessary only if your screen is disturbed by a "foreign influence."
^F^S	Shift default case. Characters toggle between upper- and lowercase.
^F^C	Change the case of the character at the cursor and advance the cursor one character space.
^R	Insert last deleted item at cursor position.

COMMAND OVERVIEW 2.6

A command consists of one, two, or three characters that are entered into the command buffer. The command is executed by striking the ± key twice. This escape character sequence appears on the screen as two dollar signs. **Throughout this manual, $ indicates an escape.** Control characters appear in the command line as a caret (^) followed by the associated uppercase character. A Return, for example, appears as ^M.

NUMERIC ARGUMENTS

Many commands can take "numeric arguments" — numbers that precede the command and give additional information to PMATE-86. For example, while D$$ deletes one character from the text, 3D$$ deletes the next 3 characters. Numeric arguments take integer values from −32768 to +32767. They can be complex expressions (explained in Chapter 3). If an argument is missing, it is usually assumed to be 1. A minus sign before a command is usually equivalent to −1.

COMMAND STRINGS

PMATE-86 gets much of its power by combining a number of commands into a command string. For example, M is the command to move the cursor a number of character positions, and M$$ moves the cursor one character space. The command string D5MD$$ deletes one character, then moves over 5 character spaces and deletes the character there. The command string appears on the screen as it is entered; it is not until two consecutive ±s have been entered that any change takes place in the text. At this time, the whole command string is executed. Single ±s can be freely inserted between commands without causing execution. So, D5MD$$, D5M$D$$, and D$5M$D$$ all have the same effect.

STRING ARGUMENTS

While numeric arguments often precede commands, some commands are followed by "string arguments." These arguments can be a string of characters you want to insert into the text, a string you want to search for, or a string you want to do some other operation upon. For example, I is the command that inserts the string argument following it into the text buffer. Suppose the text buffer contains:


```
PMATE-86 is a very easy-to-use and helpful text editing program.
```

Typing the command Inot $$ might help you express your true feelings:

```
PMATE-86 is not a very easy-to-use and helpful text editing
program.
```

If you want to enter the Insert command as part of a command string, you must tell PMATE-86 that the string argument is finished and a new command is being entered. Do this by using an escape to separate the string argument from the command.

REEXECUTING COMMANDS

What happens to the command string after it has been executed? Simple — it stays in the command area of the display, followed by the two escapes that caused it to be executed. If you type another escape the command is repeated. If you type a backspace, the second escape is deleted, and the old command string can be modified or extended. If you enter any other command character, the old command string disappears, and the new character becomes the first character of a new command string.

The ability to repeat commands easily can be extremely useful. For example, "S" is the search command. Shello$$ searches through the text (starting at the cursor) and leaves the cursor located just after the first "hello" that it finds. The text display shows you immediately whether this is the occurrence of "hello" you are interested in. If not, hit the Escape key and PMATE-86 finds the next occurrence of "hello." Continue until you locate the section you want.

ERROR MESSAGES

Some commands and conditions produce error messages. These messages are usually self-explanatory. If Shello$$ is executed and "hello" is not found, then the "String not found" message appears. PMATE-86 stops executing the command string as soon as the command produces an error. The cursor in the command display area points to the command just after the one that caused the error. **The error message won't go away until you hit Return or the Space bar**. The command remains in the command area as if it has finished executing. You can now reexecute, modify, or ignore it.

BASIC COMMANDS

PMATE-86 has enough commands to keep you busy for a long time. However, a few basic ones are all you need for most text editing work. In the following list, n indicates a numeric argument.

nD — Delete n characters starting at the cursor. If n is missing, it is assumed to be 1.

nK — Kill n lines starting at the cursor. If n is missing, it is assumed to be 1.

I — Insert the string that follows. (End the string by typing a ±.) Igarbage$$ inserts "garbage" just ahead of the cursor.

S — Search for the string which follows S. The string ends with a ±. The search starts at the cursor. If you enter Sgarbage$$, PMATE-86 looks through the text for "garbage" and moves the cursor to just after the next occurrence. If the string is not found, an error message is produced.

C — Change the first occurrence of the first string following to the next string following. If you enter Cgarbage$junk$$, PMATE-86 will search for "garbage" and change the next occurrence to "junk." If "garbage" is not found, an error message is given. Remember that the search for garbage begins at the current cursor location.

You can use the following commands for moving blocks of text:

nBC — Copy n lines of text into a special buffer. If n is missing, it is assumed to be 1.

nBM — Move n lines of text into a special buffer. If n is missing, it is assumed to be 1. BM is like BC except that the lines copied into the special buffer are deleted from the text buffer.

BG — Insert contents of special buffer into text just before cursor.

To move 5 lines of text, put the cursor at the beginning of the lines to be moved (using the cursor control Instant commands). Then, type 5BM$$. The 5 lines disappear from the text. Then, move the cursor to the new location of the lines and type BG$$. This restores the lines. (The special buffer still contains those 5 lines.)

Tags

If you want to move a large block of text around, it may not be obvious how many lines are in the block. There is an alternative to counting lines. First, put the cursor at the beginning of the section you want to move. Use an Alt-T Instant command to "tag" that location. Then, move the cursor to the end of the block. An Alt-F followed by an Alt-T exchanges the tag and the cursor, letting you examine the boundaries of the block. This procedure does not affect the definition of the block. The special symbol "#" causes

the next command to act on this whole block. So, #BC copies the block into the special buffer, #BG copies the block from the special buffer into the text, and #K deletes it.

When used in front of a command that takes a numeric argument, # indicates the number of lines or characters that the command acts upon (such as D or K). Even if the command normally acts upon a fixed number of lines (such as the B commands), parts of a line can be moved by tagging a position.

The Garbage Stack

When PMATE-86 deletes text, it dumps that text on a "garbage stack." A certain amount of space is reserved for this stack; any memory space not used by text is also used for garbage. If you accidentally delete a line, it's easy to restore that line with an Alt-R. Alt-R "pops" the last item off the stack, and puts it back in the text. The most recently deleted item is available first. Items lost long ago may have gone permanently out to pasture if there is not enough memory to hold them all.

You can also use the garbage stack as an easy method to move a bit of text around. To move a line of text, for instance, put the cursor at the beginning of that line and type Alt-K. Then, move the cursor to the required destination, and type Alt-R. Use Alt-W to move a word or two around in a sentence.

GO TO IT 2.7

At this point, you know enough to utilize PMATE-86 very effectively. By using the commands, Instant commands, and other information covered in this chapter, you can confidently enter and modify programs and other text. It is important to master the material in this chapter before attempting to learn the complete command set.

3

ADVANCED TEXT EDITING

SIGNED NUMERIC ARGUMENTS 3.1

Until now, we have assumed that all numeric arguments are positive integers. They can in fact be much more complex expressions. For now, we'll look just at negative numbers.

Commands with a negative argument work backwards through the text. −3D deletes the three characters just preceding the cursor (leaving the cursor located at the same character). Similarly, −2K deletes the two lines preceding the cursor. −S searches backwards through the text (from the cursor) until it finds the string which follows the "S".

TEXT FORMATTING 3.2

WORD WRAP

Editing text presents very different problems than editing programs. Suppose you enter the following paragraph:

> **Editing textual material presents very different problems than editing programs. You write the following paragraph:**

Soon you decide that you want to insert another sentence between the two you've already written. You need to be able to add this sentence and still keep the right number of words on a line. As you add words between the existing sentences, the PMATE-86 Format mode automatically takes care of line formatting.

In Format mode, lines end not only on a Return, but also on the last empty space before the line exceeds the allowed length. (Words are never broken up.) The rule is to enter text without any Returns — PMATE-86 takes care of the line length for you. Use a Return only to end a paragraph or at a place where a new line is always necessary. When in Format mode, each return character is displayed as a "<".

To enter Format mode, use the command F. Repeating F restores PMATE-86 to normal mode. You can also use the F command with a numeric argument. 30F enters Format mode and sets the maximum line length to 30 (the default is 77).

PMATE-86 always keeps the screen up to date and properly formatted. You may find this annoying while entering text in the middle of a paragraph, since the margins can change with every keystroke and produce a display that jumps around quite a bit. If this bothers you, an Alt-L Instant command inserts a Return and stabilizes the situation by moving the cursor to the end of a paragraph. As soon as you are finished with the addition, type an Alt-D to delete the excess Return.

INDENTS AND MARGINS

It is very useful to be able to indent sections of text. One method is to precede each line with one or more Tabs. This method causes a problem, however, as the Tab is now fixed between two specific words. As words are deleted or inserted, the Tabs slide around to different locations on the screen and play havoc with your margins. For this reason, PMATE-86 interprets the Tab as a margin indent character when in Format mode. If an indent is set to the same column as the Tab, putting a Tab ahead of an indented section causes each subsequent line to indent to the same point until a Return is reached.

An indent is set with a YI command.

nYI Set an indent at column n. Any Tab to column n causes the rest of the paragraph to be indented.

For example, 8YI sets an indent at column 8 (the first Tab stop). If you follow nYI with a Tab to column 8, the rest of the paragraph is indented.

When you use the YI command, be sure not to use a Tab to indent the first word in a paragraph. If you do, it indents the whole paragraph. Type in the spaces instead.

For some applications, you might want to change the left and right margins only for a particular section of text — say to move the left margin over 40 spaces to accommodate a picture. PMATE-86 lets you enter margin and Tab information in a special non-printing control line in the text. This control line begins with an Alt-F (F for Format), and ends in a Return:

^FL20;R60←┘

This control line changes the left margin to 20 and the right to 60 from that point onward. These margins are reflected in the text display. (You might find it hard to enter that Alt-F in text, since Alt-F is also an Instant command. See Section 3.3 for help.) After altering a format line, an F$$ command tells PMATE-86 to recompute its formatting and bring everything up to date.

One final thought: Format mode is useful when writing programs, too. If the language you are using supports a start- and stop-comment command (so that comments don't automatically end with a line), using PMATE-86 in Format mode lets your programs read like a book, with extensive, easily modified comments. Of course, program lines must all be terminated with Returns, but comments can have as many lines as you want.

CONTROL CHARACTERS 3.3

Since control (Alt) characters are used as Instant commands, it might seem difficult to enter a control character into the text. You can do this using an Alt-7. When you enter this character, nothing happens. The next character to be entered, however, becomes the equivalent control character. To enter an Alt-F into the text without executing a command, enter an Alt-7 and then F.

3.4 INPUT FILES, OUTPUT FILES, AND AUTOMATIC DISK BUFFERING

You must answer two questions before PMATE-86 can do any editing:

1. Where is the text to be edited?

2. Where should the text be put after it's been edited?

You answer these questions when you first invoke the editor. For example:


```
PMATE GARBAGIN GARBAGOU
```

is the command to start editing the file GARBAGIN. It is opened as the input file. Changes and additions are made, and the result is left in the file GARBAGOU — the output file.

PMATE-86 normally operates with automatic disk buffering in effect (referred to as "auto-buffer mode"). This means that you can edit a file larger than available memory without having to read text in and write it out explicitly. PMATE-86 reads text in from the input file as needed and writes it out to the output file when it has been processed. XE completes the edit and finishes transferring the edited text from the input file to the output file. XE does not return you to the operating system — you are still in PMATE-86. You can return by using an XH command or you can use XF to open new input and output files. (XFGARBAGIN GARBAGOU$$ opens the same files as above.)

It is also possible to use PMATE-86 in a manual mode, giving you exact control over which sections of text are in memory, and which sections are on the disk. Manual mode is discussed in detail in Section 3.5. Even if you always use PMATE-86 in auto-buffering mode, read that section to understand the actions PMATE-86 is automatically doing for you.

Often, you'll do an editing operation to update a file. When you finish, it's convenient to give your output file the same name as the input file. You can do this by deleting the old input file and assigning its name to the output file. PMATE-86 does this for you automatically if you specify only one file name in the command line (or in an XF command). PMATE-86 opens an input file

with the specified file name and an output file with the same name and extension .$$$. XE then outputs everything to the output file as usual. When the edit is finished, the old input file is given extension .BAK and the output file is given the name of the original input file.

For example:

PMATE JUNK.ASM

opens JUNK.ASM as the input file and JUNK.$$$ as the output file. XE then renames JUNK.ASM to JUNK.BAK, and JUNK.$$$ to JUNK.ASM.

In the PMATE-86 command line (or in an XF command), the input file or the output file can be preceded by a drive specifier (A:, B:, etc.) that tells which disk contains the file. If there is no specifier, the logged disk is assumed to be the one you want.

MANUAL MODE 3.5

In manual mode, you can break the input file into pages that you can read from the input file one or two at a time. Then, you can write them to the output file a few at a time. XA is the command that reads in the next page, appending it to the text buffer. XA can even take a numeric argument — 5XA will append 5 pages. The command nXW writes out n pages from the beginning of the text to the output file.

It is also possible to edit backward through a file. –XA brings back text already written to the output file by the XW command. –XW writes out text from the end of the current buffer, effectively putting it back at the end of the input file. (Actually, it is written to a temporary file called PMATE.TMP in order to preserve the input file.)

The nXR command is a very useful one. (It is equivalent to nXWnXA.) 2XR writes two pages from the beginning of the buffer to the output file and then reads two pages from the input file into the buffer file. The XE command means "all done." XE writes the text buffer to the output file, reads in the rest of the input file, and then writes it to the output file.

Once you set it, the size of a page is a fixed number of lines. This number is set by using the nQP command. 75QP sets the page size to 75 lines: the command 3XA appends 225 lines and XW writes 75 lines. Pages can be ended prematurely by a form feed character (Alt-L). If the page size is set to 0 (0QP), form feeds are the only method of separating pages.

If you run out of memory space while you are entering text, use XW to write out some of the text at the beginning of the buffer. XA brings in more text from the disk. If you need to start a new pass, XJ writes all text out to the output file and then reopens the buffer for input.

3.6 GLOBAL COMMANDS

Although some commands work only on text presently in memory, many commands also have a "global" version. Global commands begin with U, and proceed through an entire file, reading and writing to disk as necessary. For example, A and Z move the cursor to the beginning and end (respectively) of text in memory. UA and UZ move to the beginning and end of the entire file. (If you are editing near the end of a long file, XJ is often a better choice than UA for getting back to the beginning. XJ writes the remainder of the edit file to disk and reopens the edit file from the beginning; UA must scroll back through the entire file.)

The search command S is a "local search," proceeding only through the text currently in memory. The global command US carries the search through the entire file. By using S rather than US, you are protected from inadvertently searching the entire file for something you expect to find nearby. Similarly, the C command performs a local replace operation, while UC is used globally.

3.7 DIRECTORY MAINTENANCE

PMATE-86 lets you do disk directory lists and file deletes. This is handy if you get a "Disk full" message when trying to write a file to disk. You can then list your directory, delete unwanted files, and make another attempt to write the file out to disk.

XL lists the entire file directory, entering it into the text buffer at the cursor location. This lets you scroll through large listings and edit the directory just like any other text. There can be problems, though, because the directory may appear in the middle of your working text. In this case, you can always delete it. Other ways are to edit in another buffer (see Section 3.9) or use the ^_ Instant command to edit the command string.

Partial directory listings are obtained by following XL with a file name. As in the MS-DOS DIR command, the file name can contain ?s and *s.

XLJUNK inserts the file name JUNK at the cursor if file JUNK exists; otherwise it does nothing. XL*.COM inserts the names of all files with extension .COM at the cursor.

Files are deleted with the XX command. XXfile deletes "file" from the disk. The file name cannot contain the ambiguous characters ? or *. *Do not delete the input or output files.*

You can also switch the logged disk drive by using the XS command. XSA selects drive A.

ITERATION 3.8

Often you'll want to repeat a command or a command string. The iteration brackets ([ ]) let you to do this easily. The command string:

```
5[Igood morning!
$]
```

produces this display:

```
good morning!
good morning!
good morning!
good morning!
good morning!
```

3[K] produces the same result as 3K.

Iteration brackets can be "nested," as long as you make sure you have the same number of left and right iteration brackets. The command:

```
100[40[I*$]I
$]
```

fills up your text buffer with 100 lines of 40 stars each.

If there is no numeric argument in front of the iteration brackets, the operation is repeated forever (that is, about 65,000 times) or until an error occurs. [I*$] fills up all available memory with stars, and then tells you that there is no more memory left. [K] starts erasing lines and continues until there are none left. [Cgoodbad] changes all occurrences of "good" (after the cursor) to "bad".

3.9 OTHER BUFFERS

PMATE-86 has 11 buffers that you can enter text into (as well as two buffers for command strings). The size of a buffer varies — any one expands to fill most of the remaining memory. If you delete text from one buffer, the space that you free is available to any of the others.

Usually, you edit in the T (text) buffer. This is the only buffer that has an associated edit file and that supports automatic disk buffering. The ten other buffers are labeled 0-9. (You already know about the 0 buffer — all special buffer commands, like BC and BG, copy to or from buffer 0.) These buffers are used for temporary work space, for storing blocks of text that will later be moved, and for storing command strings (macros). To edit in a buffer other than the T buffer, type BnE (buffer n edit) where n is 0-9 or T. B3E opens buffer 3 and BTE puts you back in the text buffer again.

MACROS 3.10

A macro is a long command string that tells PMATE-86 to do a series of operations in a particular order (much like a subroutine). If you have a command string you need to use several times, put that command string in buffer n. Any time you like, you can execute it with the command .n. If you find that you're using a macro frequently, you can make it a permanent part of PMATE-86. These "permanent macros" are executed by the command .x (where x is any character other than the digits 0-9).

There are several ways to put a command string into a buffer. The easiest way is to start editing in the buffer using the BnE command. Then, go into Insert mode and enter the command into the buffer.

Macros can be "nested"; that is, one macro can in turn call other macros. Macros can also require passed string arguments. You will learn later how to pass arguments to macros and how to create your own permanent macros.

ERROR TRACEBACK 3.11

Sometimes errors occur while a macro is executing. In this case, the usual error message appears in the text area of the screen. In the command area, however, the macro string that caused the error is displayed with the cursor pointing to the command character just after the one that caused the error. The status line tells which buffer (or which permanent macro) was being executed at the time of the error. You have a choice of hitting a Return or the Space bar. A Return behaves as usual — you'll be ready to enter the next string. Striking the Space bar "pops a level," and lets you see the command string that called the troublesome macro. If this command string is also a macro, you can hit the Space bar again and pop another level. When you reach the original command string, the Space bar and Returns have the same effect. If a macro is called from several places in a command string, this error traceback facility lets you find out exactly where the trouble occurred.

