Chapter 6

Peripheral Devices: Tape Cassette Drives, Diskette Drives and Printers

A computer system contains more than a keyboard, a screen, and the computer itself. To avoid keying in a program every time you want to run it, you will store the program on a floppy disk or magnetic tape cassette. As described in Chapter 2, you can then load the program into memory and run it, thus avoiding repeated key entry.

You will also store data on magnetic tape cassettes or floppy disks. Consider a mailing list program. This program will be stored on a cassette or floppy disk. A mailing list program is used to create a list of names and addresses. The list of names and addresses is also stored on cassette or floppy disk. Later the names and addresses are read off the cassette or floppy disk in order to print mailing labels. But that requires a printer.

Most computer systems include a line printer. Line printers are used to print output, such as mailing labels. Also, a line printer is indispensable if you want to write programs. The most efficient way of changing or correcting a program is to print a listing of the program as it currently exists, mark intended changes to this printed listing, then enter the changes that you have written down.

In this chapter we are going to describe CBM BASIC program logic needed to handle cassette drives, floppy disk drives, and printers.

CBM computers have an IEEE 488 bus connector. This is an industry standard bus which is used by the floppy disk drives and the printer. The IEEE 488 bus is also used by instruments and sundry electronics in industrial applications. Although we describe floppy disk drives and printers, this book does not describe the IEEE 488 bus itself, or programming required by any instruments connected to it.*

* To learn about the IEEE bus, see *PET and the IEEE 488 Bus (GPIB)* by Eugene Fisher and C. W. Jensen, Osborne/McGraw-Hill, 1980.

STORING DATA ON MAGNETIC SURFACES

THE CONCEPT OF A FILE

Information is stored as "files" on cassettes or diskettes.

In order to understand the concept of a "file," think of a bookshelf. The cassette or diskette is the bookshelf; each book on the shelf is equivalent to a file.

To a computer user, a "file" is a very simple concept. When you "open" a file, all information stored within the file becomes accessible. The information remains accessible until you "close" the file. This is much like taking a book down off the bookshelf and opening it up. But unlike the book, writing to a file is as easy as reading from it. When the computer writes a program or data to a cassette or diskette, it creates a new file, or it adds to an old file.

A file can have any size, limited only by the capacity of the cassette or diskette. You can create a new file and put nothing in it, in which case the file is empty. This is equivalent to having a book with covers, but no content. A file must fit on a single tape cassette or floppy disk, therefore the maximum size of a file depends on the storage capacity of the cassette or floppy disk.

You can have up to 256 files per diskette; there is no limit per cassette. Of course, if many files are stored on a single diskette or cassette, the individual files will have to be very short.

The amount of memory in your CBM computer has no impact on the size of a data file. A data file may be much larger than available computer memory. Having "opened" a data file, you can read one character from it, or as much information as will fit in the available computer memory. When writing to a data file, information that you output from computer memory can be, and usually is, added to data already stored on the cassette or floppy disk.

Program Files

There are two types of files: program files and data files. A program file, as its name would imply, contains program statements.

You create a program file whenever you SAVE a program on diskette or cassette. You read a program file when you LOAD a program into memory. These operations were described in Chaper 2.

Every file can have a name assigned to it; the name which you assign to a program file will become the name of the program. **CBM computers recognize file names of up to 128 characters, but only the first 16 characters are displayed. Disk file names must have 16 or fewer characters. Therefore, it is a good idea to restrict all file names to 16 characters or less.**

The amount of memory in your CBM computer does affect the maximum size of a program file. This is because you create a single program file when you SAVE a program on cassette or diskette. When you load a program into memory, you load the entire contents of a program file. You cannot load part of a program file into memory. Therefore the maximum size of the program file must be less than the program memory capacity of your CBM computer. If you have a very long program, and it will not fit in the available computer memory, you can break it up into a number of files, each of which will fit in the available memory space. When each section of the program completes execution, you simply load the next section into memory and run it; in this

fashion you get to execute the entire program. Later in this chapter we will describe the programming steps needed to execute large programs in this fashion.

An advantage of program files is that you do not need to know anything about their internal organization. When you save a program on diskette or cassette, it becomes a file which you can subsequently load back into memory. You must be able to identify the program file (via its file name or its location) in order to load the file back into memory, but that is all.

Data Files

A data file, as its name would imply, contains information which gets interpreted as data, in contrast to program statements. Data files are created, written, and read by programs.

Records and Fields

Data files are divided into "records," which in turn are subdivided into "fields."

A single field contains information which can be represented by a single variable name. Therefore a single field can contain an integer number, a floating point number, or a single string variable.

A record contains one or more fields. Records usually represent units of repeated information within the file, but this does not have to be the case.

Consider a mailing list. The entire mailing list will become a single data file. Each name and address within the mailing list will become one record within the file. If names and addresses are entered using the program described in Chapter 5, then each record will contain five fields: the name, the street, the city, the state, and the zip code. This file organization is illustrated in Figure 6-1.

A file may contain one or more records. Each record may contain one or more fields. **The number of records in a file and the maximum length of a record varies with the type of file, as described later in this chapter. However, for all practical purposes the size of a file is limited only by the capacity of the diskette.**

No restrictions are placed on the length of tape cassette records. A record can have any length that will fit on the tape cassette.

DATA TRANSFER TO AND FROM CASSETTE AND DISKETTE

A novice accessing tape or diskette data files is frequently mislead into thinking that something is wrong. One would instinctively expect the tape or disk unit to move in response to every statement that reads from the unit, or writes to it. A cassette drive should move the cassette; a disk drive should activate the diskette. Sometimes you will see such activity; at other times you will see no activity. This is because **a small amount of memory acts as a data buffer connecting the computer with the cassette or disk drive.**

When the computer reads from one of these drives, enough data is read to fill the buffer. You will see no further drive activity until a program accesses data that is not contained in the buffer.

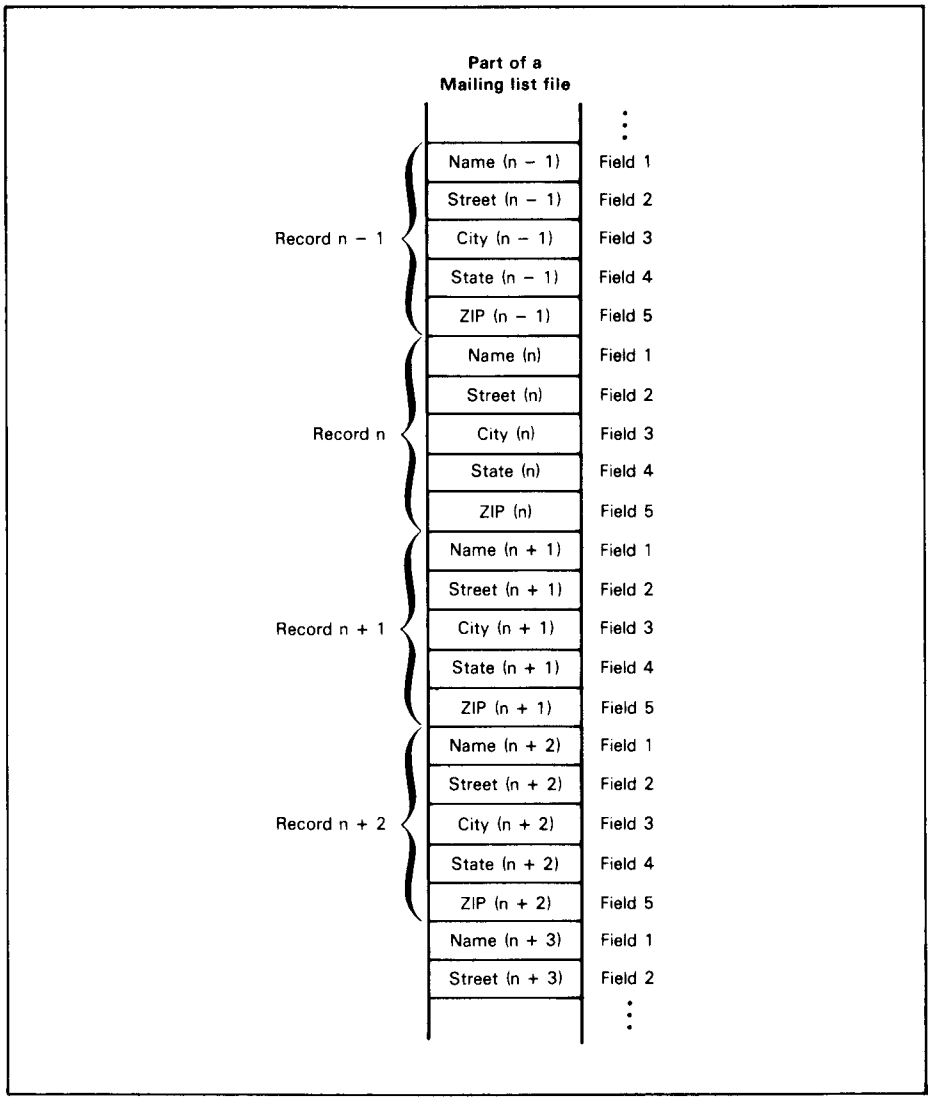


Figure 6-1. Conceptual Illustration of Four Records in a Mailing List File

Data being written to a cassette or disk drive is first written to the data buffer. As soon as the data buffer is full, all the contents of the buffer are output to the cassette or diskette, at which time you will see some activity. You will see no further activity until the buffer is again filled.

The cassette drive buffer is located in CBM computer memory. It is 192 bytes long and holds 191 bytes of data. The diskette drive buffer is located in the diskette unit itself, not in CBM computer memory. Each diskette drive buffer is 256 bytes long and holds 254 bytes of data. The diskette and cassette buffers are relatively large. In consequence, drives are inactive for much of the time while the computer is accessing drive buffers to read or write data.

Logical Files and Physical Units

We use the term "input/output programming" to describe program logic that transfers data between the computer and external physical units. Disk drives, cassette drives and printers are all external physical units.

In order to perform any input/output operation, program logic must identify the external physical unit being accessed; that will come as no surprise to you. But what about the computer end of the data transfer? This end cannot simply be specified as "the computer." Think of the problem in programming terms; programs identify data as variables or constants within BASIC statements. Therefore, the computer end of the data transfer must be specified in similar program logic terms. This concept is easy to understand if you **think of the CBM computer keyboard and display as external physical units (which in fact they are). When an INPUT statement is executed, data which you input at the keyboard gets assigned to variable(s) whose name(s) are specified as INPUT statement parameter(s).** For example, when the statement:

```
10 INPUT A
```

is executed, some number which an operator enters at the keyboard is assigned to floating point variable A. **The PRINT statement, likewise, will output variables(s) or constant(s) to the display.** Thus the PRINT statement:

```
20 PRINT A
```

takes the value assigned to floating point variable A and outputs this value to the display.

Thus the INPUT and PRINT statements have specified the computer end of the data transfer using a variable name, in this instance floating point variable A. When an INPUT statement is executed, the external physical unit is assumed to be the keyboard. When a PRINT statement is executed, the external physical unit is assumed to be the display.

Input/output programming becomes more complex when data is transferred to or from cassette drives, disk drives, printers, and external physical units other than the keyboard and display. For these more complex input/output operations you must first open a "channel" between the program and the selected physical unit. After performing required input/output operations you must close the channel. **CBM BASIC identifies individual channels using a channel number which can range between 0 and 255.**

You OPEN a channel using the CBM BASIC OPEN statement; statement parameters identify the physical unit being accessed, and the nature of the access, as illustrated in Figure 6-2. Until the channel is closed, any input or output statement need only specify the channel number in order to fully describe the nature of the input or output operation.

Every physical unit has its own, unique physical unit number. This number is used as a parameter when opening a channel in order to identify the physical unit to be accessed. Channel numbers have no equivalent permanent assignments. **Channel numbers are** therefore **frequently referred to as "logical file" numbers, or "logical unit" numbers.**

The name "logical file" describes a channel very accurately, since a channel establishes a link between a program and a data file.

Logical file numbers are a programming concept. As illustrated in Figure 6-2, you initiate any input or output operation using an OPEN statement. One of the OPEN statement parameters is a channel, or logical file number; other OPEN statement parameters identify the physical unit, the data file being accessed, and the way in which the access is to occur. After the input or output operation has gone to completion, you execute a CLOSE statement which closes down the channel. The CLOSE statement requires just one parameter: a channel or logical file number. This logical file number links the CLOSE statement to an OPEN statement. In between the OPEN and CLOSE statements, all input and output statements use a channel or logical file number to identify the device being accessed, and the way in which the device is being accessed.

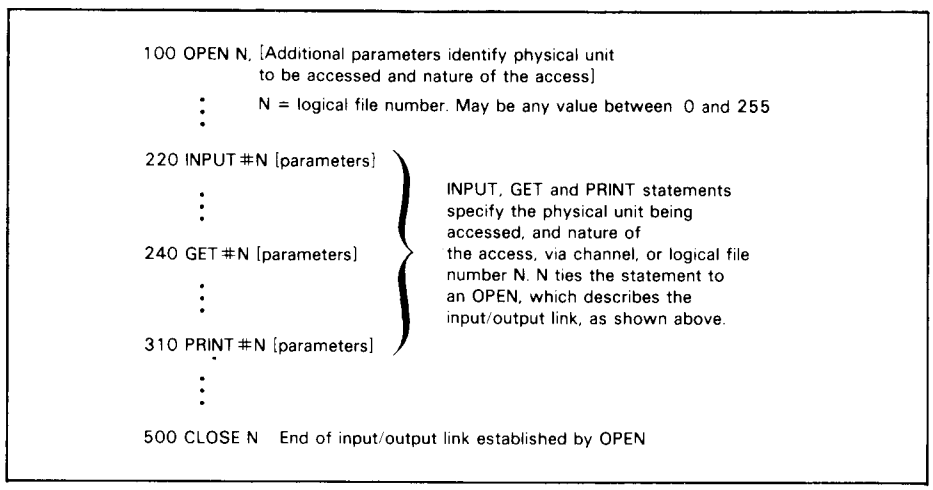


Figure 6-2. Conceptual Illustration of Logical Field Number as Used in a BASIC Program

The logical file number relates OPEN, CLOSE, INPUT, GET, and PRINT statements with each other.

Once you have used a logical file number in an OPEN statement, you cannot reuse the same logical file number to establish a different input or output channel until the logical file is closed. If you do, CBM BASIC will give you a syntax error. But otherwise no restrictions are placed on the way you assign logical file numbers within your program.

Device numbers identify the physical unit being accessed by the computer. The device number appears as a parameter in the OPEN statement. Every physical unit which can communicate with a CBM computer has a permanently assigned device number. Upon encountering a device number in an OPEN statement, the CBM computer activates appropriate electronic logic to establish communications with the specific physical unit identified by the device number. **Table 6-1 summarizes device number assignments** recognized by a CBM computer. 256 device numbers are available, ranging between 0 and 255. However, as shown in Table 6-1, only device numbers 0 through 30 are currently in use.

Table 6-1. Device Numbers with Secondary Addresses used by CBM Computers

Device	Device Number	Secondary Address	Operation Performed
Keyboard	0	None	
Cassette Drive #*	1 (Default)	0	Open for read
		1	Open for write
Cassette Drive #2	2	2	Open for write, but add End of Table mark (EOT) on close
Video Display	3	None	
Line Printer Models 2022 and 2023	4	0	Print data exactly as received
		1	Print data using previously defined format
		2	Received format to be used in subsequent formatted printout
		3	Receive lines per page specification
		4	Enable printer diagnostic message
		5	Create a special character
		6	Set spacing between lines (Model 2022 only)
Disk Drives (all models)	8	0	Load a program file to the computer
		1	Save a program file from the computer
		2-14	Unassigned
		15	Open command/status channel
Other devices connected to IEEE 488 Bus	5,6,7 and 9 through 31		Device numbers and secondary addresses are selected and assigned by the manufacturer of the device connecting to the IEEE 488 Bus.
	32 to 255 unavailable at this time		

* This is the cassette drive mounted

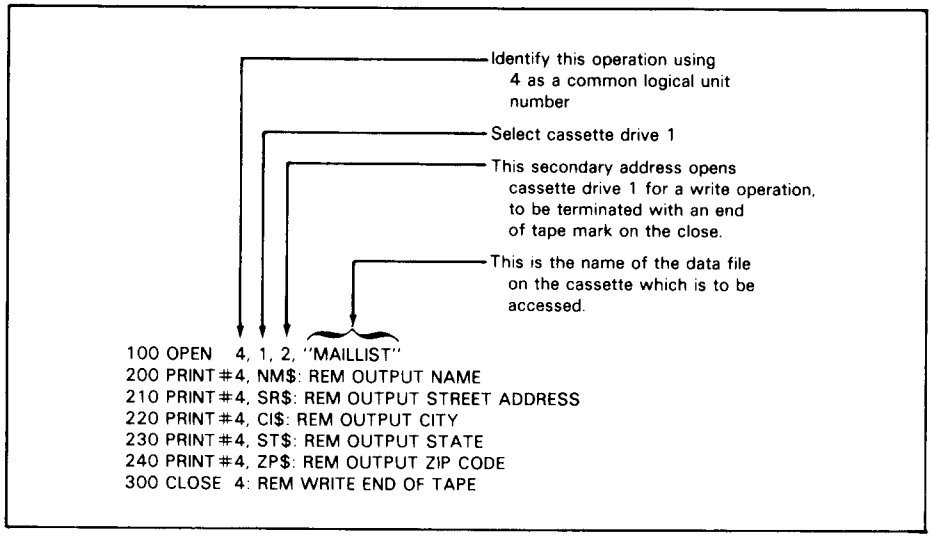


Figure 6-3. Use of Parameters by Input/Output Statements

In addition to having a device number, most physical units respond to a variety of secondary addresses. Secondary addresses are best visualized as "commands" from the computer telling the physical unit what operations it is to perform. **Secondary addresses are summarized in Table 6-1** for the physical units that are commonly connected to a CBM computer. You should not bother studying secondary addresses at this time; later when we describe input and output programming in detail, the function of secondary addresses will become obvious through their frequent use.

