Chapter 5 DISK FILES

This Chapter describes procedures for creating and accessing files on flexible disks (including disk image RAM) with QX-10 MFBASIC. The types of files which are covered include program files, random access files, and sequential access files. In reading this Chapter, keep in mind that disk file management is a process which involves a number of interrelated commands and statements, each of which must be prepared with consideration for the others. Also be sure to specify file descriptors in accordance with the rules described in the "Files" section of Chapter 2.

A summary of procedures for handling errors occurring during disk access is included at the end of this Chapter, together with a review of general precautions to be observed in using flexible disks.

5.1 Program Files

This section reviews the commands and statements used to manipulate program files. In specifying these commands/statements, remember that the disk drive which is currently logged in (that from which MFBASIC was started) is assumed unless otherwise specified in <file descriptor>. Also remember that the CP/M operating system will automatically assume that the file name extension is ".BAS" unless another extension is explicitly specified.

```
SAVE <file descriptor>[ |,A| ]
                        |,P|
```

This command writes the program in memory to the disk under the file name specified in <file descriptor>. If neither the A nor P options are specified, the program is written to the disk in compressed binary format. If the A option is specified, the file is written as a series of ASCII characters. If the P option is specified, the program is saved in encoded binary format. A program saved using the P (protect) option cannot be listed or edited when it is later reloaded; therefore, it is recommended that you also save an unprotected copy of the program for future listing or editing.

```
LOAD <file descriptor>[,R]
```

This command loads the program specified in <file descriptor> into memory from the disk. If the R option is specified, the program will be automatically executed as soon as loading is completed. Executing this command without the R option closes all files which are currently open; however, files will not be closed if the R option is specified. This makes it possible to chain programs which access the same data files.

```
RUN <file descriptor>[,R]
```

If <file descriptor> is omitted, this command executes the program which is currently in memory. Specifying <file descriptor> causes the specified program to be loaded into memory from the disk (deleting any program currently in memory) and immediately executes it. As with the LOAD command, all open files are closed upon execution of this command unless the R option is specified.

MERGE <file descriptor>

The MERGE command loads the specified program into memory from the disk and merges it with the program in memory. (The program MERGEd must have been stored in ASCII format.) If any lines of the program loaded have the same line numbers as those of the program in memory, corresponding program lines in memory are replaced with those of the program from the disk. MFBASIC always returns to the command level after execution of a MERGE command.

KILL <file descriptor>

This command deletes the specified file from the disk. The function of this command is the same regardless of whether the file specified is a system file, program file, or random or sequential access data file; therefore, great care should be exercised in using it.

NAME <old filename> AS <new filename>

This command is used to change the name of a disk file. Specify the current file name in <old filename> and the new name which is to be assigned to the file in <new filename>. This command can be used to rename any type of file.

5.2 Sequential Files

This section describes procedures for creating, accessing, and updating sequential data files. Sequential files are easier to create than random files, but they are not as easy to update and take longer to access. As the name implies, the items included in a sequential file are stored in the file in the order in which they are written, and must be read back in the same order.

The statements and functions used to write or read sequential files are as follows.

```
OPEN
PRINT#, PRINT# USING, WRITE#
INPUT#, LINE INPUT# CLOSE, EOF, LOC
```

5.2.1 Creating sequential files

The steps involved in creating and accessing a sequential disk file are as follows.

(1) Execute an OPEN statement to assign a file number to the file and to open it in the "O" (output) mode.

Example `OPEN"O",#1,"CLIENTS.DAT"`

(2) Write data to the file using the PRINT# or WRITE# statement.

Example
```
PRINT#1,A$;",";B$;",";C$
WRITE#1,A$,B$,C$
```

(3) Close the file by executing a CLOSE statement. (This must be done before the file can be reopened in the "I" (input) mode for input.

Example `CLOSE#1`

5.2.2 Accessing sequential files

The procedures for accessing sequential files are as follows.

(1) Execute an OPEN statement to open the file in the "I" mode.

Example `OPEN "I",#1,"CLIENTS.DAT"`

(2) Read data from the file into variables in memory by executing the INPUT# (or LINE INPUT#) statement.

Examples
```
INPUT#1,A$,B$,C$
LINE INPUT#1,A$,B$,C$
```

(3) Close the file after input has been completed by executing a CLOSE statement.

Example CLOSE # 1

Note that data is read from the beginning of the file each time the file is opened; also note that, if all data included in the file is to be read at once, a DIM statement must be executed to dimension one or more variable arrays of appropriate size.