3.12 AUXILIARY FILE I/O

At any time, PMATE-86 can output sections of your current edit buffer to disk, or it can read disk files into the current buffer. This can occur while input and output files are defined, and does not upset them. XIfile inputs all of "file" and puts it just before the cursor. nXIfile reads in n pages from "file." NxI can subsequently read in more pages. (If no file name is specified, input continues from the last named auxiliary input file.) nXOfile outputs the next n lines of text (after the cursor) to "file." If there is no numeric argument n, the entire edit file is output.

You can use these commands to merge sections of files (even if larger than available memory), load macros into buffers for execution, and use the disk for temporary storage.

3.13 DUPLICATING PMATE-86

You probably have noticed that PMATE-86 has a number of "parameters" that are easily changed. Sometimes your favorite parameters differ from the default values. If you like a page size of 100 lines, you can give the command 100QP every time you begin editing — or you can create a custom version of PMATE-86 that automatically makes 100-line pages. Here's how:

1. Execute PMATE-86 without input or output files and make any desired changes.

2. Give the command XDfile (D for "duplicate") where file is the name of your new version of PMATE-86 (the .COM extension is added automatically).

3. Give your new PMATE version a name (PMATE1 or PMATE2 or anything you like).

4. Use XH to return to the operating system, and then verify the new version. If you are happy with it, you can erase the original PMATE.COM or you can keep several versions around for different purposes.

GET SOME HARD COPY 3.14

PMATE-86 outputs your entire edit file to a printer if you use the command XT. If there is a numeric argument, nXT prints the number of text lines you specify, starting at the cursor. Use this command when you want to print out only the changes you have made to a file. If you feel ambitious, you can write macros to output text in any format you want.

4

COMPLETE COMMAND SET

NUMERIC ARGUMENTS AND VARIABLES 4.1

Numeric arguments are integers — usually signed numbers between −32,768 and 32,767. Sometimes they are unsigned numbers from 0 to 65,535. However, numeric arguments are more than just decimal numbers — they can be complex expressions consisting of numbers, variables, arithmetic and logical operations, and parentheses.

In numeric arguments, operations are done from left to right. Any operator precedence must be determined by parentheses. So, 5 + 3*2 has the value 16 and 5 + (3*2) has the value 11. You can put up to 15 levels of parentheses in an expression.

Numbers in command strings are usually decimal numbers (base 10). However, the base (or current input radix) can be changed. (See the Q commands.) 10D$$ usually deletes 10 characters, but if the input radix is 8 (octal), it deletes only 8 characters.

If the radix is greater than 10, several rules must be observed. In hex, for instance, PMATE-86 must know if D is the hex digit D or if it is the command to delete a character. The rule is that any number must begin with a digit from 0-9; then each succeeding character is interpreted as a digit if at all possible. If the input radix is hex, DDK means to delete two characters and then erase a line. 0DDK, however, erases 221 lines (the value of DD in hex). If you need to terminate a hex number, an escape can be used. 0D$DK deletes 13 characters and then erases a line. 2$D will delete two characters, while 2D is interpreted as 45 (decimal).

You can display numeric arguments on the status line. If you type a numeric argument followed by two escapes, the status line shows the value of that argument in the current output radix (decimal by default) after the words ARG = . In this way, you can use the editor to do integer arithmetic. If you make the output radix different from the input radix, you can do number conversions (such as hex to decimal).

ARITHMETIC OPERATIONS

The following are valid arithmetic operations within a numeric argument.

+ Addition.

– Subtraction or negation. –(3) is a valid expression.

* Multiplication.

/ Integer division, leaving just the quotient. The remainder of the last division performed is available as @R (see the @ numeric arguments).

LOGICAL OPERATIONS

Logical operations leave the value –1 if true, and 0 if false. The following are valid logical operations within a numeric argument. (In the expression 3=5, 3 is the first operand, and 5 is the second operand.)

= Equal. True if the first and second operand are equal.

< Less than. True if the first operand is less than the second.

> Greater than. True if the first operand is greater than the second.

& And. True if both operands are true.

! Or. True if either operand is true.

' Logical complement (Not).

Examples:

3<2 has the value 0.

3<2 ' has the value –1.

2<3 has the value –1.

$2<3!(5=2)$ has the value -1.

$2<3\&(5=2)$ has the value 0.

$5+3=(1+7)$ has the value -1.

$5+3=(1+7)'$ has the value 0.

VARIABLES AND THE NUMBER STACK

There are ten numeric variables (0-9) available for your use. They are set with the V command and used as part of a numeric argument having an @ argument. A number stack is also available. Any numeric argument can be "pushed" on this stack (see "n;") and "popped off" later (see @S). The stack holds up to 20 entries during the execution of a command, but is cleared upon completion. Some of the variables used internally in the editor are also available for use in numeric arguments. Here is the complete list of @ arguments:

@i	The value of variable i, where i is a digit 0-9.
@A	The numeric argument preceding the last macro call.
@B	The current edit buffer: 0 if buffer T, 1 if buffer 0, 2 if buffer 1, . . . 10 if buffer 9, and 11 when editing the command line.
@C	The current character number. This is the number of characters from the beginning of the text buffer to the character at the cursor. This value is 0 when the cursor is at the beginning of the buffer.
@D	Returns the number of lines (set by QL) scrolled by the multiple-line movement Instant commands ^U and ^J.
@E	The value of the error flag.

@Ffile$ Returns −1 if the file exists on the current directory, 0 if it doesn't.

@G The length of the string argument just referenced by an I, S, or C command.

@Hstrng$ Compares "strng" to the characters at the cursor in the current text buffer. Returns 0 if they match, otherwise 1 or −1, depending upon which string is greater. Wildcards (as in S command) are acceptable in the command string.

@I The number of pages read in from the input file. Pages are counted only if they are delimited by form feeds. Pages written backward to or read from the temporary file PMATE.TMP are not counted.

@J The number of screen lines available for text display. Does not include the three status and command lines at the top of the screen.

@K The ASCII value of the key struck after a command.

@L The current line number. If the cursor is on the first line, this is 0. This value is the line number of text in memory if auto-buffering is off (or when not in buffer T), and the line number in the entire edit file when auto-buffering is on.

@M The amount (in bytes) of working memory space remaining.

@O Number of pages written to the output file. Pages are counted only if they are delimited by form feeds. The page number is decremented for each page read back in by the automatic disk buffering action, or by a −XA. @O gives the page number of the line at the top of memory (A command), so the current page can be quickly computed.

@P The absolute memory address pointed to by the cursor.

@Q The column of the previous Tab stop.

@R	The remainder of the last division performed.
@S	The value of the top of the number stack. The number stack is popped.
@T	The ASCII value of the character pointed to by the cursor.
@U	Indicates whether auto-buffering is in effect. Returns −1 if it is, and 0 if not.
@V	The current mode — 0 for Command, 1 for Insert, 2 for Overtype.
@W	The right margin.
@X	The column containing the cursor.
@Y	The left margin.
@Z	The column of the next Tab stop.
@@	The value of the memory byte pointed to by variable 9.
@/	The indent setting.
"x	The ASCII value of the character x, where x is any character.

BLOCK OPERATIONS

Commands which take a numeric argument to indicate the number of characters or lines can also be used to act upon a defined block.

T	Tag the current cursor position as the beginning of a block. (Equivalent to Alt-T Instant command.)
#	Move the cursor to the tagged position, and use the difference between the old cursor and the tagged position as the numeric argument.

If you want to type out a large block of text, move the cursor to the beginning of the block and tag that position with T$$ or Alt-T. Then, move the cursor to the end of the block and print out your text with #XT. # is also useful with delete and buffer commands.

A tagged block must reside entirely in memory. If a tagged position is moved to the disk, attempting to access the block with a # command produces the "Block too large" error message.

VARIABLE AND NUMBER STACK COMMANDS

nVi — Set variable i (a digit from 0-9) to the value of numeric argument n. @C+3V2 sets variable 2 to 3 greater than the current character position.

nVAi — Add the value of numeric argument n to variable i. If n is missing it has the default value of 1. 3VA5 adds 3 to variable 5.

n, — Push numeric argument n on the number stack.

4.2 THE ERROR FLAG

Certain commands produce non-fatal error conditions. For example, an M command cannot move the cursor if it is already at the end of the text buffer. The command string execution is not interrupted to give an error message. However, you can determine that an error condition exists by looking at the error flag. It is possible to suppress some fatal errors, such as those that occur if a string cannot be found during a search command. If these error messages are suppressed, the error flag tells you whether an error has occurred.

@E — Gets the value of the error flag. The error flag is reset before executing a command string, and every time it is tested by @E. It is also reset when beginning an iteration.

E — Set the error suppress flag. This flag is reset before executing a command string, and by every command that might test it.

MODE AND FORMAT COMMANDS 4.3

nN — Change modes. If n is 0, remain in Command mode. If n is 2, go into Overtype mode. For any other n (or if n is missing), go into Insert mode.

PMATE-86 has an automatic word-wrap feature when in Format mode. This feature ends a line after the last complete word that fits within the allowed line length. A Return is entered only to indicate that the next word must begin on a new line (i.e., end of paragraph).

nF — Enter Format mode. The line length is set to n.

F — Toggle in and out of Format mode.

CURSOR MOTION COMMANDS 4.4

The following commands move the cursor. While you can also move the cursor using Instant commands, you need cursor motion command characters when constructing command strings.

+/−nM — Move the cursor n characters. If n is positive, the cursor moves forward. If n is negative, the cursor moves backward. If n is 0, no action is taken.

+/−nL — Move the cursor n lines. Consider the following example:

```
line a
line b
line c
line d
line e
```

Suppose the cursor is on the e in line c. 1L or L moves the cursor to the beginning of line d. 2L moves it to the beginning of line e. 0L moves to the beginning of the current line (c). −L or −1L moves the cursor to the beginning of line b, while −2L moves it to line a.

+/−nP — Move the cursor n paragraphs. When not in Format mode, this works like L. When in Format mode, it seeks the Return that makes the next word begin on a new line.

+/−nW — Move the cursor n words. Words are separated by any combination of spaces, Tabs, and Returns. 0W moves to the beginning of the current word. If n is negative, the cursor is moved to the beginning of the nth preceding word. If n is positive, the cursor is moved to the beginning of the nth following word.

A — Move the cursor to the beginning of the text buffer.

Z — Move the cursor to the end of the text buffer.

If an M, L, P, or W command would make the cursor go past the end of the edit buffer, the cursor goes only to the end of the buffer and the error flag is set. Similarly, if the cursor would be moved past the beginning of the buffer, it moves only to the beginning, and the error flag is set. The value of the error flag is obtained with the numeric argument @E. It is −1 (true) when set, 0 (false) when clear.

4.5 DELETION COMMANDS

+/−nD — Delete n characters starting at the cursor. If n is positive, characters are deleted from the cursor position to the end of the text buffer. If n is 0, no action takes place. If n is negative, the first character deleted is the one just before the cursor. Characters are then deleted toward the beginning of the text.

+/−nK Kill n lines starting at the cursor. K deletes all characters from the cursor up to and including the Return at the end of the line. 2K deletes the next line as well. 0K deletes characters from just before the cursor up to (but not including) the Return at the end of the preceding line. −1K deletes the preceding line also. 0KK deletes the line containing the cursor.

INSERTION COMMANDS 4.6

I Insert the following string into the text just ahead of the cursor. Istring$ inserts "string".

nI If I has a numeric argument, the character represented by that ASCII value is inserted into the text. If the input radix is decimal, 65I inserts A. Any character can be inserted with this command.

n\\ Insert the ASCII string representing the value of argument n in the current output radix. The string is inserted immediately before the cursor. If variable 0 has the value 23, @0\\I $@0+3\\ inserts "23 26" into the text.

R Replace the text immediately after the cursor with the following string. No text is moved; the new characters overwrite the old text. If the cursor is too close to the end of the text buffer, an error message is given and the substitution is not performed.

nR When R has a numeric argument, the character represented by that ASCII value replaces the one already at the cursor position.

4.7 STRING SEARCH COMMANDS

+nS — Starting at the cursor, search forward for the following string. If n is present, search only the next n lines (defined as in the L command). If n is missing, continue the search until the end of the edit buffer is reached. The cursor is left positioned just after the located string.

−nS — Starting just before the cursor, search backward for an occurrence of the following string. If n is present, search only the preceding n lines (defined as in the L command). If n is missing, continue the search back to the beginning of the edit buffer. The cursor is left positioned on the first character of the located string.

Normally, an error message is given if the string is not found. However, in some instances you need to continue execution of a command string after all occurrences of the string have been found. In this case, no error message is given. Command execution continues if the error message suppress flag is set. This flag is set by the E command, and is reset on completion of every search. If an error does occur while the error message suppress flag is set, the error flag is set. The value of the error flag is given by @E. It is −1 (true) when set; 0 (false) when clear.

Uppercase characters in the search string match only uppercase characters in the text. Lowercase characters match either upper- or lowercase in text. (To match only lowercase, see ^L wildcard.)

The following "wildcards" are used in the search string to match any of several specified characters.

^N — Match anything but the following character. SMA^NTE$ finds "MALE" or "MADE" but not "MATE".

^E — Match any character. MA^EE matches "MALE", "MADE", and "MATE".

^L — Take next character literally. This lets you search for a wildcard character. SMA^L^EE matches neither "MALE" nor "MADE", but only "MA^EE".

^S	Matches either a space or a Tab.
^W	Matches any word terminator (a character other than a letter or a number).

STRING CHANGE COMMAND

nC	Search forward or backward for the string which follows, as in nS. Change that string to the second following string. Cstring1$string2$ locates the first occurrence of string1 and replaces it with string2. If the string is not located, errors are treated as with S. In particular, error messages can be suppressed.

GLOBAL COMMANDS

Some commands which act only on the text in memory have "global" counterparts. These, if necessary, read in more text from the disk, and search through the entire edit file.

UA	Move the cursor to the beginning of the edit file.
UZ	Move the cursor to the end of the edit file.
nUS	Starting at the cursor, search forward for the following string. If n is present, search only through the next n lines (defined as in the L command). If n is missing, continue the search until the end of the edit file is reached. The cursor is left positioned just after the located string.
nUC	Search forward or backward for the following string, as in nUS (wildcards are allowed). Change the located string to the second following string.

SETTING TAB STOPS

Tab stops are set every 8 spaces by default. However, this assignment is easily modified. A maximum of 15 Tab stops can be defined.

YK	Erase all Tab stops. A Tab is now equivalent to a space.
nYS	Set a Tab stop at column n.
nYD	Delete the Tab stop at column n (if there is one).
nYE	Erase all old Tab stops, and set new ones at every nth column. 8YE restores the conventional settings.
nYI	Set the default indent to column n. If n is 0, no indent is used. (See the next section for use of indents.)

For example, YK10YS30YS sets Tab stops at columns 10 and 30. This might be useful for assembly language programming with labels in the first column, then instructions, and then comments. You could then save a version of PMATE-86 containing these Tab settings (see XD command).

The following commands make it easy to change Tab settings without changing the rest of the text.

nYF	For the next n lines (beginning at the cursor), replace all Tabs with an equal number of spaces.
nYR	For the next n lines (beginning at the cursor), replace blocks of spaces by Tabs wherever possible.

IN-LINE TEXT FORMATTING

While in Format mode, you can set Tab stops and margins in non-printing control lines placed directly in the text. You must do this when any of these parameters change within the text. Even when this isn't the case, it's still a good idea to put this format information on the first line of the text file. You won't need to wonder which margins and Tab stops you used the last time you edited the file.

Control lines must begin with an ^F and end in a Return. These lines are not printed by the XT command, so that usually unprintable language can be entered. Certain letters are recognized as commands and must often be followed by a number. These commands can be strung together if separated by semicolons.

Ln	Set the left margin to column n.
Rn	Set the right margin to column n.
K	Erase all Tab stops.
Sn	Set a Tab stop at column n.
Dn	Delete the Tab stop at column n.
En	Erase old Tab stops, and set new ones at every nth column.
In	Set an indent to column n. If n is also a Tab stop, tabbing to this column causes subsequent text to indent to column n until a Return is reached.

The line:

^FL5;R50;E10

sets the left margin to column 5, the right to column 50, and sets a Tab stop at every 10th column.

Any margin or Tab stop information not specified in the format line reverts to the default: 0 for the left margin; the right margin default is set by the F command; and the Tab stop defaults according to the Y commands.

FLOW CONTROL COMMANDS

Conditional branching and iteration within commands let you construct command strings equivalent to small text editing programs. Iteration is

accomplished as follows:

n[...m]

where:

"..." represents any command string.

This command string is executed n times. If n is missing, it is iterated 64K times. If n is 0, the command string in brackets is skipped over. If n is −1, the command string is executed once. If iteration brackets are preceded by a logical expression, the enclosed command string is executed once if the expression is true, and skipped over if the expression is false. If m (an optional numeric argument) is present, iteration of the loop ends prematurely if m becomes non-zero (true). If m is missing, its value is that of the error flag; that is, iteration of the loop terminates if the error flag is set.

5[D] has the same effect as 5D. 5V0[D−VA0@0=0] also has the same effect as 5D. 5V0 initializes variable 0. Within the iteration brackets, −VA0 decrements variable 0. The iteration continues until the final numeric argument is true, when variable 0 is 0.

[Chello$goodbye$] changes all occurrences of "hello" to "goodbye". [Chello$goodbye] changes the first occurrence of "hello" to "goodbye" (remember, all string arguments must be terminated by an escape).

Iterations can be nested to a maximum depth of 15.

I[...] Execute the expression in brackets if logical expression I is true. Skip past matching bracket if it is false.

I[...][...] Execute instructions within first set of brackets if logical expression I is true; otherwise execute instructions within second set. (NOTE: There must not be any spaces between the two sets of brackets.)

Further control of these iteration and if-then loops is provided by the next and break commands. These work only within matching iteration brackets.

n^ Next. If n is non-zero (true) or missing, proceed to the next iteration.

n_ Break. If n is non-zero (true) or missing, exit immediately from the enclosing iteration brackets.

As with other command characters, upper- or lowercase brackets ({ } or [ ]) are used for iteration. However, the break and next commands do distinguish case. They skip past } to the next]. Put if-then-else constructions in uppercase brackets ({ }) so that any break or next command within exits the iteration loop (not just the if clause).

NOTE: Be careful. PMATE-86 is easily fooled by iteration brackets within strings. Make sure the next bracket PMATE-86 finds is intended as an iteration bracket, and not as part of a search or insert string.

Conditional and unconditional branching within a command string is permitted. The proper point to branch to is designated by a label. (A label is any character, preceded by a colon. ":A" and ":#" are examples of valid labels.) The branch command is:

nJ If n is missing or non-zero (true), transfer control to the command immediately following the referenced label. If n is 0, proceed with normal command execution. @M>100JL$10K:L erases 10 lines if there are fewer than 100 bytes of memory left. JL$1000K:L does nothing.