Figure 6-3 fully illustrates the use of parameters in input/output statements. The five PRINT# statements occurring on lines 200 through 240 write the five parts of a name and address to a file named MAILLIST, located on a cassette in cassette drive 1. On encountering each PRINT# statement, the computer knows what to do, because it checks the logical file number appearing after #. In Figure 6-3 this logical file number is 4, therefore an OPEN statement specifying logical file number 4 describes the nature of the operation; this OPEN statement occurs on line 100. If the computer could not find an OPEN statement with the required logical unit number, it would not attempt to perform the input or output operation, since it would not know what to do. In Figure 6-3 there is an OPEN statement with logical file number 4. This OPEN statement specifies physical unit number 1, therefore cassette drive 1 is selected. The secondary address is 2, therefore (on this occasion) it will be possible to write to the cassette in drive 1, but it will not be possible to read from it. Moreover, when this operation is closed, an end of tape mark will be written to the cassette, preventing any further data from being added to it. The OPEN statement specifies that the data file to be accessed has the name MAILLIST.

On line 300 there is a CLOSE statement. This CLOSE statement specifies logical file number 4, therefore everything which the OPEN statement initiated on line 100 will be terminated by the CLOSE statement on line 300. Furthermore, since the OPEN statement on line 100 specified secondary address number 2, the CLOSE statement on line 300, when executed, will cause an end of tape mark to be written to the cassette.

Thus, logical file number 4, occurring in statements on lines 200 through 300, links these statements with the OPEN statement on line 100. Additional parameters appearing in the OPEN statement on line 100 describe the operation for all of the other statements appearing on lines 200 through 300.

Physical Unit Status

Line printers can receive output from a computer. You cannot input data to a computer from a line printer. Yet there is nothing to stop you from executing an INPUT statement that references the logical file number which an OPEN statement used to initialize printer output.

Although a cassette drive can receive data from the computer, or transmit data to it, the secondary address used in the OPEN statement which initializes the cassette drive will specify either a cassette read or a cassette write operation. Nevertheless, you could erroneously execute a statement which attempts to access the cassette drive in the wrong direction.

When you execute a PRINT, GET, or INPUT statement attempting to do something which the physical unit either is incapable of handling or has not been programmed to handle, the physical unit will register an error status. A physical unit will not attempt to perform an operation that was not allowed by the OPEN statement, even if it could perform the operation. For example, if you OPEN a cassette drive for write operations only, then an INPUT or GET statement accessing the cassette unit will not execute; an error status will be generated, and that is all.

Physical units return status information following every input or output operation, whether it executes successfully or unsuccessfully. An 8-bit status is returned. **To access status, simply reference the variable ST.** For example, the statement:

```
10 X=ST
```

assigns the current status value, whatever it may be, to variable X.

Table 6-2 summarizes the way statuses are generated by all of the devices commonly attached to a CBM computer. You should refer to Table 6-2 later when writing programs that access various physical units.

Do not use status to check for keyboard or display operations, even though the keyboard and display have external device numbers.

Standard status returned by the IEEE 488 bus is shown on Table 6-2 for completeness, but interfacing to this bus is not described in this book.

CASSETTE FILE HANDLING

We are now going to describe the program steps needed to handle cassette files. We will describe how data files are created, read, and modified under program control.

Some of the file handling BASIC statements we are about to use have not yet been introduced in this book. Remember that all CBM BASIC statements are described completely in Chapter 8. If you have difficulty following any discussion in this chapter because you do not understand the BASIC statement being used, then you should go to Chapter 8, read the complete description of the statement which is giving you trouble, return to this chapter and continue.

Table 6-2. Status Byte Returned by External Devices Via Variable ST

Device Operation	Status							
	00000001 Read as 1	00000010 Read as 2	00000100 Read as 4	00001000 Read as 8	00010000 Read as 16	00100000 Read as 32	01000000 Read as 64	10000000 Read as −128
Read from Cassette drive #1 or #2	Operation OK	Operation OK	Short Block. Data block read had fewer bytes than expected	Long Block Data block read had more bytes than expected	Unrecoverable read error	Checksum error. One or more data bits read incorrectly	End of file encountered	End of tape encountered
Verify cassette drive #1 or #2					Any verify mismatch		None	
Disk drives (all models)	Receiving device not available	Transmitting device not available	None	None	None	None	End of file	Disk drive not present
IEEE 488 Bus	Time out on listener	Time out on talker					End of Identify	Device not present

You can program the CBM computer to write data onto a cassette, or to read data off the cassette, but you cannot program physical cassette movement. It is important that you understand the way cassette drives operate; otherwise, you may attempt to perform operations which the cassette drives cannot handle.

Files are stored sequentially on cassette tape. A header precedes the first file, and an end-of-tape mark follows the last file. Each file ends with an end-of-file mark.

The header is written automatically at the beginning of cassette tape when you first write to it. At this time, you may notice cassette activity which you did not expect, but otherwise the existence of the header is of no concern to you.

The computer can find files while the tape is moving forward at PLAY speed, but not at FAST FORWARD speed. An end-of-file mark identifies the end of one file. The computer can also sense an end of tape mark. **A status of 64 is returned by an end-of-file mark. A status of −128 is returned by an end of tape mark.**

The computer cannot rewind a tape nor can it detect anything on the cassette while the cassette is being rewound.

You must start cassette movement manually by pressing appropriate keys on the cassette drive when instructed to do so by the CBM computer. Do not depress any cassette drive keys before being instructed to do so via a displayed message. Subsequently, the computer will automatically stop the cassette drive at the proper time, and providing you leave appropriate keys depressed (which you should do), the computer will automatically restart the cassette drive as needed by subsequent cassette accesses.

Let us examine the impact on cassette operations of these cassette drive capabilities.

When writing data to a cassette drive, the cassette must be correctly positioned when writing begins. This is the responsibility of the CBM computer operator. Previous data on the tape under the write head will be overwritten. If the transparent tape leader is under the write head, the tape drive will start writing nevertheless, but nothing will be recorded. The safest policy is to start writing on a blank cassette, or a cassette that contains data you no longer need, and **position the cassette at the beginning of its magnetic surface;** you can then write records and files one after another until you reach the end of the cassette. The cassette drive will make sure that sufficient space is left between the end of one record or file and the beginning of the next. You do not have to, and should not, space forward on the cassette tape after writing one record or file, and before beginning the next. **You cannot back up a cassette and re-record a record or file,** since your chances of precisely rewinding the tape to the correct position are not very good. Even a small error will cause the drive to write files which you subsequently cannot read back.

When reading prerecorded data files, you must make sure that the tape is rewound to a point preceding the first file that you wish to read. The CBM computer can find any named data file while playing the tape forward, but it cannot automatically rewind the tape to find a file occurring earlier on the tape.

Never attempt to rewrite a small portion of a file that was previously recorded on tape; the operation is simply too risky. For example, suppose you have ten names and addresses stored on a tape cassette and you wish to change the fifth name and address. In theory, you could read the first four names and addresses, which would leave the tape positioned at the beginning of the fifth name and address. Then you could write a new fifth name and address over the old one. In practice, this seldom works. The cassette drives are not very precise, and there is a strong probability that you will start

writing the new name and address a little too soon or a little too late. Then a small piece of the old name and address will be left in front of the new one, or after it, but in either case you will not be able to read the new data.

To update cassette data files you *must* use two cassette drives. Read the old data off the cassette on one drive, and write the new updated data to the cassette in the other drive. You should use this procedure even if you want to change one data item among hundreds.

CBM BASIC has no statements that simply move a cassette or position it in any fashion.

PROGRAMMING CASSETTE DATA FILES

Three program steps are needed in order to access a cassette data file:

1. OPEN the data file.
2. INPUT from the data file, or PRINT to it.
3. CLOSE the data file.

OPEN a Cassette Data File

You must use an OPEN statement to open a data file. You will get a syntax error if you attempt to access an unopened data file. When opening a cassette data file, you can use any one of these OPEN statement formats:

OPEN	Open logical file N. Select the first file encountered on cassette drive 1 and allow a read operation.
OPEN N,D	Open logical file N. Select the first file encountered on device D and allow a read operation. D must be 1 for cassette drive 1, or 2 for cassette drive 2.
OPEN N,D,S	Open logical file N. Select the first file encountered on device D and allow the operation specified by secondary address S (see Table 6-1). D must be 1 for cassette drive 1, or 2 for cassette drive 2.
OPEN N,D,S,FILENAME	Open logical file N. Select the file named FILENAME on device D and allow the operation specified by secondary address S (see Table 6-1). D must be 1 for cassette drive 1, or 2 for cassette drive 2.

You can use the OPEN statement with a variety of other parameter combinations. N is the only parameter which must be present. D, if absent, is assumed to be 1. S, if absent, is assumed to be 0. If FILENAME is absent, the first file encountered is accessed.

When the OPEN statement is executed to open a tape cassette unit for a *read*, the CBM computer will display the following message if no tape control keys are pressed:

```
PRESS PLAY ON TAPE #1
OK ←———————————— A tape control key is depressed; tape begins moving.
```

The CBM computer then reads the tape header. In immediate mode the messages continue as follows (bracketed items are shown only if a filename was specified by the OPEN statement):

SEARCHING [FOR filename] **FOUND filename a** **FOUND filename b** **FOUND filename c**	Lists the first 16 characters of all files found, if any, between beginning tape position and requested file location
FOUND filename d	Format for named file
FOUND	Format for unnamed file
FOUND [filename]	Found file
READY.	File is opened for read

In program mode this block of messages is not displayed.

When the OPEN statement is executed to open a tape cassette unit for a *write*, the CBM computer displays the following message if no tape control keys are pressed:

```
PRESS PLAY & RECORD ON TAPE #1
OK ←———————————————— A tape control key is depressed; tape begins moving
```

The CBM computer writes the tape header; tape movement then stops. Here are some sample OPEN statements:

OPEN 1	Open logical file 1. No physical unit is specified, so select cassette #1, the default physical unit. No secondary address is specified, so select a read operation (the default secondary address is 0). Since no filename is specified, read from the first cassette file encountered
OPEN 1,1	Same as above, since the second parameter has the default value.
OPEN 1,1,0	Same as above, since the second and third parameters have default values
OPEN 1,1,0,"DAT"	Same as above, but the file named DAT is accessed. The second and third parameters have default values
OPEN 3,1,2	Open logical file 3 for cassette #1. Write a new file and an End of Tape mark. The new file is unnamed
OPEN 3,1,2,"PENTAGRAM"	Same as above, but give the new file the name PENTAGRAM

CLOSE a Cassette Data File

Since file opening and closing are conceptually related, for the sake of clarity we are going to describe how to CLOSE a file before describing file access program logic. But remember, CLOSE must be the last statement in the file access sequence. You cannot access a file once you have CLOSEd it.

To CLOSE a file you execute the statement:

```
CLOSE N
```

where N is the logical file number appearing as the first parameter in the OPEN statement.

When you CLOSE a cassette file after reading from it, all further read accesses are inhibited. **No harm is done if you forget to CLOSE a file after reading from it, but you are indulging in sloppy programming practices.**

You *must* CLOSE a file after writing to it. Recall that data written to the cassette file is stored in a memory buffer. Whenever the buffer is filled, buffer contents is automatically written to the cassette. Any residual, partial buffer contents is written to the cassette when you close the file. If the file is not closed for any reason, then this residual partial buffer contents will not be written out, and that can cause problems. Also, when you close a file after writing to it, an end-of-file mark is written on the tape cassette. The computer needs this end-of-file mark to separate one file from the next. Without the end-of-file mark, the computer would start reading the next file as though it were part of the previous file, and that would certainly cause errors.

When you close a cassette file after opening it with secondary address 2, an end-of-tape mark is written on the cassette. The end-of-tape mark tells the CBM computer that there is no more data on the cassette tape. If there is no end-of-tape mark, on the subsequent read the CBM computer would keep searching beyond recorded data files, and any previously recorded garbage will be interpreted as valid data, and that will generate read errors.

You do not have to execute CLOSE statements in order to close cassette data files. **The END statement closes cassette files logically but not physically. If you write to a file, you must close it with a CLOSE statement to avoid losing data.**

So why bother individually closing files that you don't write to? There are two reasons:

1. It makes you think through all file operations in a logical fashion, and that reduces programming errors.
2. A maximum of two cassette files can be open at one time.

Few programs need more than ten cassette files open at one time. However, if you do not bother to close files after accessing them, your program can finish up with a lot of open files that are no longer being used. That can cause problems, particularly in large programs which are written in small modules. If each module leaves a few files open, then ten open files can quickly accumulate, in which case the eleventh OPEN statement will cause an execution error. This is the worst kind of error to debug, since it will occur in a program which previously might have executed without error for a long time.

It takes very little program space, or execution time to CLOSE files individually after accessing them. And by doing so, you can avoid future execution errors.

CLOSE may be executed in either immediate or program mode. After writing to a file, if no tape control key is depressed when a CLOSE is issued, the CBM computer displays the following message:

```
PRESS PLAY & RECORD ON TAPE #1 <-- Press cassette keys
OK <-------------------------- Tape begins moving to write tape buffer
```

No tape control keys need to be down for a CLOSE after a READ access.

Here are some examples of CLOSE statements:

```
10   CLOSE 1        Close logical file 1
100  CLOSE 14       Close logical file 14
210  A=14           Same as above
220  CLOSE A        Same as above
```

Accessing Cassette Data Files

Having OPENed a cassette data file you can either read from it or write to it. The secondary address specified in the OPEN statement determines the allowed access. Accesses can continue until the file is CLOSEd. But remember, **whether you read from a cassette data file or write to it, you must do so sequentially.** The first cassette record written or read will always be the first record of the file. If you wish to read the tenth record of a file, you must first read records one through nine. Conversely, you cannot write the tenth record of a file without first writing records one through nine.

You must make sure that the proper tape cassette is loaded in every drive that is to be accessed by an executing program.

If you have just one cassette drive, the safest procedure is to mount the program tape in this drive, load the selected program into memory, remove the program tape and replace it with a data tape before executing the program. If you have two cassette drives, then make sure that data tape(s) are loaded in the correct drive(s). You may or may not have to remove the program tape after loading a program into memory, depending on which drive(s) the program needs for data tapes.

No cassette drive keys should be depressed prior to the first cassette access. The CBM computer will display a message telling you which keys to depress.

Remember, it is the operator's responsibility to make sure that a cassette tape is correctly positioned. The cassette drive will start writing immediately, wherever the tape happens to be positioned. When reading from tape, the drive will search forward for a data file, but it cannot find a file that has been recorded earlier on the tape.

You write data to cassette tape using the PRINT# statement:

```
PRINT#f,data
```

where:

f	is the logical file number. It must match f in the OPEN and CLOSE statements and must have a value ranging between 1 and 255.
data	is the data to be written.

PRINT# cannot be typed as ?#. PRINT# must be completely spelled out.

PRINT# transfers data to a cassette buffer in computer memory. When the cassette buffer reaches its maximum capacity of 191 data bytes, the data is written to tape as a "block." A block may contain a partial record, a single record, or several data records.

Either numbers or strings may be written to tape using the PRINT# statement.

Writing Numbers to Cassette Tape

When numbers are written to cassette tape, every number must be followed by a carriage return character.

We will write a program called NUM.PRINT# to write the numbers 1 through 10 on cassette tape.