5.2.3 Programming examples for sequential file access

The following is a short program which creates a sequential disk file.

```
10 OPEN "O",#1,"A:EMPLOYEE.DAT"
20 INPUT "NAME";N$
30 IF N$="XX" THEN CLOSE:END
40 INPUT "SECTION";S$
50 INPUT "DATE OF BIRTH";D$
60 PRINT #1,N$;",";S$;",";D$
70 PRINT: GOTO 20
```

```
RUN
NAME? UNCLE SCROOGE
SECTION? ACCOUNTING
DATE OF BIRTH? 01/12/44

NAME? MICKEY MOUSE
SECTION? ENTERTAINMENT
DATE OF BIRTH? 02/22/43

NAME? LUCKY FUNG
SECTION? ACCOUNTING
DATE OF BIRTH? 03/13/48

NAME? VELMA BOOT
SECTION? CUSTOMER SERVICE
DATE OF BIRTH? 07/30/55

NAME? XX
Ok
```

In this example, data INPUT into variables N$, S$, and D$ is written to sequential file "EMPLOYEE.DAT" each time the PRINT # statement on line 60 is executed. Program execution ends and the file is closed when "XX" is input in response to "NAME?".

Note that, in the example above, the PRINT# USING or WRITE# statements could have been used to write data to the file instead of the PRINT# statement.

The following is an example of a program which reads data from the file created above and displays the names of all employees working in the ACCOUNTING section. In this example, the statement on line 20 checks to see whether the end of the file has been reached before each data item is read. If not, the program goes on to read the data item; otherwise, execution branches to line 60, where the file is closed and execution ends. (Note that an "Input past end" error will occur if an attempt is made to read data from a sequential file after the end of that file has been reached.)

```
10 OPEN"I",#1,"A:EMPLOYEE.DAT"
20 IF EOF(1) THEN GOTO 60
30 INPUT#1,N$,S$,D$
40 IF S$="ACCOUNTING" THEN PRINT N$
50 GOTO 20
60 CLOSE:END

Ok
RUN
UNCLE SCROOGE
LUCKY FUNG
Ok
```

5.2.4 Updating sequential files

This section discusses a method of updating sequential files.

After a sequential file has been written to a disk, it is not possible to add data to that file once the file has been closed. The reason for this is that the contents of a sequential disk file are destroyed whenever that file is opened in the "O" mode. To overcome this, the following procedure can be used.

(a) Open the existing file in the "I" mode.
(b) Open a second file on the disk in the "O" mode under a different file name.
(c) Read in data from the original file and write it to the second file.
(d) After all data included in the original file has been written to the second file, close the original file and delete it with the KILL command.
(e) Write the new information to the second file.
(f) Rename the second file using the name which was assigned to the original file, then close the file.

The result is a sequential disk file which has the same file name as the original file, and which includes both the original data and the new data. A sample program illustrating this technique is shown below.

```
10 ON ERROR GOTO 210
20 OPEN"I",#1,"A:EMPLOYEE.DAT"
30 'IF FILE EXISTS, WRITE IT TO "A:TEMP"
40 OPEN"O",#2,"A:TEMP"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE#1
100 KILL "A:EMPLOYEE.DAT"
110 'ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="XX" THEN 180
140 LINE INPUT "SECTION? ";S$
150 LINE INPUT "DATE OF BIRTH? ";D$
160 PRINT#2,N$;",";S$;",";D$
170 PRINT:GOTO 120
180 CLOSE
190 'CHANGE FILENAME BACK TO "EMPLOYEE.DAT"
200 NAME "A:TEMP" AS "A:EMPLOYEE.DAT"
210 IF ERR=53 AND ERL=20 THEN OPEN"O",#2,"A:EMPLOYEE.DAT"
220 ON ERROR GOTO 0
```

The example above illustrates use of the LINE INPUT# statement to read data items consisting of character strings which include commas. As indicated in the description of the LINE INPUT# statement, this statement reads all string data preceding a carriage return as one item (quotation marks and commas are not recognized as delimiters).