Never jump in or out of an iteration loop. This leads to very erratic results.

Finally, it is possible to exit at any point from an entire macro.

n% Exit macro. If n is non-zero (true) or missing, exit from macro.

BUFFER COMMANDS 4.8

The editor contains 11 buffers that you can use for entering text. The buffer initially used is called the T (text) buffer. The other buffers are labeled 0-9. Independent text can be contained in each of these buffers. Text can also be transferred from one to the other. The buffer currently being edited is

displayed in the status line. The Instant command ^_ causes the command buffer to become the current edit buffer; C is displayed in the status line.

All buffers (including the command buffer) expand and contract dynamically. Each buffer uses as much memory as it needs, until available memory is exhausted.

In the following buffer commands, b refers to a buffer number (either 0-9 or T). In all cases, buffer 0 is assumed if b is left out. Some commands have a numeric argument (n) that refers to the number of lines to be moved or copied. The numeric argument can be positive or negative, and the affected lines are determined as in the L and K commands.

BbK — Erase buffer b. All the text in buffer b is deleted, and the space it occupied is reclaimed.

BbE — Buffer b becomes the current edit buffer. When the edit buffer is changed, the cursor location of the old edit buffer is preserved. When the old edit buffer is reinstated, the cursor is restored.

nBbC — Copy n lines from the edit buffer to buffer b. The old contents of buffer b are destroyed. The cursor in buffer b is placed at the end of the entered lines. The copied lines in the edit buffer are preserved, and the cursor is placed after them.

nBbD — Insert n lines from the edit buffer into buffer b (just before the cursor). The copied lines in the edit buffer are preserved, and the cursor is placed after them.

nBbM — Move n lines from the edit buffer to buffer b. The old contents of buffer b are destroyed. The cursor in buffer b is placed at the end of the entered lines. The copied lines in the edit buffer are deleted.

nBbN — Insert n lines from the edit buffer into buffer b (just before the cursor). The copied lines in the edit buffer are deleted.

BbG — Get the contents of buffer b and insert them ahead of the cursor. The contents of buffer b are not affected.

You can use these buffer commands to move or copy blocks of text. For example, BM moves one line of text to buffer 0 after deleting the old text there. You can then execute BN repeatedly, each time moving the next line of text to the end of buffer 0. This is just an alternative to counting lines and typing 15BM$$. In this way, you can assemble a whole block of text in buffer 0. You can then move the cursor in the edit buffer elsewhere, and BG moves that block of text to this new position.

When in auto-buffer mode, you don't usually get a "Memory space exhausted" error message. However, buffers other than T can take up the available memory. Since they are not disk-buffered, BM and BC commands may not have enough room to execute. If you need to move very large blocks of text, XO and XI commands can move them through a temporary file.

EXECUTING MACROS

The contents of any buffer can be executed as though it were a command.

.b Execute buffer b. There is no default option; b must be present.

An executed buffer can in turn execute another buffer. You can do this to a level of 15 deep.

You can use two methods to insert a command string into a buffer for execution as a macro. The easiest is to make the edit buffer the one that will hold the command string. Then, you can enter and edit the command string while in Insert mode. (In command mode, it is hard to enter an escape into the text area.) When finished, change the edit buffer back to the original.

Another way is to type the command string as if it were to be executed now. When it is finished, use the Instant command ^_ to edit the command string. BbM then moves the macro to buffer b where it can be executed by the command .b.

I% Return early from macro if I is true (non-zero) or missing. This is like a subroutine RET statement in that I% makes it easy to leave a macro if a specified condition is met.

4.9 STRING ARGUMENTS

Commands such as I, S, and C take string arguments. String arguments usually follow the command directly, but there are ways to get the arguments from other places (i.e., the contents of a buffer). An Alt-A tells the editor that this is not an ordinary string argument.

^A@b — Get string argument from buffer b. When a buffer is executed as a macro, the macro can get string arguments from the command string that called it. Suppose buffer 2 contains "trash". In that case, S^A@2$ searches the text for "trash". I^A@0$ is equivalent to BG.

^Aa — Get string argument from calling command, where a is a letter from A-Z. A refers to the first passed argument, B the second, and so on.

This should be clearer after an example. Suppose buffer 1 contains:

```
IDear Mr. $I^AA$I,
    You, Mr. $I^AA$I have the opportunity to be the first on
your block in beautiful $I^AB$I to own your own copy of an
exciting new editor. Imagine what you and Mrs. $I^AA$I can
do with it. The rest of $I^AB$I will be so jealous. Blahhh,
blahhh, blahhh$
```

The command .1Jones$Cambridge$$ enters the following into the 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. Blahhh,
blahhh, blahhh
```

Unfortunately, that is not all this command does. After .1 is executed, the editor comes back and executes the command J. When it tries to execute buffer 1, PMATE-86 can't tell how many string arguments are required. Consequently, PMATE-86 doesn't know where in the command string to return in order to execute the next command. Buffer 1 must contain the needed information. The number of passed string arguments must be set in the macro by the QA command (see Q commands).

If buffer I contains 2QAIDear . . . , then .1 Jones$Cambridge$$ does not attempt to execute the J.

When macros are nested several levels deep, the string arguments can also be nested.

COMMAND STRING FORMATTING 4.10

Since command strings are actually text editing programs, PMATE-86 has facilities that format these command strings for easy reading and modification. Spaces, Tabs, and Returns are all ignored as commands. They can be placed between commands to enhance readability.

; A semicolon indicates that what follows is a comment. All characters through the next Return are ignored.

A command string can be written to look like a well-commented program. For example, here's a short command string that changes all uppercase alphabetic characters to lowercase:

```
A                   ;start at beginning of edit buffer
[                   ;begin iteration
@T<"A  JA           ;if the current text character is not an
                    ;alphabetic character (if its ASCII value is
                    ;less than that of "A"), jump to label "A"
@T!"  VO            ;change character to lowercase by ORing
                    ;it with ASCII value of space (20H)
                    ;save result in variable 0.
D  @OI              ;delete old character and insert shifted one.
-M                  ;move back to same character.
:A        M         ;move cursor to next character, setting
                    ;error flag if it is at the end
          ]         ;continue with next character,
                    ;unless error flag had been set.
```

Of course, the whole command could also have been written as:

```
A[@T<"AJA@T!" V0D@0I-M:AM]
```

And here's a much better way to do the same thing:

```
A[@T<"A[M][@T!" R]@T=0]
```

4.11 PERMANENT MACROS

There are some macros you'll want to use frequently. These can be made permanent. Think of this permanent macro facility as a way to add your own commands to the PMATE-86 command set. You can make a new version of PMATE-86 that incorporates your new commands (see the XD command). You can also define your own Instant commands by configuring PMATE-86 to execute a given permanent macro when you press a particular key (see Appendix B).

Permanent macros have a label that can be any character other than a digit.

.a Execute permanent macro a, where a is any character other than 0-9.

To add or remove a permanent macro, you must edit the permanent macro area. This area can be copied to or from the text buffer with the QMG and the QMC commands (see Q commands). This area must begin and end with an ^X. The ^X also separates different macros within the area. Immediately following each ^X is the character that labels the macro, followed by the macro itself. Here is a macro area containing macros # and C:

```
^X# Iyou have just executed macro #$
^XC 2QAEC^AA$^AB$
^X
```

Executing the command .# inserts "you have just executed macro #" into the text. The command .C behaves just like C, except it does not generate an error message if the string is not found.

It is possible for you to define a macro that PMATE-86 executes each time it is initialized. The first macro in the permanent macro area is executed as part of PMATE-86's initialization procedure if it is preceded by an ^I (Tab), rather than the usual ^Xx (where x is the name of the macro). This macro can even end in XH, causing it to generate a program that acts on a file and returns without displaying anything on the screen.

If you put ^S (rather than ^I) before the first macro, the macro is still executed initially, but the files specified in the command line following "PMATE" are not opened. The command line can then be referenced as a string argument by using ^A:. (For instance, I^A:$$ inserts the command line into the text buffer.) You must call the command line immediately if you need it, since it's not available after any file activity has taken place. This lets you make a customized version of PMATE-86 that processes commands given directly from the operating system command line.

BREAKPOINTS 4.12

PMATE-86 has a breakpoint and trace facility that helps you debug complex commands and macros.

? Stop executing the command. PMATE-86 is now in Trace mode and the cursor in the command area points to the next command to be executed. The current value of the numeric argument is displayed in the status line. Instant commands are active, and you can go into and out of Insert mode. If you press the Escape key, command execution resumes until the next ? command. If you press a key that is not an Instant command, PMATE-86 executes the next command and stays in Trace mode.

If you have trouble figuring out what is wrong with your macro, insert question marks at strategic places in the macro. You can use these question marks to see what has happened after partial execution of the command.

4.13 KEYBOARD INPUT

G

Get the following key from the keyboard. Pause during the execution of the command and update the display. The string argument following G is displayed as a prompt in the command display area. Instant commands are active. The command continues executing when you enter a character (other than an Instant command) from the keyboard. The ASCII value of this key is available by using @K in a numeric argument.

For example, gTYPE A KEY$@Ki$$ displays "TYPE A KEY" on the command line. The next character entered is inserted into the text buffer in front of the cursor.

0G

The string argument following G is displayed as a prompt in the command display area. The command continues executing without further keyboard input.

This command gives PMATE-86 I/O power. PMATE-86 can stop in the middle of an editing operation, and ask you how to proceed. Macros can expand upon the power of the G command — accepting either character strings (putting them in an available text buffer) or numbers (putting them in variables).

nQA Set the number of passed string arguments to n. (See macro description.)

QB Ring bell. This can tell you that a long command string has finished executing.

nQC Set the control shift character to the character represented by the ASCII value n. The shift character is ignored when input, but enters the next character as a control character. This command lets you enter text characters that would otherwise be interpreted as Instant commands.

nQD Delay for a time proportional to n. This can be used with L and QR to implement variable speed scrolling.

nQE Set type-out mode to n. (See XT.)

nQF Set the form feed character to that represented by the ASCII value n.

nQG Turn off garbage stacking if n equals 0. If n is non-zero or missing, turn on garbage stacking.

nQH Insert n spaces at the cursor. This is useful for operations such as centering. Since all spaces are inserted at once, this operation is much faster than n[I $].

nQI Set the input radix to numeric argument n. If n is missing, set the radix to decimal. Remember, if the old input radix is octal, 10QI does not set it to decimal. Since the 10 is interpreted in the old radix, the input radix remains octal.

nQJ Shift the text display up n lines (or down if n is negative) without changing the cursor location. This command shifts the display as far as possible without moving the cursor from its allowed screen positions.

nQK Set backup mode for files. If n is 0, .BAK files are not created from old input files; if n is non-zero or missing, .BAK files are created.

nQL Set number of lines that Instant commands ^U and ^J scroll.

QMG Insert contents of the permanent macro area into the text buffer just ahead of the cursor.

QMC Copy the entire text buffer to the permanent macro area. The previous contents of the macro area are lost. If you want to save them, do a QMG first. Then, add to or modify the text before copying it back.

nQNstrng$

Direct console I/O, similar to the G command. The specified string outputs directly to the console. If n is missing or non-zero, the string is output only after a key is pressed. The ASCII value of this key is then available by using @K in a numeric argument. Unlike G input, this is direct console input, without any pre-processing or Instant command translation.

nQO Set the output radix to n. If n is missing, the radix is set to decimal.

nQP Set input and output page size to n. If n is 0, pages are delimited by form feed characters, rather than breaking when a certain number of lines has been reached.

nQQ Shift the text display left n columns (or right if n is negative) without moving the cursor. This command shifts the display as far as possible without moving the cursor from its allowed screen positions.

nQR Redraw screen. The argument @K contains the value of any key struck, or 0 if none. Use this command to create interactive command strings (strings where PMATE-86 continues doing something and showing you the results until you tell it to stop).

nQS Set the uppercase/lowercase shift character to the character represented by the ASCII value n. This shift character is ignored when input, but shifts the case of the next character entered.

nQT Type the character represented by the ASCII value n on the listing device.

nQU Set PMATE-86 to automatic disk buffering mode if n is non-zero or missing. If n is 0, automatic disk buffering is disabled.

nQV Enable Tab-fill unless n is 0. When a character is inserted past the end of an existing line, PMATE-86 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.

nQX Move screen cursor to column n on the current line. Depending on the state of the free-space flag (see QY), the cursor may not be able to go past the last character in a line.

nQY If n is 0, set the free-space flag so that the screen cursor can 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 non-zero, reset the flag so that the cursor is restricted to existing text.

nQZ Don't allow cursor to move past column n. Use this when you want to control the width of your text — usually when you need clean output on a limited-width printer. When the cursor reaches the restricted column, it stops advancing and a warning tone is sounded. If n is missing, the default width of 250 columns is restored.

nQ! Set byte in memory whose address is held in variable 9 to n. This command lets PMATE-86 alter any byte in memory (and, of course, crash the system). Used with @@, a monitor could be constructed in macros. Other macros might change I/O driver parameters. However, for altering text, just move the cursor and use nR.

nQ – Sets flag to indicate whether numbers are displayed as signed or positive only. If n is 0, numbers are displayed as positive only; otherwise they are displayed as signed numbers. This affects the argument display (ARG) in the status line, as well as numbers

inserted in the text by the \\ command. If more than 32K of memory remains, you see ARG = −30536 in the display after you type the command @m$$ (in order to see how much memory remains). Then entering 0Q− gets you a more meaningful display.

Q# Exchange the tag and cursor.

nQ/ Set the indent to n. After a Return is entered in Overtype or Insert mode, PMATE-86 advances the cursor to column n. Set the free-space flag (see QY) to use this feature, as spaces (or 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 indent by one column, and −Q/ decrements it one column.

nQm Set user variable m to value n (m is a digit from 0 to 9). These ten variables are used by user-written I/O drivers. For instance, you can control whether hard copy output goes to a TTY console, or to a line printer.

4.15 INPUT, OUTPUT, AND DIRECTORY MAINTENANCE COMMANDS

All input and output commands begin with an X. This helps prevent accidental I/O, which can cause big problems.

DISK I/O

PMATE-86 has automatic, bidirectional disk buffering facilities. When in auto-buffer mode, files as large as 512K are edited without having to transfer pages explicitly between memory and the disk. Automatic disk buffering helps you edit files larger than available memory; the commands L, M, P, and W, and the cursor motion Instant commands don't stop at the end of memory. Instead, they scroll through the disk, reading in text as needed and writing out text from the other end. Auto-buffering is only in effect when editing an open file in buffer T.

If you don't use auto-buffer mode, files that are too large for memory must be broken into pages. Pages are divided by a character you define (usually a form feed), or they given a fixed number of lines (see the QP command).

nXA Append n pages from the input file to the edit buffer.

–nXA Bring n pages already written out to the output file back into the edit buffer.

nXW Write n pages from the beginning of the edit buffer to the output file, and then delete them from the buffer.

–nXW Write n pages from the end of the edit buffer back to the input file (actually to a file called PMATE.TMP).

nXR Replace n pages by appending n pages from the input file, and writing n pages to the output file.

–nXR Replace n pages by retrieving n pages from the output file and writing n pages back to the input file.

nXY Yank n pages from the input file. Each page overwrites the old one, without writing the old one to the output file.

This command is used for reviewing an existing file. Except in special circumstances, the file should be XKed when done.

XFfile1 If file exists, open it as the input file and open file.$$$ as the output file. If the file does not exist, create it and make it the output file. The filename can be preceded by a drive specifier.

XFfile1 file2

Open file1 as input and file2 as output. file1 should already exist on the disk (if it doesn't, file1 is opened as the output file) and file2 shouldn't exist (if it does, an error message appears). Both file1 and file2 can be preceded by drive specifiers.

XE End of editing pass. Write the text buffer to the output file. Read in the remainder of the input file and write it to the output file. Close the input and output files and clear the text buffer.

If the output file is the same as the input file (with a .$$$ extension), rename the input file to file.BAK, delete the old backup, and give the output file the name of the old input file.

XEfile — End of editing pass, as above — but output file is renamed "file" and the original input file is undisturbed.

XJ — Start a new editing pass. Equivalent to an XE and then an XF of the original file name. Used for editing a page already written out with XW or XR. Don't go too long without an XJ, even on files that fit entirely in memory. This ensures that your editing work is saved if there is a power failure, or catastrophic error.

XJfile — Equivalent to XEfile, followed by reopening the new file.

XC — Close input and output files as they are. Neither the contents of the text buffer nor the rest of the input file is written to the output file. File renaming does not take place, even if the output file is temporary with extension .$$$).

XK — If in buffer T, delete the output file and clear the text buffer. If in another buffer, clear buffer without affecting the input and output files.

XH — Return to the operating system. This is the usual way to exit from the editor.

XDfile — Duplicate PMATE-86; write it as it now exists to file.COM. This output file can later be renamed PMATE.COM.

nXIfile — Auxiliary input. Read the first n pages of file into edit buffer at cursor location, even if another file is open as the input file. If n is missing, read in the entire file. If the entire file is not read in, the remainder can be read in later.

nXI — Input the next n pages from the input file last defined by the XIfile command. If n is missing, input the entire remainder of the file.

nXOfile — Create file and write n lines of text (beginning at the cursor) out to the new file. If n is missing, write out the entire edit buffer.

DIRECTORY MAINTENANCE

XSb — Change the disk to b. PMATE-86 does not respond to this command while input and output files are defined.

XLfile — Like operating system DIR command. Lists all files which match the file name in the command (* and ? can be included in the file specification). If file is missing, the entire directory is listed.

The directory listing is placed in the text buffer at the cursor location. This lets you print the directory or manipulate it as you would regular text. However, if desired text is already in the text buffer, it may be necessary to delete the directory text. Alternately, change the current text buffer before giving the XL command.

XXfile — Delete file from the disk. Ambiguous file names (containing * and ?) are not allowed.

OTHER COMMANDS

nXT — Starting at the cursor, type n lines on the listing device. If n is missing, type out the entire edit file. There are 3 type-out modes (set by the QE command):

- Mode 0 prints text almost exactly as it is displayed. Format lines are printed, escapes type out as $, and other control characters are printed as an up-arrow followed by an uppercase letter. Use this mode for printing macros and for draft output.
- Mode 1 (the default mode) is used to print programs or text on a regular printer. Tabs are expanded to spaces. Format lines are not printed, but affect the margins and Tab stop settings. Other control characters are sent through to the printer.
- Mode 2 is used with intelligent printers that do their own formatting. Returns are sent only at the end of

a paragraph; Tabs are not expanded to spaces; and all control sequences are passed on to the printer.

Even in auto-buffer mode, nXT types only lines currently resident in memory. However, TnL#XT types out n lines, reading them from the disk as necessary. In addition, XT, without an argument, types out the entire edit file.

5

MACRO EXAMPLES AND IDEAS

This chapter 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 supposed 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 the last chapter 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 a key.

ADDING OR DELETING COMMENTS 5.1

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-86, 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-86 macros themselves:

```
[S;$ -M -S^N^I$ 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 (^N^I 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.