First, the program displays a message stating its purpose, and providing load instructions:

NUM. PRINT#

```
10 PRINT"□◆◆ CREATE NUMERIC DATA TAPE ◆◆":PRINT
20 PRINT"◆◆ MOUNT TAPE, PRESS <RETURN> WHEN READY ◆◆":PRINT
30 GET A$:IF A$="" THEN 30
```

Line 20 instructs the user to insert a cassette tape in the cassette unit, rewind to the beginning of the tape, and press RETURN when ready. Statements on line 30 wait for any key to be pressed. If no keystroke is entered, the computer waits. This wait loop gives the user time to mount and rewind the cassette tape.

The wait loop created on line 30 is undesirable since it can be terminated by pressing any key. The operator's elbow brushing a key can end the wait loop, despite the instruction to press the RETURN key, which would lead an operator to the logical conclusion that no other key will do. A better wait loop is created by:

```
30 GET A$:IF A$<>CHR$(13) THEN 30
```

Once the RETURN key is pressed, the program drops down to the next line where an OPEN statement opens a cassette data file:

```
40 PRINT"◆◆ OPENING DATA FILE ◆◆":OPEN1,1,2,"NUMBERS"
```

This OPEN statement opens logical file #1, selects physical unit #1 (the cassette tape unit) with secondary address 2 (OPEN for write and EOT mark at close of file). The data file is named NUMBERS.

Next, we set up a FOR-NEXT loop to display the numbers 1 through 20 on the screen, and to write these numbers on cassette tape:

```
50 FOR N=1 TO 10
60 PRINT N ←———— Display N on screen
70 PRINT#1,N ←———— Write N to data file #1 (NUMBERS)
80 NEXT N
```

PRINT N creates a screen display. PRINT#1,N writes to tape. Remember, PRINT# cannot be typed in as ?#. PRINT must be spelled out completely, with the number sign, file number, comma, and variable following respectively.

Incorrect	Correct
?#1,N	PRINT#1,N
PRINT N	
PRINT #1,N	
PRINT#1N	
PRINT1,N	

Any of the above incorrect entries will result in a syntax error, except PRINT N, which will display N on the screen.

If everything works correctly, lines 50 through 80 display numbers on the screen and write them to tape:

PET Screen

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
```

Representation of Data Tape

The PRINT# statement writes a carriage return character on cassette tape wherever a PRINT statement would display a carriage return. Thus the PRINT# statement on line 70 writes a carriage return after outputting N, just as the PRINT statement on line 60 causes a carriage return after displaying N. **To ensure that you write numbers correctly to cassette, use PRINT# statement parameter syntax which, with PRINT statement(s), would display a single, vertical column of numbers.**

After all data is written to the tape, the file is closed. You must CLOSE the file to be certain that all data is written to cassette tape.

```
90 PRINT"◆◆ CLOSING DATA FILE ◆◆":CLOSE1
100 END
```

Be sure that the same logical file number is used in the OPEN and CLOSE statements.

```
OPEN    1,1,2,"NUMBERS"
.       ↑
.       |
.       ↓
CLOSE   1
```

Here is the complete listing for NUM.PRINT#:

```
10 PRINT"□◆◆ CREATE NUMERIC DATA TAPE ◆◆":PRINT
20 PRINT"◆◆ MOUNT TAPE; PRESS <RETURN> WHEN READY ◆◆":PRINT
30 GET A$:IF A$="" THEN 30
40 PRINT"◆◆ OPENING DATA FILE ◆◆":OPEN1,1,2,"NUMBERS"
50 FOR N=1 TO 10
60 PRINT N
70 PRINT#1,N
80 NEXT N
90 PRINT"◆◆ CLOSING DATA FILE ◆◆":CLOSE1
100 END
```

Here is a run of the program:

```
◆◆ CREATE NUMERIC DATA TAPE ◆◆

◆◆ MOUNT TAPE; PRESS <RETURN WHEN READY◆◆

◆◆ OPENING DATA FILE ◆◆

PRESS PLAY & RECORD ON TAPE #1
OK

 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
◆◆CLOSING DATA FILE ◆◆
```

Writing Strings to Cassette Tape

Unlike numbers, **when you write string variables to cassette tape, you can separate variables using a comma or a carriage return.** But the effect of these two separators differs. When string variables are subsequently read off the cassette tape, each INPUT# statement will read all string variables up to the next carriage return separator. Therefore you can use commas only to separate string variables that will always be read back as a group, via a single INPUT# statement. You must use a carriage return following the last string variable to appear in an INPUT# statement.

Special programming techniques are required in order to separate string variables using commas. Moreover, the mixed use of commas and carriage returns as separators can become a source of great confusion, even to experienced BASIC programmers. Therefore make sure that you study examples carefully before attempting to write programs for yourself.

We will modify NUM.PRINT# to write the words "ONE" through "TEN" as strings. The new program is called WORD.PRINT#. The words can be supplied using either INPUT or READ/DATA statements. Our sample program uses READ/DATA statements. The READ statement is inserted in the FOR-NEXT loop at line 60. A DATA statement is added to the end of the program. The final program is listed below, followed by a sample run of the program.

WORD. PRINT #

```
10 PRINT"□◆◆CREATE WORD DATA FILE◆◆":PRINT
20 PRINT"◆◆MOUNT DATA TAPE; PRESS <RETURN> WHEN READY◆◆"
30 GET A$:IF A$="" THEN 30
40 PRINT"◆◆OPENING DATA FILE◆◆":OPEN1,1,2,"NUMWORD":PRINT
50 FOR N=1 TO 10
60 READ N$
70 PRINT N$
80 PRINT#1,N$
90 NEXT N
100 PRINT"◆◆CLOSING DATA FILE◆◆":CLOSE1
110 DATA ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN
120 END
```

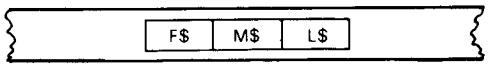
```
◆◆CREATE WORD DATA FILE◆◆

◆◆MOUNT TAPE; PRESS <RETURN> WHEN READY◆◆

◆◆OPENING DATA FILE◆◆

PRESS PLAY & RECORD ON TAPE #1
OK

ONE
TWO
THREE
FOUR
FIVE
SIX
SEVEN
EIGHT
NINE
TEN
◆◆CLOSING DATA FILE◆◆
```

As each string variable is written to cassette tape, this program terminates the string variable with a carriage return.

Let us now look at the use of commas to separate string variables that are written to cassette tape. Commas must be inserted; they are not taken from the PRINT statement parameter list. For example, when the statement:

```
10 PRINT#1,F$,M$,L$
```

is executed, contents of the three string variables F$, M$ and L$ will be concatenated into a single string variable which will be written to cassette tape as follows:



A comma can be inserted between fields using one of these two methods:

1. Enclose the separator within quotes:

```
PRINT#1,F$,",",M$,",",L$
```

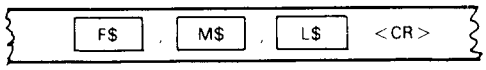
2. Use the CHR$( ) function:

```
PRINT#1,F$,CHR$(44),M$,CHR$(44),L$
```

Item Separator — Item Separator

CHR$(44) is the CHR$ function representation of the comma character.

Here is the illustration of F$, M$ and L$ written to cassette tape with commas separating F$-M$ and M$-L$:



The program below, called NAMES.PRINT#, forces separators to keep F$, M$, L$ name strings (first, middle, last) from running together:

NAMES. PRINT #

```
10 PRINT"◆◆CREATE NAME DATA FILE◆◆":PRINT
20 PRINT"◆◆MOUNT DATA TAPE, PRESS <RETURN> WHEN READY◆◆"
30 GET A$:IF A$="" THEN 30
40 PRINT"◆◆OPENING DATA FILE◆◆":OPEN1,1,2,"NAME":PRINT
50 FOR J=1 TO 4
60 INPUT F$,M$,L$
70 PRINT F$,M$,L$
80 PRINT#1,F$;CHR$(44);M$;CHR$(44);L$
90 NEXT J
100 PRINT"◆◆CLOSING DATA FILE◆◆":CLOSE1
110 END
```

The rule to follow when writing to cassette tape is that **characters written to cassette tape will be the same characters that a PRINT statement would display on the screen.** A carriage return is written to cassette tape where it would force a carriage return on the display. To create a comma separating two cassette variables, you will require the same PRINT# statement parameter list needed to display a comma between two string fields on the screen.

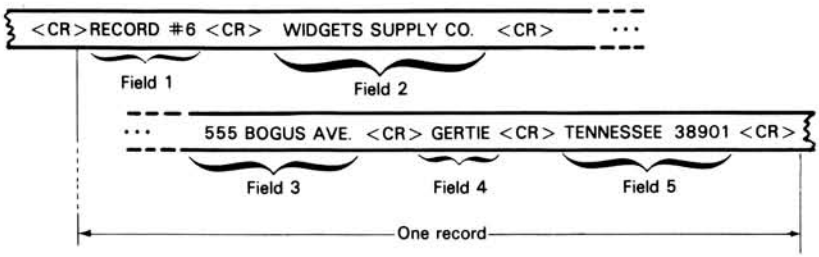
The next sample program shows how mailing list data is written to tape. A new program MAIL.PRINT# writes a mailing list named MAIL onto a cassette tape. MAIL is read by another program called MAIL.INPUT#.

In this sample program we want to demonstrate program steps needed to write cassette records. We do not want to demonstrate good data entry program design. The mailing list data entry program described in Chapter 5 illustrated good data entry program design. The mailing list program we are now about to describe has very simple (and inadequate) data entry logic, but it is short and easy to follow, allowing the discussion to focus on cassette handling.

Each name and address is written to cassette tape as one record with these five fields: 1) record number 2) name 3) street address 4) city 5) state and ZIP code. This may be illustrated as follows:

```
◆◆ RECORD #6 ◆◆       Field 1  ┐
                               │
WIDGETS SUPPLY CO.    Field 2  │
555 BOGUS AVE.        Field 3  ├ One record
GERTIE                Field 4  │
TENNESSEE 38901       Field 5  ┘
```

Of course, this is not how the data will appear on cassette tape. The data on the tape may be illustrated conceptually as follows:



Below is a program listing of MAIL.PRINT#. Type MAIL.PRINT# into your computer and save it on a cassette tape. Then list the program. (This listing assumes the standard keyboard characters.)

```
10 PRINT"□♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠"
20 PRINT"♠                              ♠"
30 PRINT"♠ MAILING LIST ENTRY ♠"
40 PRINT"♠                              ♠"
50 PRINT"♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠"
60 PRINT"■■■♠ MOUNT TAPE; <RETURN> WHEN READY ♠♠"
70 GET A$:IF A$="" THEN GOTO 70
80 PRINT"■■♠ OPENING MAIL FILE ♠♠":OPEN 1,1,2,"MAIL"
85 I=0
90 I=I+1
100 PRINT"□   ♠♠  MAILING LIST ENTRY ITEM";I;" ♠♠"
110 PRINT"                                  ■■"
120 PRINT"   (IF NO MORE ENTRIES, ENTER ";CHR$(34);"END";CHR$(34);")"
130 PRINT"■■■":INPUT "1) NAME          ";NM$
140 IF NM$="END" THEN CLOSE 1:PRINT "□";"♠♠ END OF PROGRAM ♠♠":END
150 INPUT "2) ADDR LINE 1";A1$
160 INPUT "3) ADDR LINE 2";A2$
170 INPUT "4) ADDR LINE 3";A3$
180 INPUT "■■■    ENTER FIELD # TO CHANGE (0=SAVE)";X
190 IF X=0 THEN 220
200 IF X>=1 AND X<=4 THEN GOSUB 280
210 GOTO 180
220 PRINT#1,I
230 PRINT#1,NM$
240 PRINT#1,A1$
```

```
250 PRINT#1,A2$
260 PRINT#1,A3$
270 GOTO 90
280 PRINT"■■■":ON X GOTO 290,300,310,320
290 INPUT "1) NAME          ";NM$:RETURN
300 PRINT:INPUT "2) ADDR LINE 1";A1$:RETURN
310 PRINT"■■":INPUT "3) ADDR LINE 2";A2$:RETURN
320 PRINT"■■■":INPUT "4) ADDR LINE 3";A3$:RETURN
```

The first five lines (10 to 50) display a brief description of the program function. The next segment instructs the user to mount the data tape (lines 60 and 70).

The statement on line 80 OPENs the data file:

```
80 PRINT"■■■ OPENING MAIL FILE ■■":OPEN 1,1,2,"MAIL"
```

MAIL is opened as logical file #1 on the cassette unit, with an EOT (End of Tape) mark to be written at the CLOSE of the file. The message "OPENING MAIL FILE" is displayed on the screen prior to the actual OPEN command. The operator is given this message since it takes a few seconds to open the file.

Now the tape is ready to accept data. Before data is written to the tape it should be displayed on the screen so the data may be checked for mistakes.

Statements on lines 130 through 170 input data from the keyboard and display the data on the screen.

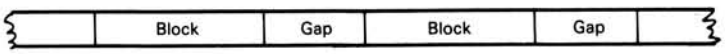
Variable "I" on line 90 is the incrementing record counter; it is displayed at line 100. Statements on lines 130 to 170 accept variables NM$ (name) and A1$, A2$, and A3$ (addresses) as separate fields. The end of each field is signaled by a carriage return. After all four fields have been entered, the statement on line 180 instructs the operator to either change a field or save the record. If a field is incorrect, the operator types the field number (1-4) and the program jumps to a field correction routine at line 280.

Using the field number input (variable X), the cursor is placed at the specified field, allowing the operator to change the selected field. The program returns to line 180 so the operator can specify another field change. When all the fields are correct, the operator inputs 0 and the program continues at lines 220 through 270. Statements on these lines write the record to the cassette data file as follows:

Be sure the logical file number referenced by the PRINT# statement is the same one specified in the OPEN statement.

After the record is saved, the program returns to line 90 to prepare for input of another record. The operator types "END" for NM$ when there are no more records to enter. The statements on line 140 close the data file and write an EOT mark (specified in the OPEN command) when NM$="END".

Notice that the tape does not move after each record is saved. As described earlier, the CBM computer stores all cassette data in a buffer. When the buffer is full, the entire buffer contents is written as a block to the tape. A block may contain a partial record, a single record, or several records. The CBM computer leaves interblock gaps between each block of data as follows:



Here is a sample program run:

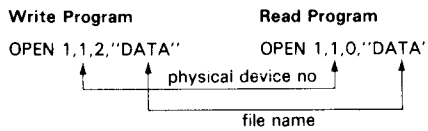
```
BREAK IN 60
READY
```

Reading Data from Cassette Tape

These are the three program steps needed to read data from cassette tape:

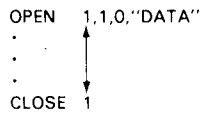
1. OPEN the data file
2. Read the data file
3. CLOSE the data file

A data file must be opened for a read with the file name it was assigned when written. A different logical file number may be assigned. The secondary address code must be 0 for the READ option.



Two statements read data from cassette tape: INPUT# and GET#. To read numeric and string fields from a data file use the INPUT# statement. The GET# statement reads one character at a time.

CLOSE the file after data has been read. CLOSE the same logical file that you OPENed.



A good way to CLOSE a file that is being read is to test for an end-of-file (EOF) via the status word (ST). When a data file is written, an EOF mark is written at the end of the file. When an EOF mark is read, the file status equals 64 and the file may be closed. You may test for an EOF mark and close the file using this one statement:

```
IF ST=64 THEN CLOSE 1
```

When ST equals 64, the file is CLOSEd.

Previously we wrote the program NUM.PRINT# to write the numbers 1 through 10 in a cassette data file named NUMBERS. Now we will write a program called NUM.INPUT# to read the ten numbers from the NUMBERS data file, and display them on the screen.

The INPUT# statement is used to read numbers and strings from cassette tape. INPUT# reads one field at a time.

The first few statements of NUM.INPUT# instruct the user to load the data tape. These statements are identical to the first three statements of NUM.PRINT#. At line 30 there is a wait loop which gives the operator time to mount the data tape. After mounting the tape, key RETURN; the program continues at the next line.