Also note that the contents of the original file could be changed by replacing the contents of variables before writing them to the second file. This could be done by adding the following sequence between lines 60 and 70.

```
61 PRINT A$
62 PRINT"CHANGE ENTRY (Y/N)?":YN$=INPUT$(1):IF YN$
="Y" THEN 63 ELSE IF YN$="N" THEN 70 ELSE 62
63 INPUT"ENTER NEW NAME";NN$
64 INPUT"ENTER NEW SECTION";SS$
66 INPUT"ENTER NEW DATE OF BIRTH";DD$
67 A$=NN$+","+DD$+","+DD$
```

5.3 Random Files

More program steps are required to create and access random files than is the case with sequential files; however, random files have two advantages which make them more useful when there are large quantities of data which must be frequently up-dated. The first is that random files require less disk space for storage because data is recorded using a packed binary format, whereas sequential files are written as series of ASCII characters. The second advantage is that random files allow data to be accessed anywhere on the disk; it is not necessary to read through each data item in sequence, as is the case with sequential files. Random access is made possible by storing and accessing data in distinct, numbered units called records.

The statements and functions which are used with random files are as follows.

```
OPEN, FIELD, LSET/RSET, GET
PUT, CLOSE, LOC, LOF
MKI$, CVI
MKS$, CVS
MKD$, CVD
```

5.3.1 Creating random access files

The steps required to create random files are as follows.

(1) Open the file in the "R" mode.

Example `OPEN "R",#1,"STOCKLST.DAT",50`

This example specifies a record length of 50 bytes.
Records consisting of 128 bytes are assumed if the record length is omitted.

(2) Using the FIELD statement, allocate space in the random file buffer for variables which are to be written to the random file.

Example `FIELD#1 10 AS S$,30 AS N$,10 AS C$`

(3) Use the LSET or RSET statements to load data into the random file buffer. Note that numeric values must be converted to strings before they are placed in the buffer; this is done using the MKI$, MKS$, and MKD$ functions.

Example
```
LSET S$=MKI$(S%)
LSET N$=A$
RSET C$=MKS$(C!)
```

(4) Write data to the file from the random file buffer with the PUT statement.

Example PUT #1,S%

The following program example allows data to be input from the keyboard for storage in a random access file. In this example, one record is written to the file output buffer each time the PUT statement on line 90 is executed. The record number which is used by the PUT statement is that which is input at line 30.

```
10 OPEN"R",#1,"A:STOCKLST.DAT",36
20 FIELD #1,2 AS S$,30 AS N$,4 AS C$
30 INPUT "ENTER STOCK NO.";S%
40 INPUT "ENTER ITEM NAME";A$
50 INPUT "ENTER COST";C!
60 LSET S$=MKI$(S%)
70 LSET N$=A$
80 LSET C$=MKS$(C!)
90 PUT#1,S%
100 GOTO 30
```

NOTE:
Once a variable name has been FIELDed, do not use it in an INPUT or LET statement. The FIELD statement assigns variable names to specific positions in the random file buffer, and using an INPUT or LET statement to store values in a FIELDed variable will cancel this assignment and reassign the names to normal string space, instead of to the random file buffer.

5.3.2 Accessing random access files

The following steps are required to access a random access file.

(1) Open the file in the "R" mode.

Example OPEN "R",#1,"STOCKLST.DAT",50

(2) Using the FIELD statement, allocate space in the random file buffer for variables which are to be read in from the random file.

Example FIELD#1 10 AS S$,30 AS N$,10 AS C$

NOTE:
If the same program both writes data to a file and reads data from it, it is often possible to use just one OPEN statement and one FIELD statement.

(3) Move the desired record into the random file buffer with the GET statement.

Example GET #1,S%

(4) Data in the random file buffer can now be used by the program. Be sure that numbers which were converted to ASCII strings for storage in the file are converted back into numeric values for use by the program; this is done using the CVI, CVS, and CVD functions.

Example PRINT CVI(S$),N$,CVS(C$)

The following sample program accesses random file "STOCKLST.DAT", created using the program example shown in paragraph 5.3.1 above. Data records are read in and displayed by entering the stock number (record number) from the keyboard.

```
10 OPEN"R",#1,"A:STOCKLST.DAT",36
20 FIELD#1,2 AS S$,30 AS N$,4 AS C$
30 INPUT "ENTER STOCK NO.";S%
40 GET#1,S%
50 PRINT USING "###";CVI(S$);:PRINT"     ";
60 PRINT USING "&";N$;:PRINT"     ";
70 PRINT USING "####.##";CVS(C$)
80 GOTO 30
```

With random files, the LOC function returns the current record number; i.e., the record number which is one greater than the number of the record last accessed by a GET or PUT statement. This function can be used to control the flow of program execution according to the total number of records which have been written to the file. For example, the following statement ends program execution if the current record number for file #1 is greater than 50.

```
IF LOC(1)>50 THEN END
```