5.2 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. It's possible for you to 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
[
        S^AA$
        GType space to replace$
        @K=32[-C^AA$^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.

TEXT OUTPUT PROCESSING 5.3

By itself, PMATE-86 does not perform many print functions often associated with word processors. However, you can use PMATE-86 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 macro above), 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.

5.4 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. In order to correct mistakes on entry, a backspace deletes the last character entered.


```
B9K                    ;delete old contents of buffer 9
[                      ;start iteration
G^A@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-86 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 all done. 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-86.

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.

5.5 TWO PRINT MACROS

This simple macro lets you type directly on your printer, using the keyboard as if it were a typewriter:

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

The third line implements an automatic line feed. If the macro finds a Return, it sends a line feed also. Any other character is sent as is.

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 file name to buffer 0 —
                    ;will try to find file names earlier
                    ;alphabetically.
[                   ;this loop finds earliest file name
@H^A@0$<0[BC][L]    ;compare next file name to earliest
                    ;already found — if this one is
                    ;earlier, copy it to buffer 0,
                    ;otherwise, advance to next
@T = 0]             ;iterate until end of directory list
A                   ;back to top of directory list
S^A@0$              ;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 5.6

Here (without comment) are the macros that PMATE-86 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]`

A

COMMAND SUMMARY

The following Instant commands are not entered into the command or text buffers; instead, they are executed immediately.

CURSOR MOTION A.1

^A	Move to the beginning of the text buffer. If cursor is already there, move to the end.
^G	Move left one character.
^H	Backspace and erase character left of cursor.
^Y	Move up one line.
^B	Move down one line.
^U	Move up multiple lines.
^J	Move down multiple lines.
^O	Move left one word.
^P	Move right one word.
^F^M	Move to beginning of line.

SCROLLING A.2

^F^G	Scroll left one column.
^F^H	Scroll right one column.
^F^Y	Scroll up one line.
^F^B	Scroll down one line.

A.3 DELETE

^D	Delete the character at the cursor.
^K	Kill the line beginning at the cursor.
^W	Delete one word beginning at cursor.
^Q	Delete one word backwards from cursor.

A.4 TEXT MOVEMENT AND RECOVERY

^T	Tag the current cursor location.
^E	Move block between tag and cursor to special buffer.
^Z	Move contents of special buffer to cursor location.
^R	Retrieve most recently deleted item from garbage stack.

A.5 MODE

^X	Go to Command mode.
^V	Go to Overtype mode.
^N	Go to Insert mode.

A.6 AUTO-INDENT

^F^I	Set auto-indent to current column.
^F^P	Increment auto-indent four columns.
^F^O	Decrement auto-indent four columns.

OTHER A.7

^S	Repeat next keystroke four times (or number immediately following).
^L	Insert line.
^C	Cancel operation in progress and return to Command mode.
^F^T	Exchange tag and cursor.
^F^F	Redraw and reformat display.
^_	Edit the command string.
^F^S	Shift default case.
^F^C	Change case of character at cursor.

These characters are not really Instant commands, but they do have special meanings:

$	The "escape" key separates commands in Command mode. Two consecutive escapes execute the command shown as $ in this text and on the screen.
TAB	The tab character in text positions the following character at the next Tab stop.
DEL	Delete the character at cursor position.
^	If entered once, the following character entered is a control character. If entered twice, an up-arrow is entered.

A

COMMANDS WHILE IN COMMAND MODE A.8

Now come the real commands. When in Command mode, these are entered into the command buffer and then executed.

CURSOR MOVEMENT

L	Move forward one line.
+nL	Move forward n lines.

−nL	Move backward n lines.
M	Move forward one character.
nM	Move forward n characters.
−nM	Move backward n characters.
W	Move forward one word.
nW	Move forward n words.
−nW	Move backward n words.
P	Move forward one paragraph.
nP	Move forward n paragraphs.
−nP	Move backward n paragraphs.
A	Move to beginning of text resident in memory.
UA	Move to beginning of file.
Z	Move to end of text resident in memory.
UZ	Move to end of file.

DELETING CHARACTERS

D	Delete character at cursor.
nD	Delete n characters, from cursor forward.
−nD	Delete n characters, from cursor backward.
K	Delete line containing cursor.
nK	Delete n lines, from cursor forward.
−nK	Delete n lines, from cursor backward.

INSERTING CHARACTERS INTO BUFFER

Istring	Insert "string" immediately after cursor.
nI	Insert character with ASCII code n.
Rstring	Overwrite text with "string".
nR	Overwrite character at cursor with ASCII code n.
n\\	Insert number n into the text.

SEARCH AND CHANGE

Sstring	Search forward for next occurrence of "string", confining search to memory.
nSstring	Search forward for next occurrence of "string", confining search to n lines.
– Sstring	Search backward for next occurrence of "string", confining search to memory.
– nSstring	Search backward for next occurrence of "string", confining search to n lines.
USstring	Search forward through the entire file for next occurrence of "string".
– USstring	Search backward through the entire file for next occurrence of "string".
Cstrng1$strng2	Change next occurrence of "strng1" to "strng2".
nC, – C, – nC	Search for "strng1" as in equivalent S command, then change it to "strng2".
UC, – UC	Search for "strng1" as in equivalent US command, then change it to "strng2".

ITERATION AND CONTROL

IJlabel	Jump if I is true to "label".
I[. .]	Execute expression in brackets only if I is true.
I[. .][. .]	Execute expression in first brackets if I is true; otherwise execute expression in second set of brackets.
n[. .]	Iterate expression in brackets n times.
[. . I]	Iterate until I is true.
I^	Proceed to next iteration if I is true.
I_	Exit enclosing iteration loop if I is true.

MISCELLANEOUS COMMANDS

E	Suppress error messages.
nF	Enter Format mode, setting line width to n.
F	Toggle in and out of Format mode.
Gstrng	Get key from keyboard, giving user prompt "strng".
0Gstrng	Give user prompt "strng", without waiting for key.
N	Go into Insert mode.
T	Tag current cursor position.
.b	Execute macro b.
I%	Return early from macro if I is true.
:x	Label this position in command with character x.
;	Comment — ignore all characters until end of line.
?	Enter Trace mode.

STORING INTERMEDIATE RESULTS

There are 10 numeric variables and a number stack for storing intermediate results, along with commands to set them.

A

nVi	Set variable i to value n.
VAi	Increment variable i.
nVAi	Add n to variable i.
n,	Push n on number stack.

Q COMMANDS

The following Q commands perform miscellaneous functions, usually setting some internal parameter:

nQA	Set the number of passed string arguments in a macro call.
QB	Ring the bell.
nQC	Set control shift character to ASCII n.
nQD	Delay for a time proportional to n.
nQE	Set type-out mode to n.
nQF	Set page separator character to ASCII n.

nQG	Enable garbage stacking unless n is zero.
nQH	Insert n spaces at cursor position.
nQI	Set input radix to n.
nQJ	Scroll display up n lines.
nQK	Create .BAK files unless n is 0.
nQL	Set number of lines for ^U and ^J commands to scroll.
QMC	Copy to permanent macro area.
QMG	Get contents of permanent macro area.
nQNstrng	Send "strng" directly to console, if n is non-zero, wait for key from console, and return as @K.
nQO	Set output radix to n.
nQP	Set page size to n.
nQR	Redraw screen — return any key struck as @K.
nQS	Set lowercase shift character to ASCII n.
nQT	Type the character represented by ASCII n.
nQU	Set automatic disk buffering unless n is 0.
nQV	Enable tab-fill unless n is 0.
nQX	Move screen cursor to column n.
nQY	Allow cursor motion in free space if n equals 0.
nQZ	Don't allow cursor to move past column n.
nQ!	Store n in memory at location pointed to by variable 9.
nQ–	Display numbers as positive only if n equals 0.
nQ/	Set auto-indent to column n.
Q#	Exchange tag and cursor.
nQm	Set user variable m (0-9) to n. These 10 user variables are available to user-written IO drivers.

X COMMANDS

The following X commands generally perform disk I/O. They begin with an X in order to make them hard to execute accidentally, as they cause major upheaval.

XA	Append next page of input file.
nXA	Append next n pages of input file.
–XA	Retrieve last page written to output file.
–nXA	Retrieve last n pages written to output file.

XW	Write next page to output file.
nXW	Write next n pages to output file.
−XW	Write page back to (temporary) input file.
−nXW	Write n pages back to (temporary) input file.
XR	Write one page to output file, read one from input file.
nXR	Do this n times.
−XR	Write one page back to (temporary) input file, read one back from output file.
−nXR	Do this n times.
XE	End of editing.Write out all remaining text from buffer and input file.
XJ	Do XE, then reopen file.
XF	Define new input and output files.
XK	Delete output file and scratch edit buffer.
XC	Close input and output files as they are.
XH	Return to operating system.
XIfile	Input entire file "file".
nXIfile	Input n pages of "file".
nXI	Input n pages of last named auxiliary input file.
XOfile	Output entire edit buffer to "file".
nXOfile	Output n lines, beginning at cursor, to "file".
XDfile	Create new version of PMATE-86, including any new changes or permanent macros. New version is called file.COM.
XSa	Log in disk drive 'a' ('a' is A, B, C, etc.).
XT	Type entire text buffer on printer.
nXT	Type n lines, beginning at cursor.
XL	List disk directory at cursor.
XLfile	List just those files in directory which match "file".
XXfile	Delete "file" from disk.

A

B COMMANDS

The following B commands act on buffers 0-9 or the text buffer T. Buffer 0 is assumed, unless a buffer number is placed between the two characters of the command.

BK	Kill the entire contents of the specified buffer.
BG	Get the contents of the specified buffer.

nBC	Copy n lines to the specified buffer.
nBD	Append n lines to the specified buffer.
nBM	Move n lines to the specified buffer.
nBN	Append move n lines to the specified buffer.
BE	Edit the specified buffer.

TAB STOP COMMANDS

nYD	Delete Tab stop at position n.
nYS	Set a Tab stop at position n.
YK	Kill all Tab stops.
nYE	Set a Tab stop every n spaces.
YF	Fill Tabs with appropriate number of spaces.
YR	Replace spaces with Tabs where possible.
nYI	Set indent at column n.

NUMERIC ARGUMENTS

Numeric arguments can be complex expressions, involving up to 15 levels of parentheses, and the following operations:

+	Addition.
–	Subtraction.
*	Multiplication.
/	Division.
!	Logical OR.
&	Logical AND.
'	Logical complement (Not).
<	Less than.
>	Greater than.
=	Equal.

In addition, the following expressions can be used with the above operations to form numeric arguments.

"a	The ASCII value of character a.
@i	The value of numeric variable i.
@A	The numeric argument when macro was called.
@B	Current edit buffer (0 for T, 1 for buffer 0, etc.).
@C	The character number.
@D	The number of lines scrolled by ^U and ^J (set by QL).
@E	The value of the error flag.
@Ffile	-1 if "file" exists on the current directory; 0 if it doesn't.
@G	The length of the last referenced string.
@Hstrng	Compare "strng" to text at cursor. Return 0 if equal; otherwise 1 or -1, depending upon which string is greater.
@I	The current input page.
@J	The number of lines in the text display.
@K	The ASCII value of the key struck after a G or QR command.
@L	The line number.
@M	The amount of memory remaining.
@O	The current output page.
@P	The absolute memory address pointed to by the cursor.
@Q	The column of the previous Tab stop.
@R	The remainder of the last division.
@S	Pop the number stack — get value of top.
@T	The ASCII value of the character pointed to by the cursor.
@U	-1 if auto-buffering is enabled; 0 otherwise.
@V	The mode.
@W	The right margin.
@X	The column.
@Y	The left margin.
@Z	The column of the next Tab stop.
@@	The byte pointed to by variable 9.
@/	The current auto-indent column.
#	Move cursor to tagged position, and get difference between tagged position and current position as argument. Can be used with any character- or line-oriented command to operate on a block of text.

B

CUSTOMIZATION GUIDE

GENERATING A CUSTOM CONFIGURATION FILE B.1

CONFIG.CNF is a file that contains a series of questions and answers (in ASCII). You can use PMATE-86 to create a custom version of this file. To do so, you need to answer a series of configuration questions; each requires a yes/no answer, a letter, or a series of numbers. All answers follow three stars (***). Numbers can be in decimal or hex. Hex numbers are identified by the ending "H". If more than one number is required, separate them by spaces.

You can give your custom version of CONFIG.CNF a new file name, 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-86, obtaining information from the file MYCONFIG.CNF, type:

```
CONPMATE MYCONFIG
```


Upon completion, you must save this custom version of PMATE-86 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 wish redrawn in foreground?

This sets the number of lines to be redrawn on the screen before PMATE-86 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-86 has the time.) The smaller this number, the faster PMATE-86'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-86 screen redraws proceed in one of two ways. The traditional method is to start at the top, and work down. PMATE-86 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-86 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-86 allows lines of up to 250 characters in length. Since displays rarely show more than 80 of those, PMATE-86 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 wish .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-86 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-86 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 room to edit text.

► Size of permanent macro area.

Enter the amount of memory (in bytes) you want to reserve for permanent macros. PMATE-86 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-86 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.

B.2 CUSTOMIZING THE KEYBOARD

PMATE-86 lets you determine the keystroke required to execute Instant commands. To better suit your preferences and hardware, CONPMATE can create a version of PMATE-86 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 (Alt) 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-86 instant command set.

The PMATE-86 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 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-86 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-86 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 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
```

EFONT

First VICTOR printing February, 1983.

ISBN 0-88182-004-0 Printed in U.S.A.

CONTENTS

CHAPTERS

1

INTRODUCTION

If you take a close look at your screen, you'll see that each letter or character on the screen is made up of many small dots. The arrangement of these dots gives each character its distinctive shape. If you could change the arrangements of the dots, you could create letters and characters that aren't found in any of the character sets supplied with your operating system. The EFONT font editor lets you create these custom characters.

With EFONT, you can change the appearance of the letters and characters that appear on the screen or you can design your own character sets from scratch. The characters that you create can be loaded into your operating system and appear on-screen as you work with an application or language program. (EFONT is especially useful with the graphics package, GRAFIX.) In addition, your edited character set can be printed if you have a dot-matrix printer.

EFONT CAPABILITIES 1.1

The EFONT font editor is a menu-driven program. This section gives a brief description of the various menu options.

- Cell mode: Lets you make changes to the currently displayed character. The character can be rotated, mirrored, put into reverse video, or erased.
- Copy: Lets you move a range of characters from one place to another within a character set, or from one character set to another.
- Line mode: Controls the insertion and deletion of horizontal and vertical lines. Using this mode, you can change the horizontal size of a character or change that character's position within the character matrix display.

- Disk mode: Lets you load and store your new or modified character sets onto a diskette. Disk mode is also used to edit the header. (The header contains important information used by the operating system configuration program.)
- Width mode: Lets you construct proportionally sized characters.
- Exit: Takes you from the DISK or other editing menu back to the Main menu. If Exit is used at the Main menu, you will return to the operating system.

1.2 USING THIS MANUAL

This manual gives you all the information you need to start using EFONT. The first portion of the manual explains the various EFONT screen displays and menus, and describes the functions which the program can perform. The second part takes you through a sample font editing session. (You may want to run through that sample session before you read the rest of the manual.)

Before running EFONT, make sure that your operating system includes a standard keyboard and the International character set. (If you want to load your edited character set onto a program or application diskette, you'll also need to have a copy of the Operating System Configuration program.) Programmers should note that the EFONT program is written in assembly language.

2

THE WORK SCREEN AND THE MAIN MENU

To start EFONT, boot your operating system. When the A> prompt appears on your screen, type EFONT and then enter a Return. This display appears:

This display is the "work screen"—the place where you edit an existing character set or create a new one. The work screen has the following parts:

- Character cell: An enlarged view of the character being edited. It contains one of the two cursors present in the work screen. The character is displayed on a grid with 16 vertical columns and 16 horizontal rows (for 256 possible positions). A period marks each spot where a row and a column intersect. If you put the cursor on one of these periods, you can enter a dot that becomes part of the character you're editing. You can also remove a dot.

- Character set table: Consists of two parts: the character set display and the attribute field.

  The character set display shows the character set you're editing (character set 0, unless you change it). Each time you change the character set, the work screen and the characters in the character set display also change. The character set display also contains the other cursor present in the work screen. The position of this cursor determines which character is displayed in the character cell and in the attribute field.

  The attribute field shows the character pointed to by the cursor in the character set display. That character is shown in several ways: reverse video, underlined, half-intensity, and all possible combinations of the three. Two additional displays of the character being edited are at the right side of the attribute field: one in normal video and the other in reverse video. These displays are larger than the others, and are used to view characters larger than the normal text mode. (Characters up to 16 dots wide by 16 dots high can be displayed; the normal size is 10 by 16 dots.)

- Status area: Consists of the five lines of information you see at the center right of your screen. Each line tells you something about the current edit.

  The first line is the file name of the character set being edited; the second is the number of that character set; the third line shows the edit mode; the fourth line tells the status of the copy buffer; and the last line tells you the width of the character under edit (useful when making proportionally spaced character sets).

- The Main menu: The row of numbered functions at the bottom of the screen. The Main menu is the first one you see when you begin a session with EFONT; each of the other menus returns you to the Main menu.

  Each function on the Main menu is controlled by a like-numbered key at the top of your keyboard (the function keys). Each function displays its own menu after you press its function key.

3

FUNCTIONS ACCESSED THROUGH THE MAIN MENU

EXIT 3.1

The EXIT key is at the left of the Main menu. If you press it, the screen clears and displays two new function keys:

```
NO    YES
```

If you press NO, the Main menu reappears. If you press YES, you exit EFONT and return to the operating system.

CELL MODE 3.2

The CELL key is the second function key on the Main menu. It lets you alter the character displayed in the character cell. When you press the CELL key, this menu appears at the bottom of the screen:

```
MENU  REVERSE  MIRROR  ROTATE  CLEAR        HELP
```

The MENU key is an exit key; it returns you to the Main menu. (This is true for the MENU keys on all other menus as well.) The last function key on this (and any other) menu is a HELP key. When you press it, the work screen disappears to be replaced by a screen full of directions on how to use the function keys in the current menu.

The REVERSE key reverses the display in the character cell—all of the "on" dots turn off, and all of the "off" dots turn on.

The MIRROR key rotates the character in the character cell on its vertical axis, producing a "mirror image" of that character.