```
10 PRINT"◻◆◆ READ NUMERIC DATA TAPE ◆◆":PRINT
20 PRINT"◆◆ MOUNT TAPE ; PRESS <RETURN> WHEN READY ◆◆":PRINT
30 GET A$:IF A$="" THEN 30
```

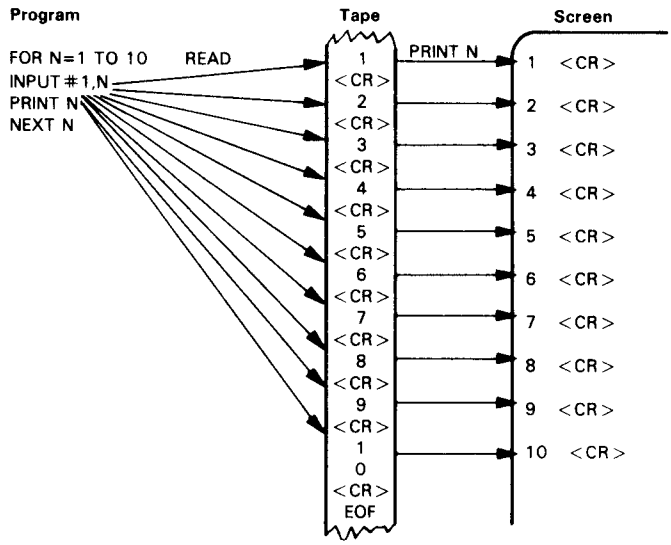
Before any data can be read, the data file must be opened. Statements on line 40 open file #1, physical device #1, with secondary address 0 (OPEN for read) and filename NUMBERS.

```
40 PRINT"◆◆ OPENING DATA FILE ◆◆":OPEN 1,1,0,"NUMBERS":PRINT
```

Next, a FOR-NEXT loop reads the first ten data items from the tape and displays them on the screen:

```
50 FOR I=1 TO 10
60 INPUT#1,N ←——— Read N from tape
70 PRINT N ←——— Print N on screen
80 NEXT I
```

The INPUT#1 statement on line 60 reads one number per execution. The FOR-NEXT loop ensures the correct number of executions. Program execution may be illustrated as follows:



After the data is read, the file must be closed.

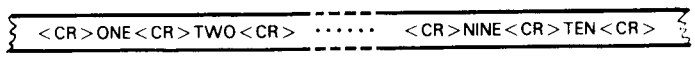
```
90 PRINT"◆◆ CLOSING DATA FILE ◆◆":CLOSE1
100 END
```

A complete listing of NUM.INPUT# is given below, followed by a sample run of the program.

NUM.INPUT#

```
10 PRINT"□◆◆ READ NUMERIC DATA FILE ◆◆": PRINT
20 PRINT"◆◆ MOUNT TAPE,PRESS <RETURN> WHEN READY": PRINT
30 GET A$: IF A$="" THEN 30
40 PRINT"◆◆ OPENING DATA FILE ◆◆": OPEN 1,1,0,"NUMBERS": PRINT
50 FOR I=1 TO 10
60 INPUT#1,N
70 PRINT N
80 NEXT I
90 PRINT"◆◆ CLOSING DATA FILE ◆◆": CLOSE1
100 END

◆◆ READ NUMERIC DATA TAPE ◆◆

◆◆ MOUNT TAPE, PRESS <RETURN> WHEN READY ◆◆

◆◆ OPENING DATA FILE ◆◆

PRESS PLAY ON TAPE #1
OK

 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
◆◆ CLOSING DATA FILE ◆◆
```

The INPUT# statement also reads fields that contain string variables. The program WORD.PRINT# wrote ten string variables to cassette tape. The data file created was named NUMWORD. NUMWORD looks like this:

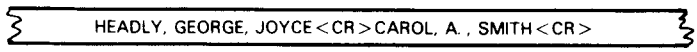


To read fields from NUMWORD, use INPUT# with a string variable parameter. With only slight modification, you can change the READ NUMERIC DATA TAPE program to read NUMWORD. The changes occur at line 40 (name the data file), and line 60 (INPUT variable). The complete changed listing appears below, followed by a sample run of the program.

```
10 PRINT"□◆◆ READ NUMWORD DATA FILE ◆◆": PRINT
20 PRINT"◆◆ MOUNT TAPE,PRESS <RETURN> WHEN READY": PRINT
30 GET A$: IF A$="" THEN 30
40 PRINT"◆◆ OPENING DATA FILE ◆◆": OPEN 1,1,0,"NUMWORD": PRINT
50 FOR I=1 TO 10
60 INPUT#1,N$
70 PRINT N$
80 NEXT I
90 PRINT"◆◆ CLOSING DATA FILE ◆◆": CLOSE1
100 END
```

```
◆◆ READ NUMWORD DATA FILE ◆◆

◆◆ MOUNT TAPE, PRESS <RETURN> WHEN READY

◆◆ OPENING DATA FILE ◆◆

PRESS PLAY ON TAPE #1
OK

ONE
TWO
THREE
FOUR
FIVE
SIX
SEVEN
EIGHT
NINE
TEN

◆◆ CLOSING DATA FILE ◆◆
```

Returning to the NAMES.PRINT# program, recall that the names in data file NAME are written as three separate string fields: F$, M$, L$. Each string field has a comma separating it from the next string field. The data tape looks like this:



If commas do not separate the fields, they will be read as a single string variable, and the three fields will be displayed on the screen as follows:

```
HEADLYGEORGEJOYCE
CAROLA.SMITH
```

A program to read data from the NAME file is listed below. The INPUT# statement on line 60 will read all fields up to the next carriage return separator. Fields lying between carriage returns are separated by commas. Since three fields lie between carriage returns, separated by commas, three string variable names appear in the INPUT# statement parameter list. The PRINT statement on line 70 displays the three string variables on a single line, with a space inserted between adjacent strings.

```
10 PRINT"□◆◆ READ NAME DATA FILE ◆◆":PRINT
20 PRINT"◆◆ MOUNT TAPE,PRESS <RETURN> WHEN READY":PRINT
30 GET A$:IF A$="" THEN 30
40 PRINT"◆◆ OPENING DATA FILE ◆◆":OPEN 1,1,0,"NAME":PRINT
50 FOR J=1 TO 4
60 INPUT#1,F$,M$,L$
70 PRINT F$;" ";M$;" ";L$
80 NEXT J
90 PRINT"◆◆ CLOSING DATA FILE ◆◆":CLOSE1
100 END

◆◆ READ NAME DATA FILE ◆◆

◆◆ MOUNT TAPE, PRESS <RETURN> WHEN READY ◆◆

◆◆ OPENING DATA FILE ◆◆

PRESS PLAY ON TAPE #1
OK
```

```
ARNOLD J. SIMPSON
BETTY S. CLARK
HEADLY GEORGE JOYCE
CAROL ANNE SMITH

◆◆ CLOSING DATA FILE ◆◆
```

The next program demonstrates how to read mailing list data which was written to data file MAIL by program MAIL.PRINT#. Each record contains five fields: record number, customer name, street, city, state and ZIP code. Below is an example of a MAIL file record:

```
◆◆ RECORD #6 ◆◆          Field 1  )
                                   )
WIDGETS SUPPLY CO.       Field 2  ) One record
555 BOGUS AVE.           Field 3  )
GERTIE                   Field 4  )
TENNESSEE 38901          Field 5  )
```

Below is a program listing of MAIL.INPUT#. Type in MAIL.INPUT# and save it on a cassette tape. Then LIST the program to follow the step-by-step discussion.

MAIL.INPUT#

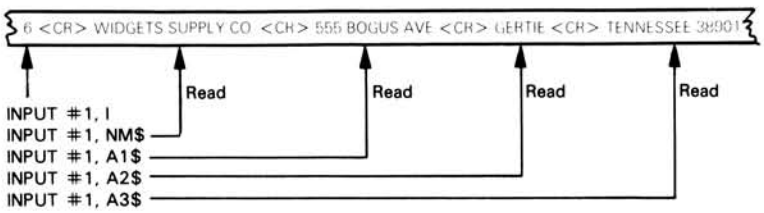
```
10 PRINT"■◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆"
20 PRINT"◆                                        ◆"
30 PRINT"◆ READ MAIL FILE W/ INPUT# ◆"
40 PRINT"◆                                        ◆"
50 PRINT"◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆":PRINT:PRINT
60 PRINT"◆◆ PRESS <RETURN> WHEN TAPE IS LOADED ◆◆■"
70 GET A$:IF A$="" THEN 70
80 PRINT"◆◆ OPENING MAIL FILE ◆◆":OPEN1,1,0,"MAIL"
90 PRINT"■◆◆ READING MAIL FILE ◆◆"
100 IF ST=64 THEN 9999
110 INPUT#1,I$
120 INPUT#1,NM$
130 INPUT#1,A1$
140 INPUT#1,A2$
150 INPUT#1,A3$
160 PRINT"■◆◆ RECORD #";I$;" ◆◆"
170 PRINT"■■■■■NAME ";TAB(9);NM$
180 PRINT"ADDR ";TAB(9);A1$
190 PRINTTAB(9);A2$
200 PRINTTAB(9);A3$
210 PRINT"■■■■"
220 INPUT"ENTER 'Y' TO READ NEXT RECORD";A$:IF A$="Y" GOTO 100
9999 PRINT"■◆◆ END OF MAIL FILE--PROGRAM TERMINATED":CLOSE1:END
```

Statements on the first five lines display a brief program description. Statements on lines 60 and 70 instruct the user to mount the data tape; the program is then ready to begin reading customer addresses. First the data file must be OPENed. MAIL is OPENed as logical file #1 on the cassette unit #1. The secondary address must be 0 for READ.

```
80 PRINT"◆◆ OPENING MAIL FILE ◆◆":OPEN1,1,0,"MAIL"
```

The statement on line 100 uses the status word (ST) to check for an end-of-file mark. If ST=64 (indicating an end-of-file mark is found), then the file is closed at line 9999. ST should be checked before data is read so that you do not attempt to read data when there is no more.

Statements on lines 110 to 150 read data using INPUT#. Each field was written to tape separated by a carriage return, so each field is read with an individual INPUT#. The variable or string names used to read data may differ from names used when the data was written. For instance, data may be written to the tape as X$ and read back from the tape as A$. The computer will not know the difference because data variable names are neither saved nor passed from one program to another.



Data is stored in the input buffer (memory) when read. Nothing is displayed on the screen unless the display is programmed. This is done by statements on lines 160 to 200, where tabs and leaders were inserted. Line 210 moves the cursor down four lines.

```
160 PRINT"■♦♦ RECORD #";I$;" ♦♦"
170 PRINT"■■■■NAME:";TAB(9);NM$
180 PRINT"ADDR:";TAB(9);A1$
190 PRINTTAB(9);A2$
200 PRINTTAB(9);A3$
210 PRINT"■■■■"
```

The screen output looks like this:

```
♦♦ RECORD #6 ♦♦

NAME:     WIDGETS SUPPLY CO.
ADDR:     555 BOGUS AVE.
          GERTIE
          TENNESSEE  38901
```

After all four fields have been displayed, the operator is asked whether the next record is desired:

```
220 INPUT"ENTER 'Y' TO READ NEXT RECORD";A$:IF A$="Y" GOTO 100
```

If the user wants the next record, the program goes to line 100 and repeats program execution until the status word (ST) signals an EOF. If the user does not wish to continue, or if an EOF is encountered, the file is closed and the program ends.

Figure 6-4 provides a flowchart of the MAIL.INPUT# program. A sample run of the program follows:

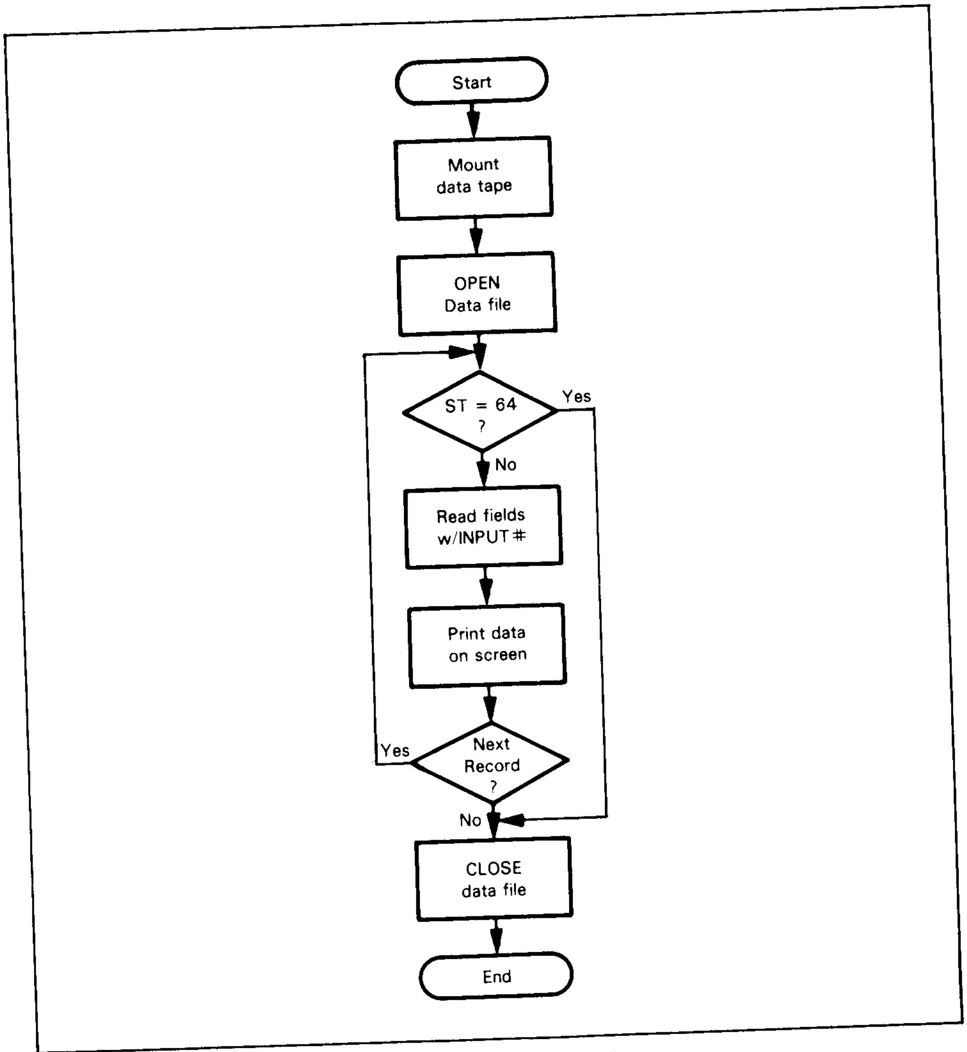


Figure 6-4. MAIL. INPUT#

```
♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠
♠                           ♠
♠ READ MAIL FILE W/ INPUT# ♠
♠                           ♠
♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠

♠♠ PRESS <RETURN> WHEN TAPE IS LOADED ♠♠

♠♠ OPENING MAIL FILE ♠♠

PRESS PLAY ON TAPE #1
OK
```

```
♠♠ READING MAIL FILE ♠♠

♠♠ RECORD # 1 ♠♠

NAME:          ACME MANUFACTURING CO.
ADDR:          1235 MAIN ST.
               DOWNTOWN
               IL  62501

ENTER 'Y' TO READ NEXT RECORD

♠♠ RECORD # 2 ♠♠

NAME:          BENJAMIN FRANKLIN
ADDR:          12 LIBERTY TOWER
               PHILADELPHIA
               PA  16524

ENTER 'Y' TO READ NEXT RECORD

♠♠ RECORD # 3 ♠♠

NAME:          NEIL ARMSTRONG
ADDR:          597 SEA OF TRANQUILITY AVE.
               EARTHVIEW
               LUNAR  000000

ENTER 'Y' TO READ NEXT RECORD

♠♠ RECORD # 4 ♠♠

NAME:          MAMMOTH DISTRIBUTION CO.
ADDR:          INDUSTRIAL PARK
               CITY OF INDUSTRY
               CA  92425

ENTER 'Y' TO READ NEXT RECORD

♠♠ RECORD # 5 ♠♠

NAME:          HENRY MUSCATEL
ADDR:          819 OAK ST.
               NAPA
               CA  95303

ENTER 'Y' TO READ NEXT RECORD

♠♠ RECORD # 6 ♠♠

NAME:          WIDGET SUPPLY CO.
ADDR:          555 BOGUS
               GERTIE
               TENNESSEE  38901

ENTER 'Y' TO READ NEXT RECORD
♠♠ END OF MAIL FILE--PROGRAM TERMINATED ♠♠
```

When you run MAIL.INPUT#, do not panic if the computer appears to stop for a few seconds. Look at the cassette drive and you will see the cassette tape moving. What is happening is that the computer is reading the next 191 bytes of data into the input buffer before continuing with the program. Once the buffer is full the computer will come to life again.

Note that **statements on line 220 do not represent good programming practice.** This program logic will cause another name and address to be read and displayed if the operator depresses the Y key. But if the operator depresses any other key, or accidentally bumps the keyboard, the program will shut down. A well-written program will respond to just two keys, perhaps "Y" for "yes" and "N" for "no". The prompt message will tell the operator to depress one of these two keys. Any other key input should be ignored. Can you rewrite the statements on line 220 to operate in this fashion?

Another method of reading data files uses the GET# statement:

```
GET #f,var
```

where:

- f — is the logical file number (1-255, matching the file number in the OPEN and CLOSE statements).
- var — is the variable name of the data to be read.

GET# reads one character at a time from the data file. It is similar to GET, which accepts one character at a time from the keyboard.

GET# reads characters, file delimiters and anything else on the tape. This is especially useful when you want to read everything that is written on a bad data tape to find the cause of any problem. GET# allows individual characters to be compared with specific values as a means of character identification.

Two sample programs will demonstrate how to read and display an entire file, including all file delimiters, and how to display the MAIL data file separated into records.

The following program, MAIL.GET#1, reads data file MAIL one character at a time and displays the contents of MAIL on the screen:

MAIL.GET #1

```
10 PRINT"♥♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠"
20 PRINT"♠                             ♠"
30 PRINT"♠  READ MAIL FILE W/ GET#     ♠"
40 PRINT"♠                             ♠"
50 PRINT"♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠":PRINT:PRINT
60 PRINT"♠♠ PRESS <RETURN> WHEN TAPE IS LOADED ♠♠"
70 GET A$:IF A$="" THEN 70
80 PRINT"■♠♠ OPENING MAIL FILE ♠♠":PRINT:OPEN1,1,0,"MAIL"
90 PRINT"■♠♠ MAIL FILE ♠♠"
100 IF ST=64 THEN 9999
110 GET#1,X$
120 IF X$=CHR$(13) THEN X$="▒"
130 PRINT X$;
140 GOTO 100
9999 PRINT"■■■♠ END OF MAIL FILE--PROGRAM TERMINATED♠♠"
     :CLOSE1:END
```

Statements on lines 10 through 90 are similar to the beginning lines of MAIL.INPUT#. These statements introduce the program, give instructions for mounting the data tape, and then open the data file.

Statements on lines 100 through 140 read data from file MAIL and display data on the screen.

The statement on line 100 checks for an end-of-file (EOF) status. If an EOF is not encountered, the next character is read by the GET# statement on line 110. #1 is the file number and X$ is the variable name assigned to the data strings. This statement will read the next character in the file.

The statement on line 120 compares the current value of X$ to a carriage return (CHR$(13)). If the value of X$ is CHR$(13), then the value of X$ is changed to a FULL GRID ■. This change avoids printing a carriage return, which would push the cursor to the next line; with the FULL GRID substituting for a carriage return, the whole file appears as one continuous line, as a good conceptual representation of the data tape. An example of this is shown in the sample run.

Make sure that a semicolon follows the variable in the PRINT statement on line 130, otherwise characters will be displayed vertically down the first column of the screen.

After each character is read from tape and displayed on the screen, the program returns to check status and GET# another character. This process repeats until ST=64 (the end-of-file). When the end-of-file is encountered at line 100, the job of MAIL.GET#1 is complete. At line 9999 the program closes the data file and ends.

Here is a sample run of MAIL.GET#1, using MAIL as the data file.

```
♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠
♠                              ♠
♠  READ MAIL FILE W/ GET#      ♠
♠                              ♠
♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠


♠♠ PRESS <RETURN> WHEN TAPE IS LOADED ♠♠

♠♠ OPENING MAIL FILE ♠♠

PRESS PLAY ON TAPE #1
OK

♠♠ MAIL FILE ♠♠

 1 ■ACME MANUFACTURING CO.■1235 MAIN ST.
■DOWNTOWN■IL    62501■ 2 ■BENJAMIN FRANKL
IN■12 LIBERTY TOWER■PHILADELPHIA    16524
■ 3 ■NEIL ARMSTRONG■597 SEA OF TRANQUILI
TY■EARTHVIEW■LUNAR    00000■ 4 ■MAMMOTH D
ISTRIBUTION CO.■INDUSTRIAL PARK■CITY OF
INDUSTRY■CA    92425■ 5 ■HENRY MUSCATEL■3
19 OAK ST.■NAPA■CA    95303■ 6 ■WIDGET SU
PPLY CO.■555 BOGUS AVE.■GERTIE■TENNESSEE
    38901■


♠♠ END OF MAIL FILE--PROGRAM TERMINATED♠♠
```

Next program MAIL.GET#2 reads MAIL and displays data on the screen, divided into records. Here is a program listing of MAIL.GET#2:

```
10 PRINT"♥♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠"
20 PRINT"♠                          ♠"
30 PRINT"♠ READ MAIL FILE W/ GET# ♠"
40 PRINT"♠                          ♠"
50 PRINT"♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠":PRINT:PRINT
65 PRINT"♠♠ PRESS <RETURN> WHEN TAPE IS LOADED ♠♠":PRINT:
70 GET A$:IF A$="" THEN 70
80 PRINT"♠♠ OPENING MAIL FILE ♠♠":PRINT:OPEN 1,1,0,"MAIL"
90 PRINT:PRINT"♥♠♠ MAIL FILE ♠♠":PRINT:
95 F=0:R=0
100 IF ST=64 THEN 9999
110 GET#1,X$
120 IF X$=CHR$(13) THEN F=F+1
130 PRINT X$;
140 IF F>=5 THEN GOSUB 160
150 GOTO 100
160 PRINT
170 R=R+1
180 IF R>2 THEN PRINT "PRESS 'Y' FOR NEXT SET OF RECORDS";:INPUT A$
185 IF A$="Y" THEN R=0
190 F=0:PRINT:RETURN
9999 PRINT"▓▓▓♠ END OF MAIL FILE--PROGRAM TERMINATED♠♠":CLOSE1:END
```

Type in MAIL.GET#2. SAVE and VERIFY the program on a cassette tape. Then LIST it.

Statements on the first ten lines (10 through 100) of MAIL.GET#2 are identical to MAIL.GET#1. This part of the program informs the user of the program's functions and procedures, and opens the MAIL data file in preparation for reading the data.

The difference between MAIL.GET#2 and MAIL.GET#1 is at line 120. If X$=CHR$(13), instead of changing the value of X$ from a carriage return to FULL GRID ■ , variable F (a carriage return counter) is incremented by +1. When MAIL.PRINT# wrote to the data file, a carriage return marked the end of each field. There are five fields in each record. MAIL.GET#2 counts fields. The conditional statement on line 140 calls a subroutine if five records have been read.

The statement on line 160 inserts a blank line between records. On line 170, variable R serves as a record counter. Statements on line 180 test to see if more than two name and address records have been read. When three records have been read, the screen is full, and the operator is asked if a new set of records is desired. If yes, the record counter R and field counter F are initialized to zero before returning to read the next set of records at line 100. This continues until the user inputs something other than a Y character or ST=64; at that time the file is closed and the program ends. Figure 6-5 illustrates program logic.

Although GET# is similar to INPUT# in some ways, it is more difficult to format the printout when using GET# if titles and indentation or spacing are desired. Just as X$ is compared with CHR$(13), so other field delimiters or characters would have to be conditionally tested in order to create a formatted display.

Following is a sample run of MAIL.GET#2 reading MAIL.

```
♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠
♠                          ♠
♠ READ MAIL FILE W/ GET# ♠
♠                          ♠
♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠
♠♠ PRESS <RETURN> WHEN TAPE IS LOADED ♠♠
♠♠ OPENING MAIL FILE ♠♠
PRESS PLAY ON TAPE #1
OK

♠♠ MAIL FILE ♠♠

 1
ACME MANUFACTURING CO.
1235 MAIN ST.
DOWNTOWN
IL    62501


 2
BENJAMIN FRANKLIN
12 LIBERTY TOWER
PHILADELPHIA
PA    16524


 3
NEIL ARMSTRONG
597 SEA OF TRANQUILITY
EARTHVIEW
LUNAR    00000

PRESS 'Y' FOR NEXT SET OF RECORDS?Y
 4
MAMMOTH DISTRIBUTION CO.
INDUSTRIAL PARK
CITY OF INDUSTRY
CA    92425


 5
HENRY MUSCATEL
819 OAK ST.
NAPA
CA    95303


 6
WIDGET SUPPLY CO.
555 BOGUS AVE.
GERTIE
TENNESSEE    38901

PRESS 'Y' FOR NEXT SET OF RECORDS?Y

♠♠ END OF MAIL FILE--PROGRAM TERMINATED♠♠
```

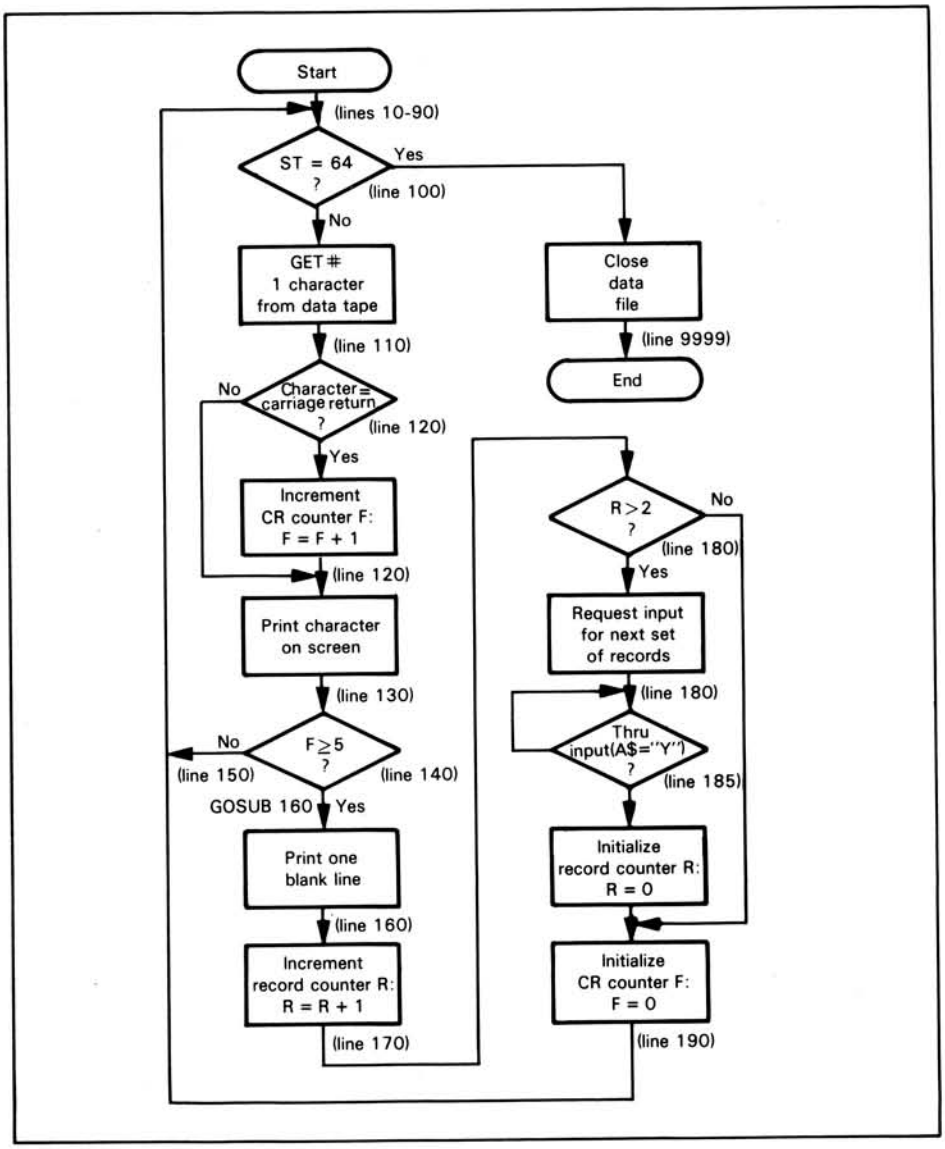


Figure 6-5. Format Printing using GET #

CASSETTE FILE FORMATS

The description of data files given at the beginning of this chapter is a conceptually accurate description of the way data is structured by computer systems in general. Data files are subdivided into records and fields. You can maintain this classical organization using appropriate CBM BASIC program logic, and we recommend that you do so. But **the actual organization of CBM cassette data files has little to do with fields and records** — as should be clear by now.

Every numeric field must be followed by a carriage return character (CHR$(13)). Therefore, a file consisting of numeric fields only could be looked upon as a sequence of numbers separated by carriage return characters. This may be illustrated as follows:

N<CR>N<CR>N<CR>N<CR>N<CR>

Nothing within the numeric file partitions fields into records, or distinguishes one record from another. It is entirely up to your program logic to keep track of records as repeating field sequences — if indeed such repeating field sequences exist.

String variables can optionally be divided into fields and records. You can use commas (CHR$(44)) to separate fields within a record, while a carriage return (CHR$(13)) follows the last field of the record. Thus, a file containing string variables only, with five fields per record, might be illustrated as follows:

<CR>S<,>S<,>S<,>S<,>S<CR>S<,>S<,>S<,>S<CR>

If you use comma and carriage return separators to divide string files into fields and records as illustrated above, then all the fields of each record must be read by a single INPUT# statement.

You are not required to use comma and carriage return separators with string variables. **You will likely be better off separating all string variable fields using carriage returns. As for numeric data, rely on program logic to group fields into records.**

Program logic needed to organize files into records and fields is usually self-evident; take the example of a mailing list. It takes no training as a programmer to see that each name and address becomes a record, while parts of the name and address must be treated as individual fields. There are a number of ways in which the parts of a name and address could be divided into fields; each option would probably do as well as any other. File organization is likely to be dictated on the needs of your program rather than the structure of CBM cassette data files. Programming difficulties, if any, will surround the PRINT# and INPUT# statement syntax.

Now we will take a simple program and, by looking at variations, identify syntax that is and is not allowed.

Key in the following program:

```
10 OPEN 1,1,1
20 FOR I=1 TO 10
30 PRINT#1,I+100
40 NEXT
50 CLOSE 1
60 STOP
70 OPEN 1
80 FOR I=1 TO 10
90 INPUT#1,J
100 PRINT J
110 NEXT
120 CLOSE 1
132 STOP
```

The OPEN statement on line 10 opens logical file 1, selecting cassette drive 1 for a write operation. **The FOR-NEXT loop on lines 20, 30, and 40 writes ten numbers to cassette tape. Numbers are followed by carriage return characters because the PRINT# statement on line 30 forces a carriage return on each execution**, just as an identical PRINT statement would cause a screen carriage return after displaying each number. The logical file is closed on line 50. Thus the ten numbers can be visualized on cassette tape as follows:

Statements on lines 70 through 120 read and display the ten numbers that were written to cassette tape by statements on lines 20 through 50.

Let us execute this program and see what happens.

Get a blank cassette tape; wind the tape forward until magnetic surface appears in front of the read gap, then mount the tape in cassette drive 1. Make sure that no cassette drive keys are depressed.

LIST the program to make sure that it is in memory and correctly entered. Now type RUN. The following message will be displayed:

```
PRESS PLAY AND RECORD ON TAPE #1
```

Depress these two keys on cassette drive 1. The CBM computer will respond by displaying OK:

```
PRESS PLAY AND RECORD ON TAPE #1
OK
```

The tape cassette will wind forward while the ten numbers 101 through 110 are written on tape cassette. After these ten numbers have been written, the drive stops moving and the following message is displayed:

```
BREAK IN 60
READY
```

The cursor flashes below the message. The STOP statement on line 60 caused the break. Now depress the STOP key on drive 1 to raise the PLAY and RECORD keys. Press the REWIND key to fully rewind the tape cassette, then press the STOP key again to raise the REWIND key. Now execute the second half of the program by typing:

```
GOTO 70
```

The message PRESS PLAY ON TAPE 1 will be displayed. Press the PLAY key on cassette drive 1. The computer will respond by displaying OK:

```
PRESS PLAY ON TAPE 1
OK
```

Nothing will happen for a while; the tape drive will move forward until the ten numbers previously written are located. Then these ten number will be displayed in a vertical column on the screen as follows:

```
101
102
103
104
105
106
107
108
109
110
BREAK IN 130
READY
```

The ten numbers are displayed in a vertical column because the PRINT statement on line 100 causes one number to be displayed per execution.

The final message is caused by execution of the STOP statement on line 130.