A program which uses random access for inventory management is shown in the example below. In this program, the random access record number is used as the stock number and a menu for selecting the type of processing to be performed is displayed by 40 lines to 100. Entering one of the numbers displayed in the menu causes execution to branch to the subroutine which does the work indicated in the menu entry.

```
10 CLS
20 OPEN"R",#1,"a:STOCKLST.DAT",39
30 FIELD#1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P
$
40 PRINT "HIT ANY KEY":A$=INPUT$(1):CLS:PRINT:PRIN
T "**** MENU ****":PRINT
50 PRINT 1,"INITIALIZE FILE"
60 PRINT 2,"CREATE A NEW ENTRY"
70 PRINT 3,"DISPLAY STOCK LEVEL FOR ONE ITEM"
80 PRINT 4,"ADD TO STOCK"
90 PRINT 5,"SUBTRACT FROM STOCK"
100 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL
"
110 PRINT:PRINT:INPUT"ENTER FUNCTION";FUNCTION
120 IF (FUNCTION<1) OR (FUNCTION>6) THEN PRINT "BA
D FUNCTION NUMBER":GOTO 40
130 ON FUNCTION GOSUB 670,150,290,380,460,580
140 GOTO 40
150 'BUILD NEW ENTRY
160 GOSUB 640
170 IF ASC(F$)<>255 THEN INPUT"OVERWRITE(Y/N)";A$:
IF A$<>"Y" THEN RETURN
180 LSET F$=CHR$(0)
190 INPUT"DESCRIPTION";DESC$
200 LSET D$=DESC$
210 INPUT"QUANTITY ON HAND";Q%
220 LSET Q$=MKI$(Q%)
230 INPUT"REORDER LEVEL";R%
240 LSET R$=MKI$(R%)
250 INPUT"UNIT PRICE";P
260 LSET P$=MKS$(P)
270 PUT#1,PART%
280 RETURN
290 'DISPLAY ENTRY
300 GOSUB 640
310 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
320 PRINT USING"PART NUMBER ###";PART%
330 PRINT D$
340 PRINT USING"QUANTITY ON HAND #####";CVI(Q$)
350 PRINT USING"REORDER LEVEL #####";CVI(R$)
360 PRINT USING"UNIT PRICE $$##.##";CVS(P$)
370 RETURN
380 'ADD TO STOCK
390 GOSUB 640
400 IF ASC(F$)=25 THEN PRINT"NULL ENTRY":RETURN
410 PRINT D$:INPUT"QUANTITY TO ADD";A%
420 Q%=CVI(Q$)+A%
430 LSET Q$=MKI$(Q%)
440 PUT#1,PART%
450 RETURN
460 'SUBTRACT FROM STOCK
470 GOSUB 640
480 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
490 PRINT D$
```

```
500 INPUT"QUANTITY TO SUBTRACT";S%
510 Q%=CVI(Q$)
520 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;"IN STOCK":G
OTO 500
530 Q%=Q%-S%
540 IF Q%<=CVI(R$) THEN PRINT "QUANTITY NOW";Q%;"R
EORDER LEVEL";CVI(R$)
550 LSET Q$=MKI$(Q%)
560 PUT#1,PART%
570 RETURN
580 'DISPLAY ITEMS BELOW REORDER LEVEL
590 FOR I=1 TO 100
600 GET#1,I
610 IF ASC(F$)<>255 AND CVI(Q$)<CVI(R$) THEN PRINT
 I;D$;" QUANTITY";CVI(Q$);TAB(50);"REORDER LEVEL";
CVI(R$)
620 NEXT I
630 RETURN
640 CLS:INPUT "STOCK NUMBER";PART%
650 IF (PART%<1) OR (PART%>100) THEN PRINT "BAD PA
RT NUMBER":GOTO 640:ELSE GET#1,PART%:RETURN
660 END
670 'INITIALIZE FILE
680 INPUT"ARE YOU SURE (Y/N)";B$:IF B$<>"Y" THEN R
ETURN
690 LSET F$=CHR$(255)
700 FOR I=1 TO 100
710 PUT#1,I
720 NEXT I
730 RETURN
```

5.4 Hints for Increased Performance

(a) When MFBASIC is started, memory is automatically reserved for use as random file buffers. The amount of memory reserved equals the number of bytes specified in the /S: option (the maximum record length of random files) times the number of files specified in the /F: option (the maximum number of files which can be opened at one time). Specify 0 in the /S: option to conserve memory if random files are not to be used. Also, specify /F:<number of files> in the MFBASIC command if fewer than three files (the default value) are to be used.