If you press the ROTATE key, the character in the character cell rotates 90 degrees clockwise. The ROTATE function can be used to create sideways character sets.

The CLEAR key erases the character in the character cell.

3.3 COPY

The third function key on the Main menu is the COPY key. It lets you move characters from one location to another. You can use COPY to move characters from one character set to another, for example, or within a character set to exchange the positions of the upper- and lowercase letters.

When you press COPY, the Main menu is replaced by the Copy menu:

```
MENU    RANGE    COPY                              HELP
```

As with the last menu, the MENU key returns you to the Main menu.

The RANGE key lets you identify a group of characters (range) that you want to copy to another location. To do this, you must define the starting character of the range and limit the range with an ending character.

COPY moves a range of characters from one place to another. To use COPY:

1. Define the range of characters to be moved. Mark the first character of the range with the RANGE key. Then, mark the end of the range by moving the character set cursor to the character immediately after the last character in the range. Then, press RANGE again. The range you've marked is highlighted with bright video. Make sure that it includes all the characters you want to move; if it doesn't, redefine the range.

2. After you define the range, the fourth line in the status area of the work screen tells you that the copy buffer is full. (The copy buffer holds up to 256 characters.) An error message appears if you try to move a larger range of characters.

3. When the range is correctly defined, move the character set cursor to the place to which you want to move the range of characters. Then, press COPY to copy your range to the new location.

The remaining key on the Copy menu displays HELP information.

LINE MODE 3.4

The LINE key controls the insertion and deletion of horizontal or vertical lines. It lets you change the horizontal size of a character or that character's position within the character cell. When you press the LINE key, the Line menu appears:

```
MENU  INSERT H  DELETE H  INSERT V  DELETE V  HELP
```

As with other menus, MENU returns you to the Main menu.

The INSERT H key scrolls the lines between the cursor location and the bottom of the character cell matrix down one position. At the same time, a row of periods is inserted at the cursor location.

If you press the DELETE H key, it deletes the row of dots at the cursor position and scrolls up all the rows below the cursor position. At the same time, a row of periods is added at the bottom of the character cell.

If you press INSERT V, it inserts a column of periods at the cursor position. All columns to the right of the cursor position are scrolled one position to the right. The rightmost column is erased.

The DELETE V key erases the column of periods including the cursor position and all columns to the right of the cursor scroll left one position. At the same time, a column of periods is added at the right edge of the cell.

3.5 DISK MODE

The fifth key on the Main menu is the DISK key. It lets you load and store any character set you create onto diskette. The DISK key also lets you edit the header. When you press the DISK key, the Main menu is replaced by the Disk menu.

```
MENU    LOAD    SAVE    HEADER              HELP
```

As before, the MENU key returns you to the Main menu when you're finished with DISK operations.

The LOAD key loads any existing character into the active character set shown on the work screen. To load a character set:

1. Return to the main menu and choose a number for the character set you want to load. (To do this, press the plus (+) key or minus (−) key until the desired number appears in the status area of the work screen. These numbers range from 0 to 7; in most cases, 2 through 7 are empty.)

WARNING: Don't select character set 0 unless you actually want to replace the system character set.

2. Press the DISK function key to call the list of file names of the available character fonts.

3. Use the cursor movement keys (described in Chapter 4) to move the highlight to the name of the character set you want to load. Then, press the LOAD key. The new character set is loaded, and you return to the Main menu.

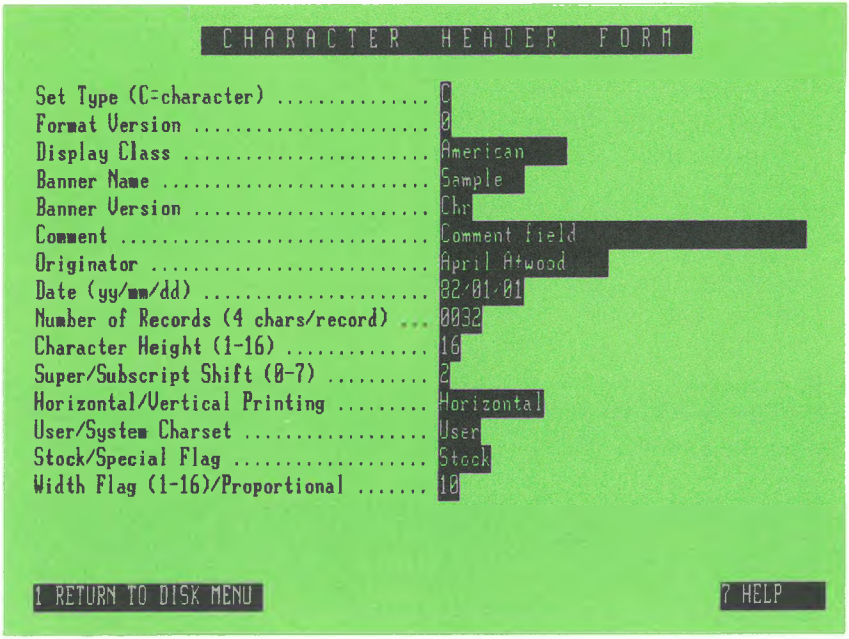
The SAVE key is used to save any new or edited character set on disk. If you press SAVE, the screen clears, displays the file name of the character set being edited, and then replaces the Disk menu with the following:


```
DISK    NAME OK                                    HELP
```

If you want to change the file name of your character set, use the cursor keys to position the cursor in the file name field and type new characters. (The BACKSPACE key erases characters preceding the cursor. The DEL key deletes characters at the cursor.) When the name is correct, press the NAME OK key to save the character set on the disk. When the set is saved, you return to the Disk menu.

If you don't want to save your character set, press the DISK key to return to the Disk menu.

The HEADER key allows you to edit the header block. (This block is saved at the same time you save a character set file. It contains important information for the system configuration program and the GRAFIX graphics package.) Pressing the HEADER key clears the current screen display and replaces it with the header form and the Header menu.



There are three types of fields in the header form:

- The first nine are ASCII fields. They are changed in the same way as the file name fields in the DISK SAVE function described above.
- Three form increment fields contain numeric data. These are: Character Height, Super/Subscript Shift, and Width Flag. You can change the numbers in these fields by pressing any key.
- There are three toggle fields: Horizontal/Vertical Printing, User/System Charset, and Stock/Special Flag. When the cursor is in one of these fields, you can choose an option by pressing any key.

Use the up- and down-cursor keys if you want to move from one header block field to another. The cursor-left and cursor-right keys move from one character to another within an ASCII field.

Press RETURN TO DISK when all of the header fields are correct. The HELP key displays a listing of the fields and their types.

WIDTH MODE 3.6

The WIDTH key is used when you construct proportionally sized characters. The width of a character dictates how close the next character in a word can appear. If you press the WIDTH key, a new menu appears:

The MENU and HELP keys work as described earlier. The AUTO function key tells EFONT to automatically set the width of the character under edit. If you press MANUAL, the width of the character under edit is increased. (The character width is listed in the fifth line in the status area of the screen. If you want proportional characters, be sure to set the width field in the header form to proportional.)

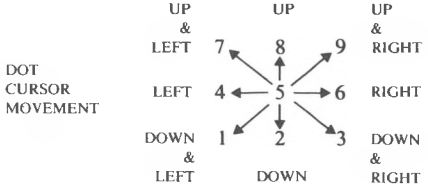
4

EDITING KEYS

CURSOR KEYS 4.1

The EFONT program has several cursor keys that operate in most edit modes.

- The cursor arrow keys (→ , ← , ↑ , ↓) move the character set cursor to a new character.
- The number keys on the numeric pad move the cursor within the character cell. (The HELP key on the Main menu shows you an example of how the number keys move this cursor.) The number keys are positioned around the 5 key in the same way that they move the cursor.



- The RETURN key (and the ENTER key on the 10-key pad) are toggles for the dots in the character cell. Pressing one of these keys either makes a dot appear at a blank location or erases an existing dot.
- The BACKSPACE key restores the character cell to its original condition. This is a big help when you want to erase your changes and start the edit process over with the original character.

DOT EDIT MODES 4.2

The Dot Edit mode is displayed in the third line of the status area. You change the mode by using the Equals (=) key on the 10-key pad. The

character set under edit is changed by using the plus (+) key or the minus (−) key on the 10-key pad.

There are four edit modes.

- Position mode: The safest of the four modes because it is non-destructive—it lets you move the dot cursor around the character cell without changing any dots. The cursor keys described above function normally.
- Reset mode: Dots are turned off when the cursor passes over them, regardless of their current status.
- Toggle mode: Dots beneath the cursor switch to the opposite status when the cursor is moved: "On" dots are turned off; "off" dots are turned on.
- Set mode: Dots are turned on when the cursor passes over them, regardless of their current status.

5

SAMPLE FONT EDITING SESSION

Now that we've seen the various menus, Function keys, and editing keys that EFONT uses, we'll use them to create a character set. Our example will go through the process step by step. If you're unsure of any step, check the previous sections for details.

1. After booting your operating system, type "EFONT" to load the font editing program.

2. When the work screen appears, press DISK at the Main menu to call the directory of character set file names.

3. Use the cursor arrow keys to move the reverse video highlight over NORMAL.CHR.

4. At the Disk menu, press LOAD to load the Normal character set into EFONT. (The letters on your screen display remain the same, since the Normal character set is also used by the SAMPLE.CHR set automatically loaded when the work screen appears.)

5. As practice, we're going to copy character set 0 of the Normal font, rather than editing the original. Press MENU to return to the Main menu, then press COPY. The Copy menu appears at the bottom of your screen.

6. The character set cursor should be at the first character of character set 0. Press the RANGE key to set the beginning of the range at the first character.

7. Using the cursor arrow key, move the cursor over ALL the characters in the Normal character set. When you reach the last character, press the right-arrow one more time. This puts the cursor on the first character in character set 1.

8. Press RANGE again. This defines the range of characters that we'll copy. If you move the character set cursor back one space, you'll see that all of character set 0 is highlighted in bright video. The copy buffer line in the status area now reads "COPY BUFFER FULL."

9. Using the cursor arrow keys again, move the cursor to the upper left character in character set 1.

10. Press the plus (+) key on the 10-key pad. Character set 2 is displayed. (This set is blank.)

11. Press COPY. The character range we defined earlier (all of character set 0) appears in the character set display. Then, press MENU to return to the Main menu.

12. Use the cursor arrow keys to move the character set cursor to the uppercase A. The uppercase A appears in the character dot matrix.

13. Using the cursor movement keys and the ENTER key on the 10-key pad, add or delete dots until you're satisfied with the new look of the uppercase A. When you have finished editing the character, go on to uppercase B, and then to each uppercase letter in the character set display.

14. When you have edited all of the uppercase characters, press SAVE at the Disk menu to save your edited character set on the EFONT diskette.

15. Now, EFONT asks you to verify or change the name of your edited character set. Use the cursor arrow keys to move the cursor to the file name field, and enter the name NEWONE.CHR. Then, press NAME OK to record the file name on the diskette.

16. If you press DISK, you can see that the file name NEWONE.CHR is now in the directory.

Now that you have edited and saved a character set, you still need to make it available for use with your system and application diskettes. To do this, you need an Operating System Configuration diskette (available from your dealer). If you have the diskette, follow these instructions:

1. With the EFONT diskette in drive A, insert the Operating System Configuration diskette into drive B.

2. Using the operating system's copy command, copy NEWONE.CHR from drive A to drive B.

3. Remove the EFONT diskette, and move the System Configuration diskette into drive A.

4. Re-boot your operating system. When the first System Configuration menu appears, select GENERATE A NEW OPERATING SYSTEM and press the Return key.

5. Select the appropriate keyboard at the Keyboard table, and press the Return key.

6. Answer Yes when the program asks if you want a second character set.

7. When the next display appears, choose NEWONE as your second character set. Then, press the Return key.

8. Select the other elements of your operating system as you are asked for them.

9. When the "Current Configuration" display appears, accept the configuration as listed and press the Return key.

10. When the next display appears, select USER ENTERED FILENAME and press Return.

11. Enter NEWSYS and press the Return key.

12. Answer Yes when the program asks if you're sure you want to write your operating system to NEWSYS. Then, press the Return key.

13. Insert a program or application diskette into drive B. When the BOOTCOPY program prompt appears on your screen, copy your operating system onto the diskette in drive B.

14. Your edited character set is now the second character set in your operating system; however, one more step is needed before you can use that new character set. When you load your new operating system into your computer, press the Shift key and enter an Alt-N while at operating system level. This loads your edited character set into the operating system. If you want to use the other character set in the operating system, enter an Alt-O without pressing the Shift key.

KEYGEN

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

First VICTOR printing February, 1983.

ISBN 0-88182-005-9 Printed in U.S.A.

CONTENTS

EXHIBITS

CHAPTERS

1

INTRODUCTION

KEYGEN lets you modify the keyboard of your computer by specifying the character code generated by a particular key. In this way, you can customize the keyboard to suit your particular needs.

The keyboard of your computer is a "soft" keyboard—the code generated by pressing a key is under software control. The keyboard "table" (a disk file with the .KB extension) defines the codes generated by each key. These "ASCII codes" are the computer's internal representation of characters. KEYGEN lets you create your own keyboard table, giving each key the character code of your choice.

Each key generates up to three different sets of codes—one set each for unshifted, shifted, and alternate modes. (Control mode is also available as a function.) Each key also has two key attributes and two mode attributes. The key attributes indicate whether the key is affected by Caps Lock or Shift Lock. The mode attributes determine whether the generated codes are automatically repeated when the key is held down (Repeat) or if they are sent directly to the console, bypassing the operating system and the application program (Local).

Character sets are also under software control. The character set defines the shapes of symbols displayed on the screen. Existing character sets can be selected when configuring the keyboard with KEYGEN, or you can create your own character sets with the EFONT utility. This gives you complete control over the keyboard and the character set you'll be using.

When used with MODCON and SYSELECT, a keyboard table defined using KEYGEN can be installed into the operating system. MODCON lets you choose the new keyboard table before entering an application program. It also lets you save the old keyboard table for restoration after you've finished using the application program. SYSELECT configures your operating system to contain the keyboard table defined by KEYGEN.

The following restrictions apply when using KEYGEN:

1. The computer must be connected to a printer when you issue print commands.

2. The printer must support the character set selected.

3. Most applications programs use certain codes to generate particular functions. When designing a keyboard table, make sure that the keys will function properly with all desired applications requirements.

4. KEYGEN makes new keyboard tables by altering existing ones. To use KEYGEN, at least one keyboard file must already exist (keyboard files have the .KB extension).

5. You must select a character set when you define the new keyboard table. Character sets are obtained by:

   - Selecting an existing character set (character sets have the .CHR extension).
   - Modifying any available character set file using the character font editor, EFONT.

6. The files KEYGEN.EXE and KEYGEN.DAT must be on the same disk.

2

FUNCTIONAL DESCRIPTION

This chapter gives a brief overview of the operation of KEYGEN. More detailed information is available in the next chapter.

FILE OPERATIONS 2.1

1. Load a character set file into memory for use in configuring a keyboard (Menu 1—Character Set File Selection).

   Names of existing character set files are displayed on the screen. You can move the cursor over the file names (using the cursor arrow keys) until it is at the desired file. Then, press LOAD to load the file into memory. Menu 1 appears.

2. Load a keyboard table file into memory for modification (Menu 1—File Selection).

   After a character set file is chosen, the names of existing keyboard files are displayed on the screen. You can move the cursor over the file names until it is positioned at the desired file. Then, press LOAD to load the file into memory. Menu 2 appears.

3. Change keyboard file name (Menu 2—Modify Header Information and Keyboard File Name).

   You may want to save your reconfigured keyboard in a file other than the one that was loaded. The name of the loaded file is displayed in Menu 2, along with header information. The file name is modified by placing the cursor at the "File name" heading line and typing in the new name. You edit by using the destructive backspace. Pressing ENTER enters the new file name onto the disk when the file is saved.

4. Save a modified table file on diskette (Menu 2—Modify Header Information and Keyboard File Name).

   Pressing SAVE at Menu 2 saves the reconfigured keyboard file on diskette. The program uses the name of the file that was originally loaded, unless you supply a different name. If the diskette is full, you must use the original file name and overwrite the original file with the newly configured file.

2.2 TABLE OPERATIONS

1. Maintain header in keyboard table (Menu 2—Select Header Information and Keyboard File Name).

   All header information is displayed on the screen. You select the data to be changed by using the cursor arrow keys to move the cursor to the desired line. New information is typed over the existing line with the destructive backspace. Hitting the Return key erases the old header information and enters the new information into the table.

2. Position to a key (Menu 2—Select Header Information and Keyboard File Name).

   You can move the cursor up, down, left, and right over the keyboard picture with the arrow keys. When the cursor is at a particular key, KEYGEN can access the keyboard table information that corresponds to that key. The value of that key (for all three modes) is displayed on the screen below the keyboard picture.

3. Position to a mode (Menu 2—CH KEYS; Menu 3—UNSHIFT, SHIFT, ALT).

   Once you move the cursor to the desired key, press CH KEYS. Menu 3 (Select a Key) appears. New menu choices UNSHIFT, SHIFT, and ALT each let KEYGEN access the value associated with the specified key and mode in the table, and each causes Menu 4 (Select Characters) to appear.

4. Display all key values (Menu 2—DISPLAY).

When you press DISPLAY, the entire keyboard configuration appears in a non-pictorial format, one screen at a time. Keys are displayed by key number. Their ASCII and hex values are shown for all three modes, as are their assigned attributes and mode attributes. Press CONTINUE to continue to the next screen. Press STOP to return to Menu 2.

5. Print all key values (Menu 2—PRINT).

   PRINT sends the entire keyboard configuration (in the format described for DISPLAY) to the printer, along with some header information. STOP stops the printout. Control is returned to Menu 2 when the printout is finished (or stopped).

KEY OPERATIONS 2.3

1. Change key code (Menu 2—CH KEYS; Menu 3—UNSHIFT, SHIFT, ALT; Menu 4—UNDO).

   While at Menu 2, you must press CH KEYS to see Menu 3. When the cursor is on the key to be modified, the key code can be changed in the table. You can now select one of three modes (UNSHIFT, SHIFT, ALT). Any of the three causes Menu 4 to appear, allowing you to select a character by moving the cursor over the name of the character set. When you press RETURN, the character at the cursor is entered on the scratch pad display line. Subsequent character choices are entered to the right of earlier ones. Use the Backspace key to edit your entries. Once the desired characters are entered on the display line, press RETURN to enter the key code into the table (deleting the previous value) and return control to Menu 3.

   If you choose a control character from the symbol set, its ASCII mnemonic is displayed to the left of the symbol set.

   While at Menu 4, you can change the key code that you've modified; pressing UNDO restores the original value of the key to the display line. Once you have pressed RETURN, however, you can't undo the character selection except by choosing a new key value at Menu 4.

2. Change key attributes (Menu 3—LOCK and CAPS LK).

   Once you select a key at Menu 3, you can change the ways in which Shift and Caps Lock affect that key. LOCK and CAPS LK toggle the setting of their respective key attributes; they are displayed if they are set for the key in question. If the chosen key is unaffected by Shift Lock, press LOCK to assign that attribute to the key. If Shift Lock is already assigned, press LOCK to unassign it.

3. Change key mode attributes (Menu 4—REPEAT and LOCAL).

   Menu 4 helps you change the key mode attributes. REPEAT (auto repeat) and LOCAL are toggles; they are displayed if their attributes are assigned to the chosen key. (See the explanation of LOCK and CAPS LK.)

4. Change function keys (Menu 4—FUNCTN).

   Menu 5 appears if you press FUNCTN at Menu 4. Then, you change the special function keys by moving the cursor to the desired function and pressing RETURN. This enters the appropriate code into the table, replacing the previous value. Control is returned to Menu 3.

3

SAMPLE RUN

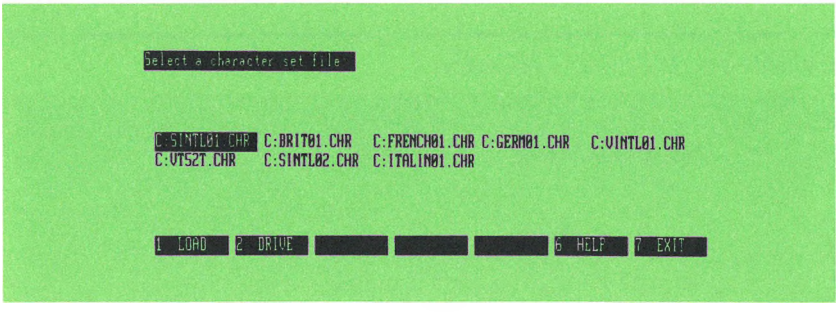
This sample run of KEYGEN takes you through the steps needed to define a keyboard table. You should duplicate the sample run to fully understand the procedures being illustrated.

Start by putting the KEYGEN diskette into drive B (after booting up your operating system) and typing:

KEYGEN

Menu 1 (File Selection) appears. All character set files (files with a .CHR extension) on the default drive are displayed at the center of the screen.

Exhibit 3a: Menu 1—File Selection



The function keys and their values are displayed at the bottom of the screen. These keys change with each successive menu. A HELP key is always available to help you decide on your next step. For example, pressing HELP displays an explanation of Menu 1. To return from a Help screen, press MENU.

All prompts are issued in reverse video at the upper left of the screen. The messages ask you to select a character set file. To make the selection, use the arrow keys to move the cursor over the character set file of your choice. Then, press LOAD to load the file. If the character set you want is on another drive, enter DRIVE. KEYGEN asks you for the drive letter and then loads the character set.

After you choose a character set, KEYGEN asks you to select a keyboard file. Follow the same procedure as in selecting a character set.

If you had pressed EXIT instead of LOAD, KEYGEN would have asked if you were sure you wanted to return to the operating system. The menu choice keys would have also changed value, so that key 1 would have been YES and key 2 NO. Pressing YES would have returned control to the operating system; pressing NO would have returned to KEYGEN.

After you select a keyboard file, Menu 2 appears on the screen.

Exhibit 3b: Menu 2—Header

The keyboard file name and header information are displayed in the center of the screen. (Header information is the information about keyboard display class, name, date, and so on, located below the file name.) To change the file name or any header information, use the arrow keys to move the cursor to the desired line. The first data field (File Name) is displayed in reverse video, with its current value to the right of it. To change the value, type:

MYKEYBRD.KB

The file MYKEYBRD.KB is your new keyboard file. As the new name appears on the screen, the values of the function keys change. Key 1 is now UNDO; pressing it restores the old file name. Pressing ENTER saves the new name.

After you have ENTERed or UNDOne the new file name, the Function keys revert to their previous values. Continue to make changes in the header information as you see fit.

You now have several options. RETURN returns you to the previous menu (in this case, the file selection menu). HELP, as always, provides more information about this menu and the choices you can make. SAVE records the keyboard you have configured under the file name you have selected (or under the original file name, if no modifications were made). PRINT prints out all keyboard information, including key numbers, character codes, and attributes. DISPLAY displays on the screen a listing of all keys and their associated character codes and attributes. CH KEYS brings up a new menu that lets you choose the keys to be configured.

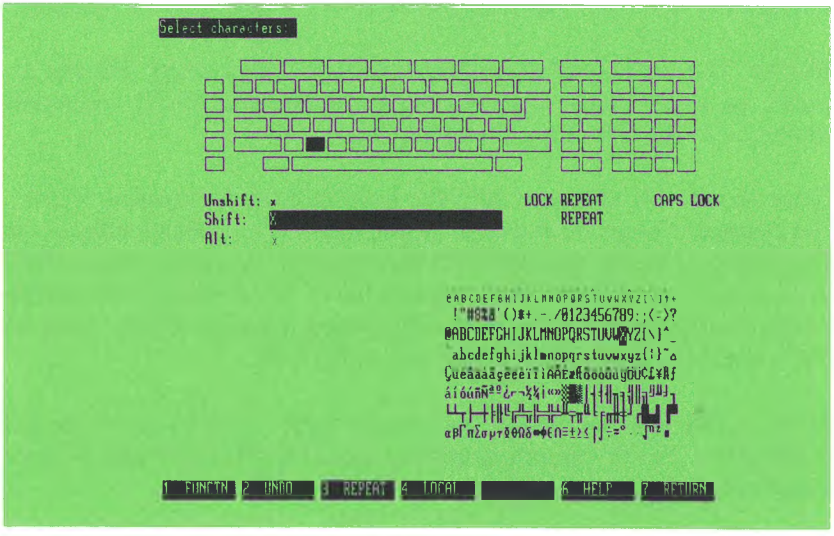
Since you haven't configured a new keyboard table, press CH KEYS to continue the configuration process. Menu 3 (Select a Key) appears. This menu lets you choose a key and configure it by giving it character codes and attributes. The keyboard is now displayed on the screen. Use the cursor movement keys to move the highlighting to the key you want to reconfigure.

Exhibit 3c: Menu 3—Select a Key

Below the keyboard, you'll see a display showing the status of the highlighted key in each of its modes (shifted, unshifted, and alternate). The values of each key are displayed to the right of the keyboard. The character set is displayed at the lower right.

The menu choices available at Menu 3 are UNSHIFT, SHIFT, ALT, LOCK, CAPS LK, HELP, and RETURN. HELP displays a Help screen for this menu, while RETURN redisplays the last menu. CAPS LK acts as a toggle on the Caps Lock attribute of the selected key. (If the key already has the Caps Lock attribute, this menu choice turns it off; if it doesn't have the attribute, CAPS LK turns it on. The same is true for the [Shift] LOCK.) UNSHIFT, SHIFT, and ALT let you configure a key in the Unshifted, Shifted, and Alternate modes. For this sample run, enter UNSHIFT to define the Unshifted mode. Menu 4 (Select Characters) appears.

Exhibit 3d: Menu 4—Select Characters



Menu 4 is used to assign characters to the selected key. Since you chose UNSHIFT at the last menu, assign characters to the key in its Unshifted mode.

The keyboard picture reappears, with the chosen key highlighted. The three modes are again displayed at left, with their values at right. The value for Unshifted mode is a blank block of highlight. The Caps Lock and Lock attributes appear to the right of this block if they were set at the previous menu. The Local and Repeat mode attributes are also displayed if the key has these attributes.

The Function keys for this menu are: FUNCTN, UNDO, REPEAT, LOCAL, HELP, and RETURN. RETURN and HELP work the same as they have in previous menus. LOCAL toggles the mode attribute Local, which is displayed when it is on (set to Local). LOCAL sends all character codes and attributes directly to the display driver. REPEAT toggles the mode attribute Repeat, which automatically repeats the generated codes when the REPEAT key is held down. Repeat is also displayed when toggled on.

To select a character (or characters) for assignment to the chosen key and mode, use the cursor arrow keys to move around the character set displayed at the lower right. Place the cursor on the desired character.

NOTE: The top row of characters are control characters. Placing the cursor on one of these displays the corresponding ASCII mnemonic character to the left of the character set.

Enter the chosen character into the keyboard file by pressing Return. The character is now displayed in the highlighted field. Enter subsequent characters (to a maximum of 32) in the same way. By doing this, a single key can produce a string of characters (such as a corporation's name). The Backspace key lets you delete characters. Pressing UNDO restores the original character field to the key mode field.

FUNCTION lets you assign a function (such as Caps Lock) to the selected key. For this demonstration, press FUNCTN. Menu 5 (Select a Function) appears.

Exhibit 3e: Menu 5—Select a Function

A list of special functions is displayed. Any function you select from this list is assigned to the chosen key and mode. (A function assignment cannot be made in conjunction with character assignments.) The available functions are:

- Null: Causes the selected key and mode to have no effect and no value.
- Right Shift: Right Shift and Left Shift have the same effect on other keys. Pressing either shift key and another key forces the second key into Shift mode. Right and Left Shift functions are commonly assigned to keys on opposite sides of the keyboard.
- Left Shift: See Right Shift.
- Shift Lock: A key with this assignment acts as a toggle. Pressing it once forces all keys with the Shift attribute into the Shift mode. Pressing the key again undoes the previous action.
- Caps Lock: This function works in the same way as Shift Lock.
- Control: The Control key is used with other keys to produce one of the 32 ASCII control codes. Only certain keys are affected by the Control key: those keys with character codes corresponding to the hex values 40 through 7F. The Control key has no effect on keys with values outside this range.
- Clear Keyboard: Disables the keyboard. Keys are inoperable until the computer is rebooted.
- Hold Screen Page Advance: The selected key becomes a Page Advance key when used with another key that has been assigned the Hold Screen escape sequence. The Hold Screen escape sequence must be enabled for this function to be used. Each time the Hold Screen Page Advance key is pressed, 24 lines of text (one page) scroll up the screen.
- Hold Screen Line Advance: The selected key becomes a Line Advance key when used with another key that has been assigned the Hold Screen escape sequence. The function is used much like Hold Screen Page Advance, but it causes the screen to scroll up one line instead of an entire page.

- Alt: When used with a second key, Alt forces the second key into Alternate mode.
- Repeat: When used along with another key, Repeat causes that key to be repeated as long as the Repeat key is depressed.

Use the cursor arrow keys to move the cursor to the desired function. Then, press ENTER to enter the assignment to the keyboard file and automatically return to Menu 4. If you decide not to assign a function after all, press MENU to return to the previous menu. As always, HELP is available if you need it.

You have just assigned a function to the selected key and are now at Menu 3, "Select a Key". The key that you have just reconfigured is highlighted on the keyboard display; its unshifted value reflects the function assignment. At this point, you can make more changes until you have fully configured the keyboard. When the configuration process is complete, press RETURN to return to the Header menu (Menu 2).

Press SAVE to save your new keyboard file. When the save is successfully completed, a message to that effect appears. You might also want to press DISPLAY, which displays all keys and their associated character codes and attributes on the screen; or PRINT, which sends the same information (along with header information) to the printer. After this is done, press RETURN to return to the file selection menu (Menu 1). Now, you can create another keyboard or you can press EXIT to leave KEYGEN. If you choose to EXIT, a message asks you if you really want to return to the operating system. Press YES to return to the operating system.

MODCON Utility

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

First VICTOR printing February, 1983.

ISBN 0-88182-006-7 Printed in U.S.A.

CONTENTS

CHAPTERS

1

OVERVIEW

MODCON is designed to take advantage of one of the most powerful features of your computer—its flexibility. With MODCON, you can modify the configuration of the operating system with respect to the keyboard table, character set, and translate table. When used with EFONT and KEYGEN, MODCON provides you with a personalized environment not possible on other machines.

MODCON lets you select a new keyboard table, translate table, and/or character set before entering an application program. The current set(s) can be saved and restored when you exit the application program.

MODCON is supported by the following operating systems:

- MS-DOS: V1.25/2.5 or later
- CP/M: V1.1/2.4 and later

If you try to use MODCON with earlier versions of either operating system, an error message appears.

Note: Translate table functionality is available with MS-DOS V1.25/2.6 and later.

2

OPERATING ENVIRONMENT

Before you can use MODCON, you must set up any keyboard and character files and translate tables.

CHARACTER SETS, KEYBOARD TABLES, AND TRANSLATE TABLES 2.1

Character sets are obtained by:

- Selecting a set provided on the system selection (SYSELECT) diskette included in the Programmer's Tool Kit.
- Selecting a graphics character set provided with CHARGRAF in the Graphics Tool Kit.
- Modifying the current set (or any available set) using the EFONT character-font editor.

Keyboard tables are obtained by:

- Selecting a table provided on the system selection (SYSELECT) diskette included in the Programmer's Tool Kit.
- Modifying the current table (or any available table) using the keyboard table editor KEYGEN.

Translate tables are associated with character sets requiring dead key sequences. These are provided for each appropriate language on the SYSELECT diskette.

2.2 MODCON OPERATION

If you request it, MODCON saves the current keyboard table or translate table and/or character set in a file on the drive you specify. The keyboard file and translate table is 2K bytes and the largest character set file is 10K bytes. All header fields are initialized with blanks except for the following:

- TYPE is K (keyboard), C (character).
- VERSION is 0.
- BANNER NAME is the file name you specify.
- FILE SECTOR COUNT is the appropriate value.

Translate table header fields are set to nulls.

Any valid keyboard or character file created by KEYGEN, EFONT or MODCON—or taken from a system selection, graphics or other diskette—can be loaded from a file and set as the active keyboard table or character set.

Translate tables are processed only when a character set is being processed. If there is a translate table with the same name as the character set you have selected, that translate table is automatically included in your configuration. If no such translate table exists, a translate table is not included.

When you save a character set, any active translate table is automatically saved on the same diskette. The translate table has the same file name as the character set file, and the extension .XLT.

2.3 COMMAND SYNTAX

To invoke MODCON, type:

MODCON <**command**>

The command portion of the invocation can have any of these formats (items enclosed in brackets are optional):

<source file>[.<source ext.>]<save file>[.<save ext.>]
<source file>[.<source ext.>]
***<save file>[.<save ext.>]**

where:

<source file> is the name of the file(s) that contain the sets to be made active. If you enter an asterisk (*), no new sets are made active and the configuration remains unchanged.

<source ext.> is .KB for keyboard or .CHR for character set. If the extension is omitted, then both keyboard and character files are made active.

<save file> is the name of the file(s) used to save the currently active keyboard table and/or character set. If you use MODCON to set a new configuration, <save file> is optional.

<save ext.> is .KB for keyboard or .CHR for character set. If the extension is omitted, then both keyboard and character sets are saved.

NOTES: (1) The <source ext.> option is independent from the <save ext.> option.
(2) Translate tables can be acted on only if a character set is being processed.

ERROR CONDITIONS 2.4

Any of the following errors cause MODCON to terminate prematurely. When this happens, none of your new configuration is saved, and the operating system is unchanged. Any BDOS or BIOS errors are returned in the normal manner.

CANNOT OPEN FILE <filename>

Make sure the file exists on the specified drive.

DISK FULL

Make room for the new file(s). Keyboard files and translate tables need 2K bytes; a character set file needs 10K.

DIRECTORY FULL

Make room for one or two new entries.

INVALID FILE EXTENSION

Specify the proper extension (.KB or .CHR).

INVALID DELIMITER

The correct delimiter is an asterisk (*).

SYSTEM ERROR

Run system diagnostics.

OPERATING SYSTEM MISMATCH

Reboot with the correct version of the operating system.

DISK ERROR <filename>

Use a different diskette.

3

USING MODCON

This chapter shows how MODCON is used both at the command level and to create a prepackaged set of programs intended for end-users.

COMMANDS 3.1

The following examples show how commands are used:

- **MODCON GERM01 G02SAVE**
  This saves the current keyboard table and character set in files G02SAVE.KB and G02SAVE.CHR on the default drive. The data in GERM01.KB and GERM01.CHR become the new active sets.

- **MODCON M01 GERM01**
  This does the same as the previous example except that the original GERM01.KB and .CHR files are overwritten in the save operation.

- **MODCON AUST01**
  This sets the new keyboard table and character sets, both named AUST01. The previous keyboard table and character sets are not saved.

- **MODCON B:GRAPHIC.CHR SAVE.CHR**
  This saves the current character set in SAVE.CHR on the default drive. The new active character set is GRAPHIC.CHR on drive B. The keyboard tables are not changed.

- **MODCON * BRIT01.KB**
  This saves the current keyboard table in file BRIT01.KB on the default drive while leaving it as the active keyboard. The character set is not changed.

- **MODCON FRENCH.KB B:SWEDISH.CHR**
  This saves the current character set in file SWEDISH.CHR on drive B, while leaving that character set active. It also sets the keyboard table from the default drive file FRENCH.KB, and overwrites the existing keyboard table without saving it.

3.2 APPLICATIONS

The following example shows how a series of commands in a batch file (MS-DOS) or submit file (CP/M-86) set up a dedicated keyboard for a WordStar word processing session. The original keyboard is restored when WordStar is terminated.

MODCON WORDAMER.KB SAVED.KB	The American keyboard is set as the dedicated keyboard for the WordStar session. The original keyboard is saved so it can be restored at the end of the session.
WS	WordStar is invoked.
MODCON SAVED.KB	The original keyboard is restored.
DEL SAVED.KB	This deletes the original keyboard file (MS-DOS).
[ERA SAVED.KB]	This deletes the original keyboard (CP/M-86).

A

ASCII CHARACTER CODES

In the column headings, Dec means decimal, Hex means hexadecimal (H) and CHR means character.

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

Dec	Hex	CHR	Dec	Hex	CHR
069	45H	E	099	63H	c
070	46H	F	100	64H	d
071	47H	G	101	65H	e
072	48H	H	102	66H	f
073	49H	I	103	67H	g
074	4AH	J	104	68H	h
075	4BH	K	105	69H	i
076	4CH	L	106	6AH	j
077	4DH	M	107	6BH	k
078	4EH	N	108	6CH	l
079	4FH	O	109	6DH	m
080	50H	P	110	6EH	n
081	51H	Q	111	6FH	o
082	52H	R	112	70H	p
083	53H	S	113	71H	q
084	54H	T	114	72H	r
085	55H	U	115	73H	s
086	56H	V	116	74H	t
087	57H	W	117	75H	u
088	58H	X	118	76H	v
089	59H	Y	119	77H	w
090	5AH	Z	120	78H	x
091	5BH	[	121	79H	y
092	5CH	\	122	7AH	z
093	5DH	]	123	7BH	{
094	5EH	^	124	7CH	|
095	5FH	_	125	7DH	}
096	60H	'	126	7EH	~
097	61H	a	127	7FH	DEL
098	62H	b			

Note: LF = Linefeed, FF = Form feed, CR = Carriage return, and DEL = Delete.