(b) The default value for the /S: option is 128 bytes; i.e., a buffer of 128 bytes is used for random access files. However, since data is read from and written to disks in units of 1024 bytes, a significant increase in the speed of random access can be achieved by specifying 1024 in the /S: option when MFBASIC is started and OPENing random files with a buffer length of 1024 (although the benefit will be of little consequence if only one file is open at a time).

(c) Sequential files use a 128-byte buffer; however, it is possible to use pseudo-sequential access with files opened in the "R" mode to achieve the benefits of faster access which are provided by using 1024-byte random file buffers.

For example, the following sequence could be used to create an ordinary sequential file containing 500 items.

```
10 'EXAMPLE 1
20 OPEN"O",#1,"A:FILE1"
30 A=TIME
40 FOR I=1 TO 500
50 PRINT #1,STR$(I)
60 NEXT
70 PRINT "TIME REQUIRED:";TIME-A;"SECONDS"
80 CLOSE
Ok
RUN
TIME REQUIRED: 5 SECONDS
Ok
```

To copy this file to another without making any changes in it, the following sequence could be used.

```
10 'EXAMPLE 2
20 OPEN"I",#1,"A:FILE1"
30 OPEN"O",#2,"A:FILE2"
40 A=TIME
50 IF EOF(1) THEN 90
60 INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 PRINT "TIME REQUIRED:";TIME-A;"SECONDS"
100 CLOSE
Ok
RUN
TIME REQUIRED: 26 SECONDS
Ok
```

With pseudo-sequential I/O, the original file is created by using a file opened in the "R" mode with a 1024-byte buffer; an example is shown below.

```
LIST
10 'EXAMPLE 3
20 ON ERROR GOTO 120
30 OPEN "R",#1,"A:FILE1",1024
40 A=TIME
50 FOR I=1 TO 500
60 PRINT #1,STR$(I)
70 NEXT
80 PRINT#1,"//end"
90 PRINT "TIME REQUIRED:";TIME-A;"SECONDS"
100 ON ERROR GOTO 0:PUT#1:CLOSE
110 END
120 IF ERR=50 THEN PUT #1:RESUME 60
Ok
RUN
TIME REQUIRED: 6 SECONDS
Ok
```

In the example above, items are loaded into the buffer by executing the PRINT# statement. This does not write data to the disk, so a ''FIELD overflow'' error (error code 50) occurs when the buffer becomes full. When this happens, execution branches to the error processing routine on line 120, which PUTs the contents of buffer #1 to the disk (thereby clearing the buffer), then goes back to line 60 to place the item causing the ''FIELD overflow'' error in the buffer. This is repeated until all 500 items have been placed in the buffer, after which ''//end'' (a pseudo end-of-file code) is loaded into the buffer and the buffer is written to the disk for the last time. As can be seen, the time required for file creation is approximately the same in both examples 1 and 3.

Copying the pseudo-sequential file is accomplished in much the same manner, as shown in the example below.

```
10 'EXAMPLE 4
20 OPEN "R",#1,"A:FILE1",1024
30 OPEN "R",#2,"A:FILE2",1024
40 A=TIME
50 ON ERROR GOTO 120
60 GET #1
70 INPUT #1,A$
80 PRINT#2,A$
90 IF A$<>"//end" THEN 70 ELSE ON ERROR GOTO 0
100 PRINT "TIME REQUIRED:";TIME-A;"SECONDS":CLOSE
110 END
120 IF ERL=70 THEN GET#1:RESUME 70 ELSE PUT#2:RESUME 80
Ok
RUN
TIME REQUIRED: 7 SECONDS
Ok
```

In this example, data from the file being copied is brought into file buffer #1 with GET and passed to the program with INPUT#; data is written to the new file being created using the procedure described in example 3 above. As with the PRINT# statement, successive executions of the INPUT# statement ultimately result in a ''FIELD overflow'' error, causing execution to branch to the error processing routine on line 120. When the ''FIELD overflow'' error is due to an INPUT# statement, the GET# statement is executed to bring more data into buffer #1; when it occurs due to a PRINT# statement, PUT is executed to write data accumulated in buffer #2 to the new file.

As can be seen, this method requires only about half the time consumed in copying a file with ordinary sequential access.