B

MACRO-86 DIRECTIVES

MEMORY DIRECTIVES B.1

```
                ASSUME <seg-reg>:<seg-name>[,<seg-reg>:<seg-name>...]
                ASSUME NOTHING
                COMMENT <delim><text><delim>

<varname>       DB  <exp> [,<exp>,...]
<varname>       DQ  <exp> [,<exp>,...]
<varname>       DT  <exp> [,<exp>,...]
<varname>       DW <exp> [,<exp>,...]
<varname>       DD  <exp> [,<exp>,...]

                END [<exp>]
  <name>        EQU <exp>
  <name>        =<exp>
                EVEN
                EXTRN <name>:<type>[,<name>:<type>...]
  <name>        GROUP <segment-name>[,...]
                INCLUDE <filename>
  <name>        LABEL <type>
                NAME <module-name>
                ORG <exp>

  <name>        PROC [NEAR]
  <name>        PROC [FAR]
                  |
                RET
<proc-name>     ENDP

                PUBLIC <symbol>[,<symbol>...]
                .RADIX <exp>
  <name>        RECORD <field>:<width>[=<exp>][,...]

<seg-name>      SEGMENT [<align>][<combine>][<'class'>]
                  |
<seg-name>      ENDS

<struc-name>    STRUC
                  |
<struc-name>    ENDS
  <variable>      <field>
```

B.2 MACRO DIRECTIVES

```
          ENDM
          EXITM
          IRP <dummy>,<parameters in angle brackets>
          IRPC <dummy>,<string>
          LOCAL <parameter>[,<parameter>...]
<name>    MACRO <parameter>[,<parameter>...]
          PURGE <macro-name>[,...]
          REPT <exp>
```

B

B.2.1 SPECIAL MACRO OPERATORS

& (ampersand)—concatenation
<text> (angle brackets)—single literal
;; (double semicolons)—suppress comment
! (exclamation point)—next character literal
% (percent sign)—convert expression to number

B.3 CONDITIONAL DIRECTIVES

```
ELSE
ENDIF
IF <exp>
IFB <arg>
IFDEF <symbol>
IFDIF <arg1>,<arg2>
IFE <exp>
IFIDN <arg1>,<arg2>
IFNB <arg>
IFNDEF <symbol>
IF1
IF2
```

LISTING DIRECTIVES B.4

.CREF
.LALL
.LFCOND
.LIST
%OUT <text>
PAGE [<length>] [,<width>]
PAGE [+]
.SALL
.SFCOND
SUBTTL <text>
.TFCOND
TITLE <text>
.XALL
.XCREF [<variable list>]
.XLIST

ATTRIBUTE OPERATORS B.5

OVERRIDE OPERATORS B.5.1

Pointer (PTR)
  <attribute> PTR <expression>
Segment Override (:)
  <segment-register>:<address-expression>
  <segment-name>:<address-expression>
  <group-name>:<address-expression>
SHORT
  SHORT <label>
THIS
  THIS <distance>
  THIS <type>

B.5.2 VALUE-RETURNING OPERATORS

SEG
  SEG <label>
  SEG <variable>
OFFSET
  OFFSET <label>
  OFFSET <variable>
TYPE
  TYPE <label>
  TYPE <variable>
.TYPE
  .TYPE <variable>
LENGTH
  LENGTH <variable>
SIZE
  SIZE <variable>

B.5.3 RECORD-SPECIFIC OPERATORS

Shift-count—(Record fieldname)
  <record-fieldname>
MASK
  MASK <record-fieldname>
WIDTH
  WIDTH <record-fieldname>
  WIDTH <record>

PRECEDENCE OF OPERATORS B.6

All operators in a single item have the same precedence, regardless of the order listed within the item. Spacing and line breaks are used for clarity, not to indicate functional relations.

1. LENGTH, SIZE, WIDTH, MASK: Entries can be inside parentheses (), angle brackets <>, and square brackets []. The structure of a variable operand is:

   <variable>.<field>

2. Segment override operator (:)
3. PTR, OFFSET, SEG, TYPE, THIS
4. HIGH, LOW
5. Asterisk (*), slash (/), MOD, SHL, SHR
6. Plus sign (+) and minus sign (−), both unary and binary.
7. EQ, NE, LT, LE, GT, GE
8. Logical NOT
9. Logical AND
10. Logical OR, XOR
11. SHORT, .TYPE

C

8086 INSTRUCTIONS

The mnemonics are listed alphabetically with their full names. The 8086 instructions are also listed in groups based on the types of arguments the instruction takes.

ALPHABETICAL LIST OF 8086 INSTRUCTION MNEMONICS C.1

MNEMONIC	FULL NAME
AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication
AAS	ASCII adjust for subtraction
ADC	Add with carry
ADD	Add
AND	AND
CALL	CALL
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP	Compare
CMPS	Compare byte or word (of string)
CMPSB	Compare byte string
CMPSW	Compare word string
CWD	Convert word to double word
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC	Decrement
DIV	Divide
ESC	Escape
HLT	Halt
IDIV	Integer divide
IMUL	Integer multiply

MNEMONIC	FULL NAME
IN	Input byte or word
INC	Increment
INT	Interrupt
INTO	Interrupt on overflow
IRET	Interrupt return
JA	Jump on above
JAE	Jump on above or equal
JB	Jump on below
JBE	Jump on below or equal
JC	Jump on carry
JCXZ	Jump on CX zero
JE	Jump on equal
JG	Jump on greater
JGE	Jump on greater or equal
JL	Jump on less than
JLE	Jump on less than or equal
JMP	Jump
JNA	Jump on not above
JNAE	Jump on not above or equal
JNB	Jump on not below
JNBE	Jump on not below or equal
JNC	Jump on no carry
JNE	Jump on not equal
JNG	Jump on not greater
JNGE	Jump on not greater or equal
JNL	Jump on not less than
JNLE	Jump on not less than or equal
JNO	Jump on not overflow
JNP	Jump on not parity
JNS	Jump on not sign
JNZ	Jump on not zero
JO	Jump on overflow
JP	Jump on parity
JPE	Jump on parity even
JPO	Jump on parity odd
JS	Jump on sign
JZ	Jump on zero
LAHF	Load AH with flags
LDS	Load pointer into DS
LEA	Load effective address
LES	Load pointer into ES
LOCK	LOCK bus
LODS	Load byte or word (of string)
LODSB	Load byte (string)
LODSW	Load word (string)

MNEMONIC	FULL NAME
LOOP	LOOP
LOOPE	LOOP while equal
LOOPNE	LOOP while not equal
LOOPNZ	LOOP while not zero
LOOPZ	LOOP while zero
MOV	Move
MOVS	Move byte or word (of string)
MOVBS	Move byte (string)
MOVSW	Move word (string)
MUL	Multiply
NEG	Negate
NOP	No operation
NOT	NOT
OR	OR
OUT	Output byte or word
POP	POP
POPF	POP flags
PUSH	PUSH
PUSHF	PUSH flags
RCL	Rotate through carry left
RCR	Rotate through carry right
REP	Repeat
RET	Return
ROL	Rotate left
ROR	Rotate right
SAHF	Store AH into flags
SAL	Shift arithmetic left
SAR	Shift arithmetic right
SBB	Subtract with borrow
SCAS	Scan byte or word (of string)
SCASB	Scan byte (string)
SCASW	Scan word (string)
SHL	Shift left
SHR	Shift right
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOS	Store byte or word (of string)
STOSB	Store byte (string)
STOSW	Store word (string)
SUB	Subtract
TEST	TEST
WAIT	WAIT
XCHG	Exchange
XLAT	Translate
XOR	Exclusive OR

C.2 8086 INSTRUCTION MNEMONICS BY ARGUMENT TYPE

In this section, instructions are grouped by the type of argument(s) that they take. In each group, the instructions are listed alphabetically in the first column. The formats of the instructions with the valid argument types are shown in the second column. If a format shows OP, that format is legal for all the instructions shown in that group. If a format is specific to one mnemonic, that mnemonic is shown in the format in place of OP.

C

The following abbreviations are used in this section:

- OP: Opcode; instruction mnemonic.
- reg: Byte register (AL, AH, BL, BH, CL, CH, DL, DH) or word register (AX, BX, CX, DX, SI, DI, BP, SP).
- r/m: Register or memory address; or indexed and/or based.
- accum: AX or AL register.
- immed: Immediate.
- mem: Memory operand.
- segreg: Segment register (CS, DS, SS, ES).

INSTRUCTION TYPE	MNEMONIC	ARGUMENT TYPES
General 2-Operand	ADC	OP reg, r/m
	ADD	OP r/m, reg
	AND	OP accum, immed
	CMP	OP r/m, immed
	OR	
	SBB	
	SUB	
	TEST	
	XOR	
CALL/JUMP Type	CALL	OP mem {NEAR}{FAR} direction OP r/m
	JMP	(indirect data: DWORD, WORD)
Relative Jumps	JA	OP addr (+129 or −126 of IP at start, or
	JAE	±127 at end of jump instruction)

INSTRUCTION TYPE	MNEMONIC	ARGUMENT TYPES
	JB	
	JBE	
	JC	
	JCXZ	
	JE	
	JG	
	JGE	
	JL	
	JLE	
	JNA	
	JNAE	
	JNB	
	JNBE	
	JNC	
	JNE	
	JNG	
	JNGE	
	JNL	
	JNLE	
	JNO	
	JNP	
	JNS	
	JNZ	
	JO	
	JP	
	JPE	
	JPO	
	JS	
	JZ	
Loop	LOOP	Same as relative jumps
	LOOPE	
	LOOPZ	
	LOOPNE	
	LOOPNZ	
Return	RET	[immed] (optional, number of words to POP)
No Operand	AAA	
	AAD	
	AAM	
	AAS	
	CBW	
	CLC	
	CLD	
	CLI	

INSTRUCTION TYPE	MNEMONIC	ARGUMENT TYPES
	CMC	
	CMPSB	
	CMPSW	
	CWD	
	DAA	
	DAS	
	HLT	
	INTO	
	IRET	
	LAHF	
	LODSB	
	LODSW	
	MOVSB	
	MOVSW	
	NOP	
	POPF	
	PUSHF	
	SAHF	
	SCASB	
	SCASW	
	STC	
	STD	
	STI	
	STOSB	
	STOSW	
	WAIT	
	XLATB	
Load	LDS	OP r/m (except that OP reg is illegal)
	LEA	
	LES	
Move	MOV	OP mem, accum
		OP accum, mem
		OP segreg, r/m (except CS is illegal)
		OP r/m, segreg
		OP r/m, reg
		OP reg, r/m
		OP reg, immed
		OP r/m, immed
Push/Pop	PUSH	OP word-reg
	POP	OP segreg (POP CS is illegal)
		OP r/m
Shift/Rotate Type	RCL	OP r/m, 1
	RCR	OP r/m, CL
	ROL	
	ROR	
	SAL	
	SHL	
	SAR	
	SHR	

INSTRUCTION TYPE	MNEMONIC	ARGUMENT TYPES
Input/Output	IN	IN accum, byte-immed (immed = port 0-255)
		IN accum, DX
	OUT	OUT immed, accum
		OUT DX, accum
Increment/Decrement	INC	OP word-reg
	DEC	OP r/m
Arithmetic Multiplication/ Division/Negate/Not	DIV	OP r/m (implies AX OP r/m, except NEG)
	IDIV	
	MUL	
	IMUL	
	NEG	(NEG implies AX OP NOP)
	NOT	
Interrupt	INT	INT 3 (value 3 is one byte instruction)
		INT byte-immed
Exchange	XCHG	XCHG accum, reg
		XCHG reg, accum
		XCHG reg, r/m
		XCHG r/m, reg
Miscellaneous	XLAT	XLAT byte-mem (only checks argument, not in opcode)
	ESC	ESC 6-bit-number, r/m
String Primitives (1)	CMPS	CMPS byte-word, byte-word (CMPS right operand is ES)
	LODS	LODS byte/word, byte/word (LODS one argument = no ES)
	MOVS	MOVS byte/word, byte/word (MOVS left operand is ES)
	SCAS	SCAS byte/word, byte/word (SCAS one argument = ES)
	STOS	STOS byte/word, byte/word (STOS one argument = ES)
Repeat Prefix To String	LOCK	
	REP	
	REPE	
	REPZ	
	REPNE	
	REPNZ	

Note (1): These instructions have bits to record only their operands, if they are byte or word and if a segment override is involved.

D

MS-DOS FILE CONTROL BLOCK DEFINITION

The MS-DOS File Control Block (FCB) is defined as follows:

Byte 0	Drive Code. Zero specifies the default drive, 1 is drive A, 2 is drive B, and so on. After a successful open, the drive letter specification is always a physical drive, and the default specification is changed to the current default physical drive.
Bytes 1–8	Filename. If the file is less than 8 characters, the name must be left-justified with trailing blanks. A filename must consist of letters (lowercase is converted to uppercase), numbers, and special characters. A filename cannot contain these characters: + , . / : ; = [ ] and must not contain alternate characters (<20H).
Bytes 9–11	Extension to filename. If the extension is less than three characters, it must be left-justified with trailing blanks. The extension can also be all blanks.
Bytes 12–13	Current block (extent). This word (low byte first) specifies the current block of 128 records, relative to the start of the file, in which sequential disk reads and writes occur. If the current block is zero, then the first block of the file is accessed; if one, then the second is accessed, and so on. When combined with the current record field (byte 32), a particular logical record is identified.

Bytes 14–15	Size of the record you want to work with. This word can be filled immediately after an OPEN of the file if you do not want the default logical record size (128 bytes). The Open and Create functions set this field to 128; it is also changed to 128 if a read or write is attempted with the field set to zero.
Bytes 16–19	File size. This is the current size of the file, in bytes. It can be read by user programs but must not be written by them.
Bytes 20–21	Date. This is normally the date of the last write to the file. These bytes are set to today's date by all disk write operations and by Create. It is set by Open to the date recorded in the disk directory for the file. You can modify this field after writing to a file (but before closing it) to change the date recorded in the disk directory. The format of this 16-bit field is: ▶ Bits 0–4: Day of month. ▶ Bits 5–8: Month of year. ▶ Bits 9–15: Current year minus 1980. All zeros means no date.
Bytes 22–23	Time. Similar to Date. The format is: ▶ Bits 0–4: Seconds divided by 2. ▶ Bits 5–10: Minutes. ▶ Bits 11–15: Hours.
Bytes 24–31	Reserved for MS-DOS.
Byte 32	Current record. Within the current block of 128 records, this byte identifies the record accessed by a sequential read or write function. See bytes 12–13.

Bytes 33–36	Random Record. This field is set only if the file is to be accessed with a random read or write function. If the record size is greater than or equal to 64 bytes, only the first three bytes are used. (These bytes are a 24-bit number that represents the record's file position.) If the record size is less than 64 bytes, all four bytes are used as 32-bit numbers with the same purpose. This field is large enough to address any byte in a file of the maximum size.

THE EXTENDED FCB

The extended FCB is a special format used to search the disk directory for files with special attributes. The extended FCB consists of seven bytes in front of a normal FCB, formatted as follows:

FCB–7	Flag. FF hex is placed here to signal an extended FCB.
FCB–6 to FCB–2	Zero field.
FCB–1	Attribute byte. If bit 1 equals 1, hidden files are included in directory searches. If bit 2 equals 1, system files are included in directory searches. Any reference to an FCB in the description of MS-DOS function calls, whether opened or unopened, can use either a normal FCB or an extended FCB. A normal FCB has the same effect as an extended FCB with the attribute byte set to zero.

MS-DOS BASE PAGE

The Base Page contains information about the standard execution environment established by COMMAND. The Base Page is always pointed to by DS:0 at program startup. A new Base Page can be created by a system call, but it does not override the current Base Page.

The Base Page is actually defined as the storage at CS:0 for certain system calls. Another Base Page can be created by a system call, but is not used until control is transferred to another program. A program that does not follow the CS:0 = Base Page convention must ensure that a Base Page is addressable by any system calls that depend on the CS:0 convention.

Program termination is of particular significance. It is handled through the INT 20H in the Base Page. A program can save DS at startup and, at termination, execute a long jump to the DS:0.

The MS-DOS Base Page is defined as follows:

Bytes 0–1H	These bytes contain an INT 20H instruction that simplifies termination in programs that do not preserve CS:0 = Base Page.
Bytes 2–3H	This word contains the value of the last segment in the TPA + 1. This value is the first unavailable memory segment at the high end. The low end of the TPA is contained in CS and DS at the time a program begins execution.
Byte 4H	Undefined.
Byte 5H	Alternate function request entry point.
Bytes 6–7H	This word contains the byte count of the TPA. If the TPA exceeds 64K, then this value is 0FFF0H.
Bytes 8–9H	Reserved.
Bytes 0A–0CH	Exit interrupt vector at startup (INT 22H).
Bytes 0D–11H	ALT-C interrupt vector at startup (INT 23H).
Bytes 12–15H	Fatal error interrupt vector at startup (INT 24H).
Bytes 16–5BH	Reserved.

Bytes 5C–67H	First filename parameter. By convention, the input command tail is parsed for two filenames when the Base Page is created by the invoking program (usually COMMAND). These bytes are the first filename encountered.
Bytes 68–6BH	Reserved.
Bytes 6C–77H	Second filename parameter.
Bytes 78–7FH	Reserved.
Bytes 80–FFH	Command parameters unformatted. The command name is not included here. COMMAND recognizes commands of the format: <**commandname**> <**parameters**> If present, the parameters are copied into bytes 81H through FFH, with byte 80H equal to the length of the parameters. The parameters are followed by a carriage return.

E

MS-DOS INTERRUPTS AND FUNCTION CALLS

INTERRUPTS E.1

MS-DOS reserves interrupt types 20 to 3F hex for its use. Absolute locations 80 to FF hex are the transfer address storage locations reserved by the DOS.

This section describes the defined interrupts. All values are in hexadecimal.

20 Program terminate. This is the normal way to exit a program. When this interrupt is executed, CS:0 must point to the 100H parameter area. This vector transfers control to the DOS for restoration of exit addresses (interrupts 22H, 23H, 24H) to the values they had on entry to the program. All file buffers are read to disk. Before issuing this interrupt, you should close any files that have changed in length (see function call 10 hex). If the changed file was not closed, its length is not recorded correctly in the directory. Control is then transfered to INT 22H.

21 Function request. See Section E.2.

22 Termination handler. When the program terminates, control is transferred to the address represented by this interrupt (88-8B hex) after the DOS does the processing described in INT 20H. This address is copied into the Base Page at the time the Base Page is created. If a program is to execute a second program, the first program must set the terminate address before creating the Base Page for the second program. Otherwise, control would transfer to the host program's termination address when the second program terminates.

23 ALT-C handler. If you type ALT-C during keyboard input, video output, list output or auxiliary input/output, an ALT-C and a carriage return/linefeed appear on the screen and an INT 23H is executed. If the INT 23H routine saves all registers, the routine can end with a return-from-interrupt instruction (IRET) to continue execution of the DOS request. If functions 9 or 10 (buffered output and input) were being executed, then I/O continues from the start of the line. All other functions are started anew (the ALT-C check is made before function execution).

When the INT 23H is executed, all registers are set to the value they had when the original call to MS-DOS was made, and the top-of-stack contains the interrupt return address to the DOS, followed by the interrupt return address to the program that invoked the DOS. There are no restrictions on what the ALT-C handler is allowed to do, as long as the registers are unchanged when IRET is used.

If the program creates a new segment and loads in a second program which itself changes the ALT-C address, termination of the second program and return to the first causes the ALT-C address to be restored to the value it had before execution of the second program. (See INT 20H.)

24 Fatal error handler. When a fatal error occurs within MS-DOS, control is transferred with an INT 24H. On entry to the error handler, bit 7 of AH equals 0 if the error was a hard disk error (probably the most common occurrence); bit 7 equals 1 if the error was of some other type.

With a hard disk error, bits 0–2 include the following:

- Bit 0: 0 if read, 1 if write.
- Bits 1 and 2 identify the affected disk area:

BIT 2	BIT 1	AFFECTED DISK AREA
0	0	Reserved area
0	1	File allocation table
1	0	Directory
1	1	Data area

AL, CX, DX, and DS:BX are set up to retry the transfer with INT 25H or INT 26H. DI has a 16-bit error code returned by the BIOS. The values returned are defined by the BIOS.

The registers are set up for a BIOS disk call and the returned code is in the lower half of the DI register; the upper half is undefined. The user stack looks like this:

IP CS FLAGS	Registers such that if an IRET is executed, the DOS responds according to (AL) as follows: ► (AL) = 0: Ignore the error. ► (AL) = 1: Retry the operation. Stack DS, BX, CX and DX must not be modified. ► (AL) = 2: Abort the program.
AX BX CX DX SI DI BP DS ES	User registers at time of request.
IP CS FLAGS	The interrupt from the user to the DOS.

Currently, the only error possible when AH bit 7 equals 1 is a bad memory image of the file allocation table.

25 Absolute disk read. Transfers control directly to the BIOS. On return, the original flags are still on the stack (put there by the INT instruction). This allows return information to be passed back in the flags. Be sure to pop the stack to prevent uncontrolled growth.

For this entry point, records and sectors are the same size. The request is as follows:

(AL)	Driver number (defined by BIOS)
(CX)	Number of sectors to read
(DX)	Beginning logical record number
(DS:BX)	Transfer address

The number of records specified are transferred using the given driver and the transfer address. The driver number is defined at the DOS/BIOS interface and is implementation-specific. Logical record numbers are obtained by numbering each sector sequentially starting from zero and continuing across track boundaries. For example, logical record number 0 is track 0, sector 1; logical record number 12 hex is track 2, sector 3.

All registers except the segment registers are destroyed by this call. If the transfer was successful the carry flag (CF) is zero. If the transfer was not successful CF equals 1 and (AL) indicates the error as defined by the BIOS.

26 Absolute disk write. This is the counterpart to interrupt 25. Except that this is a write, the description of interrupt 25 applies to interrupt 26 as well.

27 Terminate but stay resident. Used by programs that are to remain resident when COMMAND regains control. A program of this type is loaded as an executing .COM file by COMMAND. After the program initializes, it must set DX to its last address plus one in the segment in which the program is executing; then, it executes an interrupt 27H.

COMMAND then treats the program as an extension of MS-DOS. The program is not overlaid when other programs are executed. The area occupied by the program is at the lower end of the TPA. More storage can be reserved by first establishing a new Base Page; the TPA is adjusted relative to the Base Page at CS:0.

E.2 FUNCTION CALLS

You call a function by putting a function number in the AH register, supplying additional information in other registers as necessary for the specific function, and then executing an interrupt 21H. When MS-DOS takes control, it switches to an internal stack after pushing the registers. All user registers are preserved (except AX), unless information is passed back to the calling program. The user stack must be able to accommodate the interrupt and the environment save. We recommend that the stack be 80 hex in addition to any user registers.

There is an additional calling method that conforms to CP/M calling conventions. The function number is placed in the CL register; other registers are set as normal according to the function specification. Then, an intrasegment call is made to location 5 in the Base Page. This method is available only to functions which do not pass a parameter in AL, and whose numbers are equal to or less than 36. Register AX is always destroyed if this mechanism is used; otherwise, the method works the same as other function calls.

These functions are available. All values are in hex.

SYSTEM SERVICE CALLS E.2.1

0 Program terminate. The Base Page must be addressed by CS:0. The terminate and ALT-C exit addresses are restored to the values they had on entry to the terminating program. All file buffers are read to disk. Files that have changed in length without being closed are not recorded correctly in the disk directory. Control transfers to the termination handler (INT 22H).

1B Allocation table address. On return, DS:BX points to the allocation table for the current drive. DX contains the number of allocation units, AL contains the number of records per allocation unit, and CX contains the size of the physical sector. The byte at DS:[BX-1] (just ahead of the allocation table) is the "dirty byte" for the table. If the dirty byte is set to 01, the table has been modified and must be written back to disk. If set to 00, the table is not modified. Any programs that get the address and directly modify the table must set this byte to 01 for the changes to be recorded. The dirty byte should never be set to 00; instead, do a DISK RESET function (#0D hex) to write the table and reset the bit.

1C This function is equivalent to 1B above, except that DL is the letter drive for the allocation table wanted. If AL = 0FFH, then the supplied drive is invalid.

25 Set vector. The interrupt type specified in AL is set to the 4-byte address DS:DX.

26 Create a new program segment. On entry, DX contains a segment number at which to set up a new program segment. The entire 100-hex area at location CS:0 (the current program segment) is copied into location DX:0 (the new program segment). The memory-size information at location 6 in the new segment is updated, and the current termination and ALT-C exit addresses are saved at DX:0AH (see Base Page definition).

29 Parse filename. On entry, DS:SI points to a command line to parse, and ES:DI points to a portion of memory to be filled with an unopened FCB. Leading tabs and spaces are ignored when scanning. If bit 0 of AL is equal to 1 on entry, then one leading filename separator (at most) is ignored, along with any trailing tabs and spaces. The four filename separators are:

; , = +

If bit 0 of AL is equal to 0, then all parsing stops when a separator is encountered. The command line is then parsed for a filename of the form

d:filename.ext

If found, a corresponding unopened FCB is created at ES:DI. The entry value of AL bits 1, 2, and 3 determine what to do if the drive, filename, or extension, respectively, are missing. In each case, if the bit is a zero and the field is not present on the command line, then the FCB is filled with a fixed value. This value is 0 for the default drive of the drive field; and all blanks for the filename and extension fields. If the bit is a 1, and the field is not present on the command line, then the corresponding field in the destination FCB at ES:DI is left unchanged. If an asterisk (*) appears in the filename or extension, then all remaining characters in the name or extension are question marks (?).

These characters are illegal within MS-DOS file specifications:

" / [ ] + = ; ,

In addition, alternate characters and spaces cannot appear in file specifications. Parsing stops when any of these characters are encountered, or when the period (.) or colon (:) is found in an invalid position.

If the drive is invalid, then AL = 0FFH and filename is parsed. If / or * appear in the filename or extension of a file on a valid drive, then AL returns 01; otherwise, it returns 00. DS:SI returns pointing to the first character after the filename. ES:DI is unchanged.

2A Get date. Returns date in CX:DX.

- CX contains the year.
- DH contains the month (January = 1, February = 2 and so on).
- DL contains the day of the month.
- AL contains the day of week (Sunday = 0, Monday = 1 and so on).

The number of days in each month (and the variation in month length during leap years) is taken into account when incrementing the registers.

2B Set date. On entry CX:DX must contain a valid date in the same format used by function 2A. If the date is valid and the set operation is successful, then AL returns 00. If the date is not valid, AL returns FF.

2C Get time. Returns with time-of-day in CX:DX. Time is represented as four 8-bit binary quantities:

- CH contains the hour (0–23).
- CL contains the minute (0–59).
- DH contains the second (0–59).
- DL contains the 1/100 second (0–99).

This format can be easily converted to a printable form or used in calculations.

2D Set time. On entry, CX:DX contains the time in the same format as returned by function 2C. If any component of the time is not valid, the set operation is aborted and AL returns FF. If the time is valid, AL returns 00.

2E Set/Reset Verify Flag. On entry, DL must be 0. AL contains the verify flag:

- 0 indicates no verify.
- 1 indicates verify after write.

On each write, this flag is passed to the I/O system for interpretation.

E.2.2 BYTE I/O

1 Keyboard input. Waits for you to type a character at the keyboard, then sends that character to the screen and returns it in AL. The character is checked for an ALT-C. If ALT-C is detected, an INT 23 hex is executed.

2 Video output. The character in DL is output to the screen. If an ALT-C is detected after the output, an INT 23 hex is executed.

3 Auxiliary input. Waits for a character from the auxiliary input device, then returns that character in AL. If an ALT-C is detected after the output, an INT 23 hex is executed.

4 Auxiliary output. The character in DL is output to the auxiliary device. If an ALT-C is detected after the output, an INT 23 hex is executed.

5 Printer output. The character in DL is output to the printer. If an ALT-C is detected after the output, an INT 23 hex is executed.

6 Direct keyboard I/O. If DL is FF hex, then AL returns with the keyboard input character, if one is available; otherwise, AL returns 00. If DL is not FF hex, then DL is assumed to have a valid character. That character is output to the screen.

7 Direct keyboard input. Waits for you to type a character at the keyboard, then returns that character in AL. As with function 6, no checks are made on the character.

8 Keyboard input without echo. This function is identical to function 1, except the input key is not displayed. If an ALT-C is detected after the output, an INT 23 hex is executed.

9 Print string. On entry, DS:DX must point to a character string in memory that ends with a dollar sign (24 hex). Each character in the string is output to the screen in the same form as with function 2. If an ALT-C is detected after the output, an INT 23 hex is executed.

A Buffered keyboard input. On entry, DS:DX points to an input buffer. The first byte specifies the number of characters the buffer can hold; this byte cannot be 0. The second byte of the buffer is set to the number of characters input at the keyboard, excluding the carriage return (0D hex) which is always the last character. Characters are read from the keyboard and put into the buffer, beginning at the third byte. The buffer continues to fill with characters until you press Return. If you type characters until the buffer has room for only a single additional character, then your input is ignored until you type a Return.

Editing the keyboard input buffer is described in the *Operator's Reference Guide*. If an ALT-C is detected after the output, an INT 23 hex is executed.

B Check keyboard status. If a keyboard character has been typed, AL is FF hex; otherwise, AL is 00. If an ALT-C is detected after the output, an INT 23 hex is executed.

C Character input with buffer flush. First the keyboard type-ahead buffer is emptied. If AL is 1, 6, 7, 8, or 0A hex, the corresponding MS-DOS input function is executed. If AL is not one of these values, no further operation occurs, and AL returns 00.

E.2.3 DISK FUNCTIONS

All references to FCB can be normal format or extended format. These functions are also referred to as block I/O functions.

D — Disk reset. Reads all file buffers to disk. Unclosed files that have changed in size are not correctly recorded in the disk directory until they are closed. You do not need to call this function before a disk change if all files have been closed.

E — Select disk. The drive letter specified in DL (0=A, 1=B, and so on) is selected as the default disk. The number of drives is returned in AL.

F — Open file. On entry, DS:DX points to an unopened file control block (FCB). The disk directory is searched for the named file and AL returns FF hex if the file is not found. If the file is found, AL returns a 00 and the FCB is filled as follows:

- If the drive code is 0 (default disk), the code changes to the physical disk used (A=1, B=2, and so on). This lets you change the default disk without interfering with subsequent operations on this file.
- The high byte of the current block field is set to zero.
- The size of the record to be worked with (FCB bytes E-F hex) is set to the system default of 80 hex.
- The time, date, and size of the file are set in the FCB from information obtained from the directory.

If the 80 hex default is not appropriate, you must set the record size (FCB bytes E-F) to the size you want. You must also set the random record field and/or current block and record fields before further use of the FCB.

10 — Close file. This function must be called after file writes to ensure that all directory information is updated. On entry, DS:DX points to an opened FCB. The disk directory is searched and, if the file is found, its position is compared with that stored in the FCB. If the file is not found in the directory, it is assumed that the disk has been changed. In this case, AL returns FF hex. If the file is found, the directory is updated to reflect the status in the FCB and AL returns 00.

11 Search for the first entry. On entry, DS:DX points to an unopened FCB. The disk directory is searched for the first matching name (this name can contain the ? wild-card character) and, if none are found, AL returns FF hex. Otherwise, locations at the disk transfer address are set as follows:

- If the FCB used in the search is an extended FCB, then the first byte is set to FF hex and the following five bytes contain zeros. The next byte is the attribute byte from the search FCB, followed by a byte that contains the drive number used (A=1, B=2, and so on). Last comes the 32 bytes of the directory entry. In this way, the disk transfer address contains a valid unopened extended FCB with the same search attributes as the search FCB.
- If the FCB used in the search is a normal FCB, the first byte is set to the drive number used (A=1, B=2, and so on) and the next 32 bytes contain the matching directory entry. This disk transfer address contains a valid unopened normal FCB.

Directory entries are formatted as follows:

LOCATION	BYTES	DESCRIPTION
0	11	Filename and extension.
11	1	Attributes. Bits 1 or 2 make file hidden.
10	10	Zero field (for expansion).

LOCATION	BYTES	DESCRIPTION
22	2	Time: ▶ Bits 0–4 are the second times 2. ▶ Bits 5–10 are the minute. ▶ Bits 11–15 are the hour.
24	2	Date: ▶ Bits 0–4 are the day. ▶ Bits 5–8 are the month. ▶ Bits 9–15 are the year.
26	2	First allocation unit.
28	4	File size, in bytes (30 bits maximum).

11 Search for the next entry. After function 11 is called and finds a match, you can call function 12 to find the next match to an ambiguous request (for example, wild-card characters in the search filename). Inputs and outputs are the same as with function 11. The reserved area of the FCB keeps information necessary for continuing the search, so that area must not be modified.

13 Delete file. On entry, DS:DX points to an unopened FCB. All matching directory entries are deleted. If no directory entries match, AL returns FF; otherwise, AL returns 00.

14 Sequential read. On entry, DS:DX points to an opened FCB. The record addressed by the current block (FCB bytes C–D) and the current record (FCB byte 1F) is loaded at the disk transfer address. Then, the record address is incremented. If end-of-file is encountered, AL returns either 01 or 03:

- A return of 01 indicates that there is no data in the record.
- A return of 02 indicates that there is not enough room in the disk-transfer segment to read one record. The transfer is aborted.
- 03 indicates a partial record is read and filled out with zeros.

AL returns 00 if the transfer completed successfully.

15 Sequential write. On entry, DS:DX points to an opened FCB. The record addressed by the current block and current record fields is written from the disk transfer address. (If a record is of less-than-sector size, it is put into a buffer until a sector's worth of data is accumulated.) The record address is then incremented. If the disk is full, AL returns with a 01. A return of 02 means the transfer was aborted because there was not enough room in the disk transfer segment to write one record. AL returns 00 if the transfer was completed successfully.

16 Create file. On entry, DS:DX points to an unopened FCB. The disk directory is searched for an empty entry. If none is found, AL returns FF; otherwise, the entry is initialized as a zero-length file, the file is opened (see function F), and AL returns 00.

17 Rename file. On entry, DS:DX points to a modified FCB which has a drive code and filename in the usual position, and a second filename starting 6 bytes after the first (DS:DX+11 hex) in what is normally a reserved area. Each matching occurrence of the first filename is changed to the second (with the restriction that two files cannot have the exact same name and extension). If question marks (?) appear in the second filename, then the corresponding positions in the original name are unchanged. AL returns FF hex if no match was found; otherwise, AL returns 00.

19 Current disk. AL returns with the code of the current default drive (0=A, 1=B, and so on).

1A Set disk transfer address. The disk transfer address is set to DS:DX. MS-DOS does not let disk transfers wrap around within the segment, nor overflow into the next segment.

21 Random read. On entry, DS:DX points to an opened FCB. The current block and current record are set to agree with the random record field, then the record addressed by these fields is loaded at the current disk transfer address. If end-of-file is encountered, AL returns either 01 or 03. If 01 is returned no more data is available. If 03 is returned, a partial record is available, filled out with zeros. A return of 02 means there was not enough room in the disk transfer segment to read one record, so the transfer was aborted. AL returns 00 if the transfer was completed successfully.

22 Random write. On entry, DS:DX points to an opened FCB. The current block and current record are set to agree with the random record field. The record addressed by these fields is written (or, in the case of records less than sector-sized, buffered) from the disk transfer address. If the disk is full, AL returns 01. A return of 02 means the transfer was aborted because there was not enough room in the disk transfer segment to write one record. AL returns 00 if the transfer was completed successfully.

23 File size. On entry, DS:DX points to an unopened FCB. The disk directory is searched for the first matching entry and, if none is found, AL returns FF. Otherwise, the random record field is set with the size of the file (in terms of the record-size field rounded up) and AL returns 00.

24 Set random record field. On entry, DS:DX points to an opened FCB. This function sets the random record field to the same file address as the current block and record fields.

27 Random block read. On entry, DS:DX points to an opened FCB, and CX contains a record count that must not be zero. The specified number of records (in terms of the record-size field) are read from the file address specified by the random record field into the disk transfer address. If end-of-file is reached before all records are read, AL returns either 01 or 03:

- A return of 01 indicates end-of-file; the last record is complete.
- A 03 indicates that the last record is a partial record.

If wrap-around occurs above address FFFF hex in the disk transfer segment, as many records as possible are read, and AL returns 02. If all records are read successfully, AL returns 00. In any case, CX returns with the actual number of records read. The random record field and the current block/record fields are set to address the next record.

28 Random block write. The same as function 27, except for writing and a write-protect indication. If there is insufficient space on the disk, AL returns 01 and no records are written. If CX is 0 at entry, no records are written; instead, the file is set to the length specified by the random record field (allocation units are released or allocated as appropriate).