5.5 Precautions on Changing Flexible Disks

Before removing a flexible disk from a drive, be sure that all currently open files on that disk are closed. The reason for this is that, when disk files are opened in the "O" or "R" mode, output data (placed in the file output buffer by statements such as PRINT # or PUT) may not actually be written to the disk until the file is closed. Therefore, the contents of the file may not be complete if the disk is removed before closing all files, and there is no assurance that the file will be usable. Further, if another disk is inserted before closing files which are open, the contents of that disk may be destroyed when BDOS tries to write data to it when files are closed.

To the maximum extent possible, CP/M is designed to prevent data from being destroyed in this manner. This protection is provided by a function which automatically inhibits data from being written to a drive when the disk in that drive is replaced with another one. A "Disk write protect" error will occur if an attempt is made to write data to a drive while it is in this condition.

The RESET statement is included in MFBASIC to make it possible to reenable writes to the drive. However, note that execution of the RESET statement reinitializes the disk system, resulting in loss of the contents of any file data which is pending output at the time.

Therefore, it is recommended that the following procedures be observed whenever disks are replaced.

- **In the direct mode**

Execute a CLOSE statement.
Replace the disk.
Execute a RESET statement.

- **In the program execution mode**

```
100 CLOSE
110 PRINT "REPLACE DISK THEN PRESS ANY KEY"
120 A$=INPUT$(1)
130 RESET
```

5.6 Error Messages

The following errors may occur during access to disk files.

(a) Disk read error
This error message is issued when an error occurs while a disk file is being read.

(b) Disk write error
This error message is issued when an error occurs while a disk file is being written.

(c) Device unavailable
This error message is issued if an attempt is made to access a file in a drive which does not contain a flexible disk or if the specified disk drive is not connected.

(d) Disk write protect
This error message is issued when an attempt is made to write data to a disk to which a write protect tab has been affixed. It also occurs when an attempt is made to write data to a file after the disk in the applicable drive has been replaced with another one without executing the RESET command, or when an attempt is made to write data to a file which has been made Read Only with the SET command or STAT command of CP/M.

5.7 Error Processing

This section describes procedures to be observed in handling errors which occur during access to disk files.

(a) Errors occurring when a file is opened
Reexecute the OPEN statement after determining and eliminating the cause of the error.

(b) Errors occurring during output
Files should be closed immediately if a ''Device unavailable'', ''Disk write protect'', or ''Disk write error'' occurs during output to the disk with the PUT statement (for random access files) or statements such as PRINT # (for sequential files). The reason for this is that the contents of the file may be destroyed if output is continued without first closing and reopening the file.

(c) Errors occurring during input
Files should be closed immediately if a ''Device unavailable'' or ''Disk read error'' occurs while a file is being input using statements such as GET or INPUT #. Otherwise, there is a possibility that invalid data will be input.

(d) Errors occurring when files are closed
If an error occurs when the CLOSE statement is executed for one file, that file will be closed but there is a possibility that its contents will be destroyed.

If an error occurs during execution of a CLOSE statement for more than one file, there is a possibility that some files will not be closed. In such cases, the CLOSE statement should be executed repeatedly until the error no longer occurs.

(e) ''Disk write protect'' errors
When this type of error occurs because the disk in the accessed drive has been replaced, the write protect condition can be cleared by executing the RESET statement. However, note that execution of this statement will close all files which are currently open for output, and that this may result in another ''Disk write protect'' error. For this reason, the CLOSE statement (without file number specifications) must be executed repeatedly until no errors occur before executing the RESET statement.

5.8 Disk protection

This section describes procedures to be observed to protect flexible disks and their contents from accidental destruction.

(a) Write protect tabs
The square notch on the left side of the envelope containing a flexible disk is called a write protect notch. This notch is checked by the computer to determine whether or not data can be written to the disk when output statements such as PRINT #, PUT, or SAVE are executed. When the notch is open, executing a RESET statement as described in the section above makes it possible to write data to the disk; when the notch is covered with one of the silver-colored write protect tabs provided with flexible disks, writes to the disk are inhibited.

Covering the write protect notch with a write protect tab is a good method of protecting the disk's contents from being accidentally destroyed through inadvertent execution of a FORMAT or KILL command. Therefore, it is recommended that these tabs be used to protect the contents of disks which contain important files.

(b) Precautions concerning handling of flexible disks
Flexible disks are coated with a magnetic material which can be easily damaged through careless handling. The QX-10 Operation Manual contains a list of precautions which should be observed for handling and storage of flexible disks; be sure to read and observe those instructions.