```
BREAK IN 130
READY
```

If you forget to rewind the tape cassette before typing GOTO 70, then the drive will search the cassette endlessly looking for data which occurred earlier on the tape. You must now stop the tape cassette and stop program execution. Rewind the tape cassette, but before you restart program execution, you will have to close file 1 in immediate mode by typing:

```
CLOSE 1
```

Then restart with:

```
GOTO 70
```

Now list the program again; end the PRINT statement on line 100 with a semicolon:

```
100 PRINT J;
```

Rewind the tape cassette; then type GOTO 70.

Once again the message PRESS PLAY ON TAPE #1 will be displayed. When you press the PLAY key, OK will follow. After a short pause **the ten numbers read off the tape cassette will be displayed on a single line as follows**:

```
101  102  103  104  105  106  107  108  109  110
BREAK IN 130
READY
```

As an experiment we will now **change statements on lines 80 through 110 so that the ten numbers are input using a single INPUT statement**, as follows:

```
10 OPEN 1,1,1
20 FOR I=1 TO 10
30 PRINT#1,I+100
40 NEXT
50 CLOSE 1
60 STOP
70 OPEN 1
80 INPUT#1,N(1),N(2),N(3),N(4),N(5),N(6),N(7),N(8),N(9),N(10)
90 FOR I=1 TO 10
100 PRINT N(I);
110 NEXT
120 CLOSE 1
130 STOP
```

Again rewind the cassette and execute the second part of the program by typing GOTO 70.

Once again you will be told to PRESS PLAY ON TAPE #1, and when you do so, ten numbers will be read from the tape cassette and displayed on a single line, as illustrated previously. Thus it makes no difference whether you read the ten numbers from tape cassette by executing one INPUT# statement with ten variables in its parameter list, or by executing one INPUT# statement, with one variable, ten times.

Experimenting further with field separation punctuation, modify the first part of the program, where data is written to the tape cassette as follows:

```
10 OPEN 1,1,1
20 FOR I=1 TO 10
30 PRINT#1,I+100
40 NEXT
45 C$=CHR$(59)
46 PRINT#1,M(1);C$;M(2);C$;M(3);C$;M(4);C$;M(5)
47 PRINT#1,M(6);C$;M(7);C$;M(8);C$;M(9);C$;M(10)
50 CLOSE 1
60 STOP
70 OPEN 1
80 FOR I=1 TO 10
90 INPUT#1,J
100 PRINT J
110 NEXT
120 CLOSE 1
130 STOP
```

CHR$(59) represents a semicolon. Rewind the tape cassette, advance the tape until magnetic surface appears below the read gap and mount the tape in the tape drive. With all keys up type RUN. When instructed to do so, press the PLAY and RECORD keys. The data will record successfully and the following message will appear.

```
BREAK IN 60
READY
▒
```

Rewind the cassette tape and type GOTO 70.

When instructed to do so, press the PLAY key on tape drive 1. Data is not read successfully; an error message is displayed.

```
FILE DATA ERROR IN 90
READY
```

You cannot use any punctuation other than carriage returns to separate numeric data fields. You can use commas or carriage returns to separate string fields. To prove this change the program as follows:

```
5 DATA ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN
10 OPEN 1,1,1
20 FOR I=1 TO 10
30 READ M$(I)
40 NEXT
45 C$=CHR$(44)
46 PRINT#1,M$(1);C$;M$(2);C$;M$(3);C$;M$(4);C$;M$(5)
47 PRINT#1,M$(6);C$;M$(7);C$;M$(8);C$;M$(9);C$;M$(10)
50 CLOSE 1
60 STOP
70 OPEN 1
80 FOR I=1 TO 10
90 INPUT#1,J$
100 PRINT J$
110 NEXT
120 CLOSE 1
130 STOP
```

Rewind the data cassette, advance the tape until magnetic surface appears below the read gap, mount the tape in drive 1 and type RUN. When instructed to do so, depress the PLAY and RECORD keys of tape 1. Data will record successfully on the cassette. When the message:

```
BREAK IN 60
READY
```

appears, rewind the cassette tape and type GOTO 70.

Press the PLAY key when told to do so. You will see the string variables 1 and 6 displayed, followed by the error message:

```
STRING TOO LONG ERROR IN 90
READY
```

What went wrong? The problem is in the INPUT# statement on line 90. An INPUT# statement will read all string fields up to the first carriage return. Therefore M$(1) through M$(5) is input on the first execution of the line 90 INPUT# statement; however, only M$(1) has its value assigned to J$ since the comma is interpreted as a field separator, not a record terminator. The second time the line 90 INPUT# statement is executed, M$(6) through M$(10) is input, since these are the fields lying between two carriage returns. Once again only M$(6) is assigned to J$, since the comma is interpreted as a field terminator. The third time the line 90 INPUT# statement is executed there is no data left to read and a file error is reported. This explains the observed display. In order to resolve the problem we must **execute INPUT# statements with the same number of variables as there were in the PRINT# statement.** Consider the following program:

```
5 DATA ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN
10 OPEN 1,1,1
20 FOR I=1 TO 10
30 READ M$(I)
40 NEXT
45 C$=CHR$(44)
46 PRINT#1,M$(1);C$;M$(2);C$;M$(3);C$;M$(4);C$;M$(5)
47 PRINT#1,M$(6);C$;M$(7);C$;M$(8);C$;M$(9);C$;M$(10)
50 CLOSE 1
60 STOP
70 OPEN 1
80 INPUT#1 N$(1),N$(2),N$(3),N$(4),N$(5)
90 INPUT#1 N$(6),N$(7),N$(8),N$(9),N$(10)
100 FOR I=1 TO 10
105 PRINT N$(I);" ";
110 NEXT
120 CLOSE 1
130 STOP
```

If you repeat the execution steps for the two halves of this program, accurately manipulating the cassette tape as described for previous executions, then when the second half of the program is executed, you will obtain the display:

```
ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE TEN
BREAK IN 130
READY
```

There are a few more experiments worth trying on your own.

Can a single INPUT# statement read a number of string variables separated by carriage returns? To check this out, change line 45 in the final program so that C$ is assigned the value CHR$(13). Then re-execute the program.

How about mixing numeric and string fields in a single data file? To check this out, create the ten string variables M$(I) as shown in the final program illustration, but

in addition, create ten numeric variables M(I) by adding the following statement on line 35:

```
35 M(I)=I+100
```

Now try various combinations of PRINT# character sequences on lines 46 and 47, and see what it takes to read these sequences back correctly with INPUT# statements on lines 80 and 90.

DISKETTE FILES

Program files and data files may be recorded on diskettes. Program files store BASIC programs. Data files store numeric and string data.

There are three types of diskette data files:

1. **Sequential files**, which store data in a very compact way, but have restricted file access capabilities.
2. **Relative files**, which require more diskette surface than sequential files to store the same amount of data, but allow data to be accessed and manipulated more efficiently.
3. **Random files**, which rely on your program logic for their structure.

Program files, sequential data files and relative data files are described in this chapter. Random data files are described in Chapter 7.

A Comparison of Diskette and Cassette File Handling

Diskette file handling differs markedly from cassette file handling for these two reasons:

1. Data can be accessed off a diskette very quickly, as compared to cassette file access times.
2. There is no "beginning" or "end" to a diskette surface, as there is to a cassette tape. A diskette drive can access any point on the diskette surface with equal ease. In contrast, cassette tape has a beginning and an end.

Cassette and diskette file handling differ markedly because they use totally different data storage formatting and access methods. Mechanical speed has very little to do with it; the speed at which a diskette is rotated is comparable to the speed at which cassette tape is moved.

Cassette tape stores data on a continuous track down the length of the tape; the cassette drive moves the tape past stationary read and write heads in order to access any part of the tape.

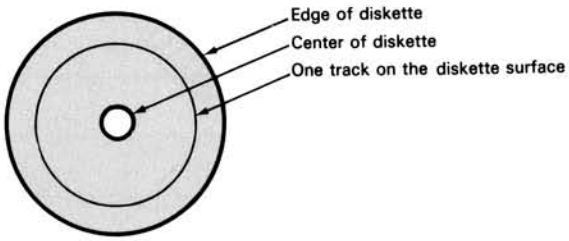
In contrast, diskettes store data on a large number of concentric circular tracks. The diskette drive read and write heads are on a moving arm that can position over any track. The diskette is rotated to bring the required section of the selected track under the read or write head.

In order to use diskettes you do not have to understand how information is stored on the diskette surface, but some knowledge will help you program diskette files more efficiently. Therefore we will begin our discussion of diskette files by describing the way data is recorded on the diskette surface.

HOW DISKETTES STORE DATA

Diskettes store data on a number of concentric tracks. Tracks are divided into sectors.

In order to imagine a single track, draw a circle to represent the diskette, then draw a smaller concentric circle to represent one track on the diskette surface. This may be illustrated as follows:



Different diskette drives write different numbers of tracks on the surface of a diskette. Some drives write on both surfaces of the diskette; other drives write on one surface only. **The CBM 2040 and 8050 diskette drives write on one surface of the diskette; as summarized in Table 6-3, the 2040 drive writes 35 tracks, whereas the 8050 drive writes 77 tracks.**

The diskette drive does not write data across the entire length of a track. To do so would make diskette surface addressing very difficult. If data were recorded over the full length of the track, no two tracks would hold the same amount of information, since no two tracks have the same length. To resolve this problem, **tracks are divided into sectors**. Every sector holds exactly the same amount of information. In the case of the 2040 and 8050 drives, 256 characters (bytes) of data are stored on each sector. Figure 6-6 illustrates this organization.

Most diskette drives write the same number of sectors on every track, even though the track closest to the edge of the diskette is much longer than the track closest to the diskette center. The 2040 and 8050 diskette drives take advantage of the longer tracks closer to the edge of the diskette by writing more sectors on longer tracks. Table 6-3 identifies the number of sectors written on various tracks. Track numbers begin at 0 for the outermost track. The innermost track has the highest track number.

If you manually rotate a CBM diskette in its cardboard jacket, you will notice a single circular hole appear in the small circular window close to the center of the cardboard jacket. A diskette with a single hole is said to be soft-sectored. In contrast, there are hard-sectored disks which have as many holes as there are sectors. CBM diskette drives can use either kind of diskette; soft-sectored diskettes are most commonly used.

Diskette Directory and Block Availability Map (BAM)

Two tracks of every diskette are used to index the diskette.

The Directory track contains the name you assign to the diskette, together with the names of all files, and their starting sector addresses.

The Block Availability Map identifies sectors which have, or have not, been allocated to files.

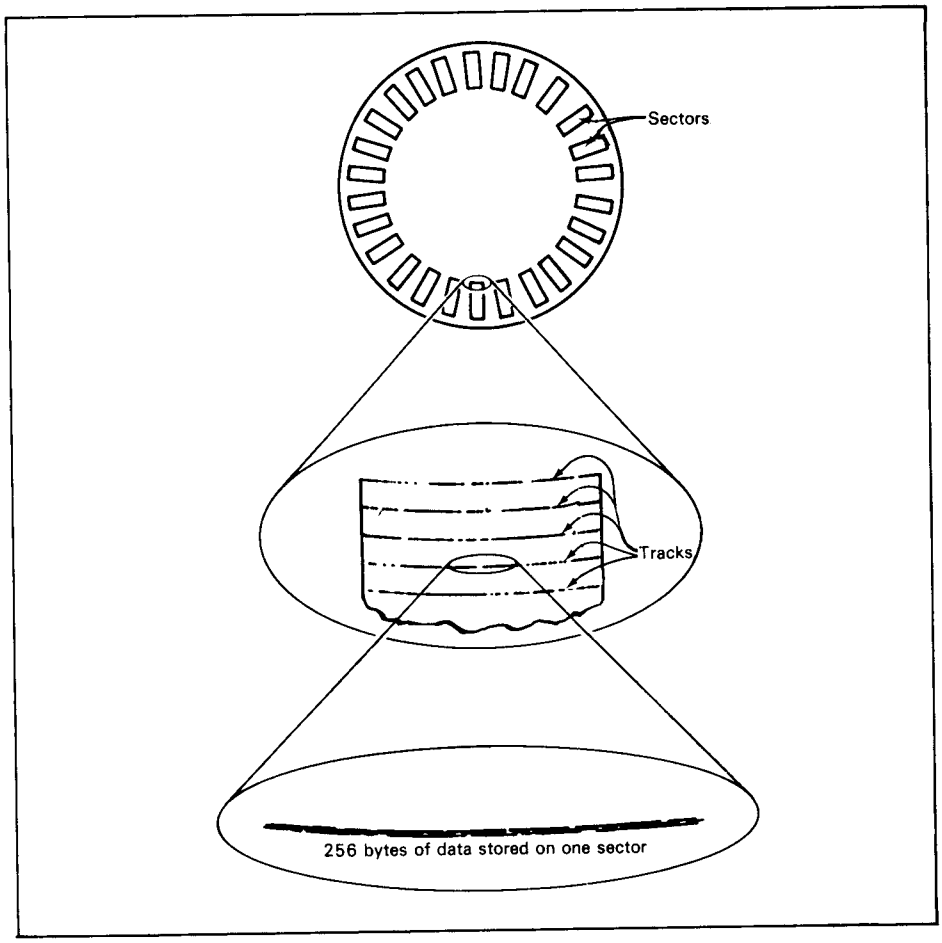


Figure 6-6. A Diskette's Recorded Surface

Files stored on cassette tape do not need a directory at the beginning of the tape. **If ten files are stored on a cassette tape, and a particular access specifies the sixth file**, having a directory at the beginning of the tape would not help the drive locate the sixth file any sooner. Since cassette files can have any length, there is no way of translating a cassette file number into a cassette tape position. You can take your chances winding the cassette tape forward to some position that precedes the file you want, thereby reducing cassette search time. Otherwise **the cassette drive must read past the first five files in order to locate the beginning of the sixth file**.

A diskette drive, in contrast, **can go directly to the beginning of any file** on the diskette surface, since every diskette sector is equally accessible. **To make this possible, every diskette has a directory** which lists the names and beginning sector addresses for all files stored on the diskette. The directory also records the file type and its current size. When a diskette data file is opened, the drive first reads the diskette directory, from which it obtains the sector address where the opened file begins. The drive can then go directly to the beginning of the opened file.

But what about the records of a diskette data file?

Table 6-3. Diskette Drive Specifications

Characteristics	2040 Drive		8050 Drive	
Total Capacity	176,640 bytes		534,272 bytes	
Usable Capacity — Sequential Files	170,180 bytes		527,812 bytes	
Usable Capacity — Relative Files			182,880 bytes	
Tracks	35		77	
Sectors per track	**Tracks**	**Sectors**	**Tracks**	**Sectors**
	0-16 17-23 24-29 30-34	21 20(or 19*) 18 17	0-38 39-52 54-65 66-76	29 27 25 23
Bytes per sector	256		256	
Total blocks (sectors)	690		2087	
Block Availability Map (RAM) track	17		38	
Directory track	18		39	
*Model 2				

Relative Data Files

All records in a relative file have the same length. It is easy to compute sector addresses for individual records of a relative file. Suppose the relative file records fit exactly two per sector. (This is unlikely to happen by chance, but it makes our illustration easy to follow.) The tenth record of this relative file will then be found on the fifth sector allocated to the file. **Relative data files are available with CBM BASIC versions 4.0 and higher, using DOS 2.0 and higher.**

Sequential Data Files

The records of a sequential file can have different lengths. We cannot compute the sector on which a particular sequential file record is to be found, since the lengths of individual sequential file records are unknown. The diskette drive can go directly to the beginning of a sequential file, since the beginning sector address is held in the diskette directory, but having gotten to the sequential file, it must access records sequentially, as a cassette drive would. For example, there is no way of reaching a sequential file's tenth record without first reading records 1 through 9. Figure 6-7 conceptually illustrates the distribution of ten records across sectors for relative and sequential files.

All versions of CBM BASIC support sequential data files.

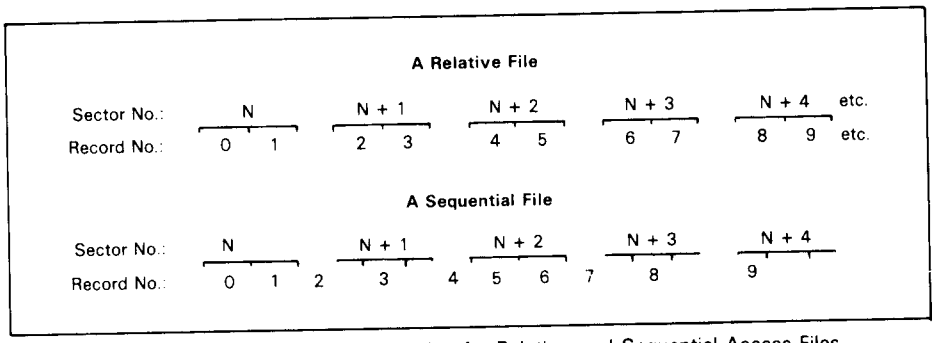


Figure 6-7. Record/Sector Correlation for Relative and Sequential Access Files

Relative versus Sequential Data Files

If sequential file records must be read sequentially, much of the diskette's random access capability is lost, so why bother with sequential files? The answer is that **sequential files store information more densely than relative files. Therefore, sequential files make better use of the diskette surface.** To illustrate this point, consider the following two names and addresses:

Cornelius J. Winkleberger
257631 Avenue of the Americas
Billinghampton
California 92804

Joe R. Smith
5 N St.
York
Iowa 50307

Suppose these two names and addresses are part of a mailing list data file. Each name and address will become one record within the data file. A relative data file must assign the same diskette space to every name and address. To avoid abbreviations the assigned diskette space must be sufficient to accommodate the longest name and address. Therefore, all shorter names and addresses will leave some space unused; and unused record space is wasted record space.

But a sequential file assigns each name and address the space it needs, however short or long this particular name and address may be. No diskette space is unused, and therefore none is wasted.

No restrictions are placed on the way you access or modify relative files. Since relative files have fixed length records that can be addressed individually, you can access a single record to read it or to change it. For example, you could rewrite the 10th name and address in a relative data file containing 20 names and addresses, leaving all other records unaltered. You can add records to a relative data file so long as the diskette has available space. You can delete any relative file record.

On the other hand **you must handle sequential files much as you would handle cassette files**. Records must be read sequentially, beginning with the first record of the file. You can append new records to the end of a sequential data file, but you cannot write new records into the middle of a sequential data file. Instead, you must rewrite the entire sequential file as a new file, modifying records in transit, as needed. **The trade-off is that sequential files make better use of the diskette surface, but they are harder to process.**

Sector Addressing

The sectors assigned to any diskette data file are unlikely to be physically sequential on the diskette surface. For example, when you add records to an existing data file, the new records may run into the beginning of the next file; therefore the file will have to be continued wherever unused sectors are available on the diskette surface. The file contracts when you erase records. Vacated sectors must be made available to other files. Therefore **diskette drive logic assumes that sectors assigned to any data file will be scattered all over the surface of the diskette. This presents no problem when dealing with sequential files.** So long as each sector points to the next sector, the drive can work its way across the diskette, sector by sector, reading the sequential file. But **for relative files** the problem is more complex, since drive logic must be able to compute addresses of individual records. **Therefore a record's displacement from the beginning of the file must be converted into a sector displacement.** Looking again at a file that has two records per sector, a request to access the 10th record becomes a request to access the 5th sector of the relative file. Since sectors are not sequential on the diskette surface, the relative file must maintain a sector index. This may be illustrated conceptually as follows:

Record number	Sequential sector number on which record begins	Actual track and sector address	
		Track no.	Sector no.
1	1	11	4
2	1		
3	2	11	5
4	2		
5	3	11	6
6	3		
7	4	13	9
8	4		
9	5		
10	5	13	10
11	6	9	3

Thus record number 6 is on the third sector assigned to the file. This sector is the sixth sector on track 11.

The term "side sector" is used to describe the relative file sector index. Currently, the 8050 diskette drive cannot use the entire diskette capacity for relative files because it runs out of space for side sectors. That is why Table 6-3 shows relative files using just 180,000 of the 8050 diskette's half million bytes. Future versions of the 8050 diskette drive will remove this restriction.

PROGRAMMING DISKETTE FILES

Different program logic is required by program files, sequential data files and relative data files. Moreover, program statements allow you to perform a variety of very necessary diskette "housekeeping" operations.

Diskette File Names

Diskette file names follow normal CBM BASIC label rules. Normally file names have 16 characters or less. Some file names are restricted to a maximum of 16 characters, but it is a good idea to observe this limit, even where it is not enforced.

DOS statements identify files via the file name. You can specify the complete file name, or you can provide the first few characters of the file name, followed by an asterisk (*) in which case the first file name encountered with matching leading characters will be selected. Here are some examples:

Specified filename: PAR*
Selected filenames:
PARITY
PARITY,SEC
PARITY,N12
PARTITION
etc.

(The first file whose name begins with PAR will be selected)

Specified filename: *
Selected filenames: Any and all, since no characters precede the *. There the first file encountered is selected.

You can also search for file names by comparing some characters, but not others. Characters that are not to be compared are specified using question marks (?). Here is an example:

Specified filename: N??,SEQ
Selected filenames:
NUM,SEQ
NXY,SEQ
NAB,SEQ
NRA,SEQ
etc.

(The first file whose name is N??,SEQ, where ? can be any character, is selected)

Instructions that specify file names can use question marks and asterisks together. Here is an example:

Specified filename: NUM??*
Selected filenames: Any filename with five or more characters, the first three being NUM. The first encountered filename is selected.

Versions of the Disk Operating System

CBM BASIC disk handling statements rely on a group of programs referred to collectively as a disk operating system (or DOS). There is very little you need to know about the disk operating system in order to use it, just as you need to know little or nothing about the BASIC interpreter in order to write BASIC programs. But you should be aware of the fact that **many CBM disk operating system versions have been released. The version is identified by a number following DOS. Currently, versions 2.1 through 2.5 are in use. These are the DOS versions we are going to describe.**

Versions of CBM BASIC

Recall that several versions of CBM BASIC are in general use. BASIC 3.0 and earlier versions were shipped with all CBM computers until March of 1980. Since then, BASIC 4.0 has been shipped on the 8000 series.

BASIC versions 1.0, 2.0 and 3.0 are very similar. As stated in the preface, we refer to these three versions of BASIC collectively as BASIC<3.0. Version 4.0 is referred to as BASIC 4.0.

BASIC<3.0 supports sequential and random files. BASIC 4.0 supports sequential, relative or random files.

BASIC 4.0 recognizes all statements from lower numbered versions of BASIC. It also has some additional disk handling statements not present in lower BASIC versions. Therefore, if your CBM computer has BASIC 4.0, you can use any disk handling BASIC statements. **The converse is not true.** For example, if your CBM computer has BASIC 1.0, you cannot use any BASIC 4.0 statements.

BASIC 4.0 does not allow the second cassette drive to be used if disk drives are present.

BASIC 4.0 disk statements assume that disk drives are the default physical unit; if no physical unit is specified, physical unit 8 is assumed. In contrast, BASIC<3.0 statements assume cassette drive 1 (physical unit 1) if no physical unit is specified.

Although BASIC 4.0 will execute all BASIC<3.0 statements, there are some file errond status incompatibilities that result when you use BASIC<3.0 file handling statements with BASIC 4.0. For example, BASIC<3.0 does not support relative files; however, if you open a file using BASIC<3.0 statements and you do not specify the file type, BASIC 4.0 will open a relative file. Also, if you execute a file operation using BASIC<3.0 statements and the file operation is illegal under BASIC<3.0, but legal under BASIC 4.0, then the error indicator will turn red at the diskette drive, the disk operation is not executed, but BASIC 4.0 will report an OK disk operation status.

OPENING A DISKETTE FILE

Twelve memory buffers in each diskette unit are used to access files on diskettes held in drives 0 and/or 1. As soon as you access any diskette file, two of these buffers are used to support overhead operations. That leaves ten buffers in each diskette unit (two drives) via which the data files themselves can be accessed.

Two buffers are needed for each open sequential file. Three buffers are needed for each open relative file. Therefore BASIC<3.0 can have up to five sequential files open simultaneously on each diskette unit (but see below). The number of files which can be held open simultaneously by BASIC 4.0 depends on the combination of sequential and random files being accessed. For each diskette unit, the following combinations are allowed:

0 Relative and 5 Sequential files
1 Relative and 3 Sequential files
2 Relative and 2 Sequential files
3 Relative and 0 Sequential files

You can increase the total number of files that can be open at one time by adding more diskette units, but only up to a point. Each open file requires a unique secondary address, and only 13 secondary addresses are available for data files.

Secondary Addresses (BASIC<3.0)

BASIC<3.0 uses 16 secondary addresses: 0 through 15. Every BASIC<3.0 OPEN statement must specify a secondary address. BASIC 4.0 automatically assigns secondary addresses.

BASIC<3.0 secondary addresses are used as follows:

1. Address 0 is used to load programs from diskette into CBM computer memory.
2. Address 1 is used to save programs from computer memory on a diskette program file.
3. Secondary addresses 2 through 14 are used to access data files. You can select any one of these secondary addresses, providing it is not being used by another OPEN data file.
4. **Secondary address 15 opens a special "command channel"** which is used to access diskette status and to perform any of the special diskette operations described later in this chapter, under "Diskette Housekeeping Operations."

The Command Channel (BASIC<3.0)

The command channel needs special mention since it is very important.

BASIC 4.0 automatically opens a command channel when any diskette file is opened. You do not have to execute any statement in order to open the command channel using BASIC 4.0.

Using BASIC<3.0 you should always OPEN the command channel before performing any diskette operation; you should leave the command channel open until you have completed all diskette operations. Use the command channel with BASIC<3.0 to interrogate diskette status, and to perform special diskette operations.

Opening Diskette Data Files (BASIC 4.0)

With BASIC 4.0 you OPEN diskette data files using the DOPEN# statement. (You can also use the OPEN statement since BASIC 4.0 includes all BASIC<3.0 statements.)

The DOPEN# statement must specify a logical file number and a file name. The diskette drive is assumed to be D0 unless you include the parameter D1 to specify drive 1.

If you specify a record length using the LX parameter, then a relative file is assumed. You can read from a relative file, or write to it; no parameter specifies a read or write operation.

If no record length is included in the DOPEN# parameter list, then a sequential file is assumed. For a sequential file you must add the parameter W if the file is to be opened for a write operation; a read operation is assumed as the default case.

The physical unit number is assumed to be 8 unless you add an ON UZ parameter. Here are some examples of BASIC 4.0 DOPEN# statements:

`10 DOPEN#1,"MAIL"`	Open logical file 1 to access a sequential file named MAIL for a read operation. The diskette is in drive 0.
`50 DOPEN#1,"MAIL",D1,W`	Open logical file 1 to access a sequential file named MAIL for a write operation. The diskette is in drive 1.
`230 DOPEN#5,"DATALIST",D0 ON U5`	Open logical file 5 to access a sequential file named DATALIST for a read operation. The diskette is in drive 0 of a diskette unit being accessed as physical unit 5.
`100 DOPEN#2,"MAIL",L100`	Open logical file 2 in order to access a relative file named MAIL. The diskette is in drive 0. If the relative file is new, then its records will each have 100 characters (bytes). If the file already exists, then it must have been assigned 100 characters (bytes) when it was first opened. Read and write accesses are both allowed.
`25 DOPEN#3,"SAMPLE",L20,D1`	Open logical file 3 to access a relative file named SAMPLE for a read or write operation. The diskette is in drive 1. If the file is being opened for the first time, then its records will have 20 characters (bytes) each. If the file already exists, then it must have been assigned 20-character (byte) records when it was first opened.

File names can be specified using a string variable instead of a string. For example, the last example could be replaced by:

```
20 S$="SAMPLE"
25 DOPEN#3,S$,L20,D1
```

Opening Sequential Diskette Data File (BASIC<3.0)

Using BASIC<3.0 you open diskette files using the OPEN statement. The OPEN statements below duplicate those DOPEN# statements shown opening sequential files above. Remember, BASIC<3.0 cannot open or handle relative files. Secondary addresses have been selected arbitrarily for the OPEN statements below.

```
10 OPEN 1,8,2 "MAIL,SEQ"
50 OPEN 1,8,7 "1:MAIL,SEQ,WRITE"
230 OPEN 5,5,3 "0:DATALIST,SEQ"
```

The string portion of the OPEN statement parameter list can be created using a string variable. For example, the OPEN statement on line 10 could be replaced by these two statements:

```
5 M$="MAIL,SEQ"
10 OPEN 1,8,2,M$
```

Here is a more complex example that replaces the OPEN statement on line 50:

```
45 M$="MAIL,SEQ"
50 OPEN 1,8,7,"1:"+M$+",WRITE"
```

File Opening Errors

These are the conditions that can cause errors when you open a data file:

1. You will get a FILE NOT FOUND error if you OPEN a new sequential data file for a read operation. The sequential file must exist, since a new file will be empty when created, and you cannot read data out of an empty file.

2. If you open an old file but specify the wrong file type, then you will get a FILE TYPE MISMATCH error. This occurs if you open an old relative file as a sequential file, or if you open an old sequential file as a relative file, or if you open a program file as any type of data file.
3. You cannot open an old sequential file for a write access. If you do, you will get a FILE EXISTS error. You can only write into new sequential data files.

Misspelling a file name in an OPEN statement is an error that can cause you a lot of trouble without generating a warning. The disk operating system will simply assume that the misspelled file is a new file. If opening the new file would otherwise be valid, no error is reported.

CLOSING A DISKETTE FILE

To close any diskette data file you execute the BASIC 4.0 statement:

```
DCLOSE#N
```

or the BASIC<3.0 statement:

```
CLOSE N
```

where N is the logical file number appearing as the first parameter in the OPEN or DOPEN# statement.

You must CLOSE a file after writing to it, otherwise some data written to the file may be lost.

You do not have to CLOSE a file after reading from it, but to do so is good programming practice.

All open files are automatically closed by the computer when you execute an END statement. (This assumes that the diskette drives are still turned on.) Nevertheless it is good programming practice to close files individually using CLOSE statements rather than using the END statement to close all files. This subject was discussed in detail earlier in this chapter for cassette data files. The discussion on closing cassette data files applies also to diskette data files.

DISKETTE ERRORS AND ERROR STATUS

There is a red warning light which acts as an error indicator on all CBM diskette drives. This error indicator lights up red when a diskette operation is not successful. No other diskette operation can be performed until this error indicator has been cleared. To clear the error indicator, stop program execution by pressing the STOP key, then read diskette error status.

It is a good idea to read status after every diskette operation, and to include status checking as a routine part of all diskette handling program logic.

Recall that you cannot write to a diskette if its write-protect slot is covered. The diskette is said to be write-protected. **If you try to copy a file to a diskette that is write-protected, then the CBM computer will hang up.** The computer will endlessly try to write, but the diskette will not send back an error status. This situation manifests itself when the computer seems to be doing nothing, but you cannot stop program execution by pressing the STOP key. When this happens, you must remove the diskette from its disk drive, turn power off at the CBM computer, then turn power on again.

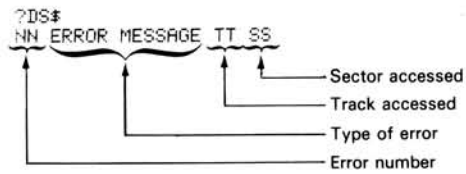
Clearing Diskette Error Status (BASIC 4.0)

Using BASIC 4.0 you can clear diskette errors in immediate mode, or in program mode.

To clear error status in immediate mode, execute an immediate mode PRINT statement to display numeric variable DS or string variable DS$.

Numeric variable DS returns status as a decimal number which should be interpreted using Table 6-2.

DS$ displays four parameters as follows:



Diskette error messages are given in Appendix B.

A program written using BASIC 4.0 should **test diskette status by referencing variable DS** as follows:

```
20 IF DS <> 0 THEN PRINT "ERROR";
```

Following any diskette error this statement will clear the error status, stop program execution and display the message:

```
ERROR
BREAK IN XXXX
```

XXXX is the line number on which the STOP statement is located. A more informative variation displays DS$ to give the operator some idea what happened. Here are the necessary statements:

```
20 IF DS <> 0 THEN PRINT DS$:STOP
```

Following any diskette error, this statement will clear the error status, stop execution and display the message:

```
NN ERROR MESSAGE TT SS
BREAK IN XXXX
READY
```

NN is the error number; see Appendix B for a summary of error numbers and what they mean. When the error occurred the sector being accessed is identified by TT (track) and SS (sector).

Diskette Errors (BASIC<3.0)

Using BASIC<3.0 you cannot access variables DS or DS$. To examine error status, **you must OPEN a logical file specifying physical unit 8 with secondary address 15. You must then input four string variables and display them.** In program mode this may be illustrated as follows:

```
10 OPEN 1,8,15
20 INPUT#1,A$,B$,C$,D$
30 PRINT A$,B$,C$,D$
40 CLOSE 1
```

The INPUT#1 statement will not execute in immediate mode.

A$, B$, C$ and D$ are the error message number (A$), the error message (B$), the track number (C$) and the sector number (D$) as illustrated above for DS$ using BASIC 4.0. A$, B$, C$ and D$ are arbitrarily selected string variable names. On lines 20 and 30 above you could use any four string variable names instead of A$, B$, C$ and D$.

When writing programs using BASIC<3.0 you should OPEN a logical file with physical unit address 8 and secondary address 15 before beginning any diskette access. Then test error status following every diskette operation by inputting the error message number. If this number is 0, the disk operation was successful. Here is necessary program logic:

```
10 OPEN 15,8,15
.
150 Disk operation statement here
160 REM TEST DISKETTE STATUS
170 INPUT#15,A$,B$,C$,D$
180 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$:ST
.
.
```

If a program contains numerous disk operations, then the statements shown on lines 170 and 180 above will reappear frequently. You may be tempted to put these statements into a subroutine. You can do so, but it will be more difficult to tell where the disk error occurred, since the STOP statement will always report a break on the same subroutine line. In contrast, if statements on 170 and 180 are repeated wherever they are needed, then by looking at the line where the break occurred, you can tell which STOP statement caused the break, and therefore which disk access caused the error.

DISKETTE HOUSEKEEPING OPERATIONS

In addition to reading and writing data files, file handling BASIC statements allow you to perform these operations:

1. Prepare a new diskette.
2. Erase an old diskette and prepare it for reuse.
3. Display a diskette's directory to see what files are stored on the diskette, and how much unused diskette space remains.
4. Check the diskette for sectors that have been allocated to a file but are still unused. Make these sectors generally available again, thereby increasing available diskette space (BASIC 4.0 only).
5. Copy a file.
6. Copy an entire diskette.
7. Rename a file (BASIC 4.0 only).
8. Delete a file from a diskette, or replace file contents.

Every BASIC<3.0 file or disk operation must begin with an OPEN statement. You can then read, write or perform one of the housekeeping operations described above. The operation must end with a CLOSE statement.

Using BASIC 4.0 you must OPEN a data file before reading from it or writing to it, and you must then CLOSE the data file. However the housekeeping operations described above are executed by special statements that do not need to be preceded by an OPEN, or followed by a CLOSE.

We will describe all of these housekeeping operations before looking at file handling program logic.

Although housekeeping operations are frequently performed in immediate mode, they can be executed in program mode.

BASIC statements used to perform housekeeping operations are described fully in Chapter 8. If you have trouble following any discussion because you do not understand a BASIC statement, read the BASIC statement description given in Chapter 8, then return and continue.

DISKETTE PREPARATION AND INITIALIZATION

You cannot take an unused diskette, load it into a disk drive and write data on it. First the diskette surface must be prepared. Sectors must be marked off on tracks, then the directory and block availability map must be written. The diskette is also assigned a name. **You can prepare a used disk; this erases all prior data and readies the diskette for reuse.**

You will usually prepare a diskette in immediate mode.

Diskette Preparation (BASIC 4.0)

Using BASIC 4.0 you prepare a new diskette using the HEADER statement, as follows:

```
HEADER "DISK NAME", DX, IYY
```

"DISK NAME" can be any string name with up to 16 characters. YY is a number which you must assign to the diskette. X is the drive number holding the diskette; it must be 0 or 1.

It takes approximately two minutes to prepare a diskette. If for any reason the diskette cannot be prepared, the following message is displayed:

```
?BAD DISK
```

This message will be displayed for any of these reasons:

1. You forgot to load a diskette into the selected drive.
2. You specified the wrong drive in the HEADER statement parameter list.
3. You forgot to specify a diskette number in the HEADER statement parameter list.
4. The diskette is write-protected (the write-protect notch is covered).
5. The diskette has a defective magnetic surface.

When preparing a used diskette you only need specify the drive number in the HEADER statement parameter list. If you specify a disk name, then it will replace the old disk name; if you do not, then the old disk name will be retained. If you specify a disk number, then it will replace the old disk number; if you do not, the old disk number will be retained. But you cannot specify a new disk number unless you also specify a new disk name. You will get a syntax error if you try it.

Recall that BASIC 4.0 assumes that the diskette drive is physical unit number 8. If for any reason you are initializing a diskette using a disk drive with a different physical unit number, then you must add this information to the HEADER statement parameter list using: ON UZ or, UZ, where Z is the physical unit number.

It takes just a few seconds to prepare a used diskette.

Below are some examples of immediate mode HEADER statements. Subsequent dialogue is not shown.

```
HEADER "SAMPLE",D0, I01
```
A diskette is prepared on drive 0. The diskette is given the name SAMPLE and the number 01.

```
HEADER D0
```
An old diskette is prepared on drive 0. The old name and diskette number are preserved.

```
HEADER "NEW", D1
```
An old diskette is prepared on drive 1. The diskette is given the new name NEW, but it retains its old diskette number.

```
HEADER "SAMPLE", D0, I05, ON U7
```
A diskette is prepared in drive 0 of a diskette drive with physical unit number 7. The diskette is given the name SAMPLE and the number 05.

```
HEADER D1, I01
? SYNTAX ERROR
```
The HEADER statement will not execute because a new diskette number has been specified without a new disk name.

Diskette Preparation (BASIC<3.0)

To prepare a diskette using BASIC<3.0 you must OPEN the diskette command channel, then execute a PRINT# statement using the logical file specified in the OPEN statement parameter list. The PRINT# statement must have the following character string enclosed in quotes:

"NEWX:DISKNAME,YY"

NEW may be replaced by N. X is the drive number; it must be 0 or 1. DISKNAME is the name which will be assigned to the diskette; it can be any valid 16 character string. YY is the diskette number.

The OPEN statement which opens the diskette command channel can specify any logical file number, but it must specify physical unit number 8 and secondary address 15.

Here are some examples of BASIC<3.0 diskette initialization statements:

```
OPEN 1,8,15
PRINT#1, "N0:SAMPLE,01"
```
A diskette is initialized in drive 0. It is given the name SAMPLE and the number 01.

```
OPEN 3,8,15
PRINT#3,"NEW1:NEW,01"
```
A diskette is initialized in drive 1 with the name NEW and the number 01.

BASIC<3.0 diskette preparation does not always work on a CBM computer that has BASIC 4.0. Sometimes the disk drive continues to spin the diskette after initialization has been completed.

BASIC<3.0 allows you to prepare an old diskette, in which case everything previously stored on the diskette is erased, and the surface is prepared for reuse. You do not have to specify a diskette number in the PRINT# parameter list when preparing an old diskette; the old diskette number will be used if no new number is specified.

Diskette Initialization (BASIC<3.0)

When using BASIC<3.0, you must initialize a diskette that has data stored on it before opening a file. To initialize the diskette you OPEN the command channel and execute a PRINT# statement with the letter "I" or the word "INITIALIZE", plus the drive number appearing as a string variable in the PRINT# statement parameter list. The drive number can be omitted, in which case diskettes in both drives will be initialized.

Diskettes are usually initialized in program mode.

When a diskette is initialized, no data on the diskette surface is changed.

Here are some examples of BASIC<3.0 diskette initialization statements:

```
10 OPEN 1,8,15                    Initialize a diskette in drive 0.
20 PRINT#1,"I0"

5 OPEN 3,8,15                     Initialize two diskettes in drives 0 and 1.
10 PRINT#1, "INITIALIZE"
```

You do not have to initialize a diskette that you have just prepared. Preparation also initializes the diskette.

DISPLAYING THE DISKETTE DIRECTORY

Displaying the Diskette Directory (BASIC 4.0)

Before accessing any diskette, it is advisable to display the diskette directory. Using BASIC 4.0 this is done using the DIRECTORY statement. The DIRECTORY statement is usually executed in immediate mode. Here are some examples of the directory statement:

```
DIRECTORY                         Display directories for diskettes in drives 0 and 1.
DIRECTORY D1                      Display directory for diskette in drive 1.
DIRECTORY D1 ON U8                This statement also displays the directory for the diskette in drive 1
                                    since the physical unit 8 is the default physical unit.
DIRECTORY D0                      Display the directory for the diskette in drive 0.
```

The word CATALOG can be used instead of DIRECTORY.
The directory is displayed as follows:

```
0     "Diskette name     "NNXX
BBBB  "Filename"  Type
BBBB  "Filename"  Type
        etc.
YYYY BLOCKS FREE
```

The diskette name and number appears at the top of the display in a reverse field. NN is the diskette number. XX is the DOS version number. Below a list of file names is displayed. These are the files recorded on the diskette. To the left of the file name is the number of blocks (sectors) assigned to the file. To the right of the file name is the file type: REL for a relative file, SEQ for a sequential file, or PRG for a program file. Finally, the number of unused blocks (sectors) is displayed. (There are also user files which are described in Chapter 7.)

There must be a diskette in every drive specified by the DIRECTORY statement. A very common error is to type DIRECTORY when you want to display the DIRECTORY for a diskette in drive 0. If there is no diskette in drive 1, then the error indicator will turn red and no directories will be displayed. Remember, you must clear the error indicator by reading diskette status (type ?DS$<CR>). You cannot use the diskette drive again until the error indicator has been cleared. You will also get an error indication if you specify the wrong drive in the DIRECTORY statement. For example, if there is a diskette in drive 1 but you enter the immediate statement:

```
DIRECTORY D0
```

then you will get an error indication, but no directory.

Displaying the Diskette Directory (BASIC<3.0)

Using BASIC<3.0 you display the directory using a LOAD statement as follows:

```
LOAD "$X",Y
```

X is the drive number (0 or 1) and Y is the physical unit number (usually 8). The dialogue that follows is standard program-loading dialogue. After the program is loaded, you list it in order to display the directory. The following example displays the directory for a diskette in drive 0.

```
LOAD "$0", 8
SEARCHING FOR $0
LOADING
READY
LIST
```

COLLECTING A DISKETTE

BASIC 4.0 has a COLLECT statement which you can use to "houseclean" a diskette.

The COLLECT statement identifies sectors that have been assigned to data files but are unused. These sectors are made available again, and the diskette directory is modified appropriately.

The COLLECT statement is usually executed in immediate mode, as follows:

```
COLLECT          Collect diskettes on both drives.
COLLECT D0       Collect the diskette in drive 0.
```

Some versions of BASIC 4.0 have a problem with the SCRATCH statement that prevents files from being scratched if they were improperly closed. If your CBM computer has this problem you can overcome it by executing a COLLECT statement before the SCRATCH. The improperly closed file will then be deleted by the SCRATCH statement.

COPYING FILES AND DISKETTES

You should make backup copies of every file that you wish to keep permanently. At least one copy of the file should be on a different diskette. Keeping a copy of the file on the same diskette will not help if the entire diskette is erased by accident.

CBM BASIC statements allows you to copy an individual file or backup an entire diskette.

Copying Files (BASIC 4.0)

The BASIC 4.0 COPY statement lets you copy a single file or an entire diskette. But the COPY statement will only address one physical unit, therefore copies must be made on the same diskette, or using the two drives in a single diskette unit.

If a file name is specified in the COPY statement parameter list, then a single file is copied. If no file name is specified, then all files on the diskette are copied. Here are some immediate mode examples:

```
COPY D0 TO D1
```
Copy all files on the diskette in drive 0 to the diskette in drive 1.

```
COPY D0, "TESTDATA" TO D1, "TESTDATA"
```
Copy file "TESTDATA" from the diskette in drive 0 to the diskette in drive 1. Keep the filename.

```
COPY D1, "TESTDATA" TO D0, "NEWTEST"
```
Copy file "TESTDATA" from the diskette in drive 1 to the diskette in drive 0. Rename the file "NEWTEST."

Copying Files (BASIC<3.0)

In order to copy files using BASIC<3.0, use the PRINT# statement with the following string parameter:

"COPYM:NEWNAME=N:OLDNAME"

Instead of COPY you can have the letter C. N is the drive number holding the (old) source file diskette; OLDNAME represents the name of the source file. M is the drive number holding the (new) destination file diskette; NEWNAME represents the name which will be assigned to the new destination file.

Here are some examples:

```
OPEN 15,8,15
PRINT#15, "COPY1:MAILDATA=0:MAILDATA"
CLOSE 15
```
Copy a file named "MAILDATA" from the diskette in drive 0 to the diskette in drive 1. Keep the filename.

```
OPEN 15,8,15
PRINT#15, "C0:NEWTEST=1:TESTDATA"
CLOSE 15
```
Copy file "TESTDATA" from the diskette in drive 1 to the diskette in drive 0. Rename the file "NEWTEST."

Concatenating Files (BASIC<3.0)

In the course of copying files, the BASIC<3.0 PRINT# statement allows two, three, or four source files to be concatenated into a single destination file. The following immediate mode example concatenates data files DATA1 and DATA2, taken from the diskette in drive 0, and writes them to the diskette in drive 1, assigning the name DATAX to the concatenated data file:

```
OPEN 15,8,15
PRINT#15,"C1:DATAX=1:DATA1,1:DATA2"
CLOSE 15
```

Concatenated source files do not have to come from the same diskette, as shown above. DATAX could be concatenated from data files residing on the same diskette and/or the other diskette.

File Copying Errors

A copy operation cannot specify a destination file name that already exists. If it does, the COPY operation will not occur. The error light of the diskette drive will turn red; when you fetch error status, a FILE EXISTS error will be reported.

When copying all files from one diskette to another using the BASIC 4.0 COPY statement, if a source file name is found to exist on the destination diskette, then the COPY operation stops immediately. The error indicator at the diskette drive turns red. No files get copied if their names appear on the source diskette directory after the duplicated file name.

Duplicating a Diskette

You can copy all files from one diskette to another; you can also backup a diskette by making a duplicate of it. The two are not the same. The backup operation creates a destination diskette which is an exact duplicate of the source, with the same diskette name and number as well as the same files. In contrast, if you copy all files from a source diskette to the destination diskette, the destination diskette name does not change, nor do any files which were previously on the destination diskette. Thus the destination diskette will have a different name, and although it will have all of the source diskette files, it may also have additional files which the source diskette did not have.

Backup a Diskette (BASIC 4.0)

Use the BASIC 4.0 BACKUP statement to duplicate a diskette. You can copy from drive 0 to drive 1 or from drive 1 to drive 0 of any valid physical unit. Here are some examples of the BACKUP statement executed in immediate mode:

```
BACKUP D0 TO D1          Make a copy of the diskette in drive 0 on the diskette in drive 1.
BACKUP D1 TO D0 ON U5    Make a copy of the diskette in drive 1 on the diskette in drive 0.
                           The disk unit is addressed as physical unit 5.
```

The BACKUP statement lets you copy onto a diskette that has not been prepared. If necessary the destination diskette is prepared before the BACKUP operation begins.

Duplicating a Diskette (BASIC<3.0)

Use the PRINT# statement to copy a diskette using BASIC<3.0. You can copy from drive 1 to drive 0, or from drive 0 to drive 1. You cannot copy from a drive in one physical unit to a drive in another physical unit. The PRINT# statement must have the following string variable in its parameter list:

```
"DUPLICATEN=M"
```

Instead of DUPLICATE you can use D. N is the destination drive number; M is the source drive number.

Here is an immediate mode example:

```
OPEN 15,8,15          Make a copy of the diskette in drive 0 on the diskette in drive 1
PRINT#15, "D1=0"
CLOSE 15
```

RENAMING A FILE

You can rename any program or data file. Most frequently program files are renamed in the normal course of writing and correcting programs.

Renaming a File (BASIC 4.0)

Use the BASIC 4.0 RENAME statement to rename a single file. Here is an immediate mode example:

```
RENAME D0, "SEQ.NUM.B4" TO "SEQNUM"
```

Rename a File (BASIC<3.0)

To rename a single file using BASIC<3.0 use the PRINT# statement with the following string variable in its parameter list:

"RENAMEX:NEWNAME=OLDNAME"

Instead of RENAME you can have R. X is the drive number holding the diskette on which the file being renamed is stored. NEWNAME is the new file name; it replaces OLDNAME, the old file name.

Here is an immediate mode example:

```
OPEN 15,8,15                          The file on drive 0 named "SEQ. NUM. B4" is
PRINT#15, "R0:SEQNUM=SEQ.NUM.B4"        renamed "SEQNUM"
CLOSE 15
```

DELETING FILES

You can delete any file from a diskette. When you delete a file in immediate mode the CBM computer will always display the prompt message ARE YOUR SURE? You must respond by typing "YES", and then a carriage return, otherwise the file will not be deleted. If you delete a file in program mode, no prompt message is displayed.

Scratch a File (BASIC 4.0)

Using BASIC 4.0 you delete a file using the SCRATCH statement. Here is an immediate mode example:

```
SCRATCH D0, "REL.NUM.B4"          Delete file REL.NUM. B4 on drive DO
```

Here is the program mode version of the immediate mode example given above:

```
.
.
240 DCLOSE
250 SCRATCH D0, "REL.NUM.B4"
.
.
```

Some versions of BASIC 4.0 have a problem with the SCRATCH statement; it **will not delete files that were not properly closed.** You can solve this problem by collecting the diskette, and then scratching the file.

Scratch a File (BASIC<3.0)

Using BASIC<3.0 you can scratch one or more files using a single PRINT# statement. The PRINT# statement must have the following string variable in its parameter list:

"SCRATCHX:FILENAME"

Instead of SCRATCH you can use S. X is the drive number holding the diskette with the file being scratched. FILENAME is the name of the file being scratched. For a single file this may be illustrated as follows in immediate mode:

```
OPEN 15,8,15                          Delete file REL. NUM. B4 on drive 0
PRINT#15, "S0:REL.NUM.B4"
CLOSE 15
```

To delete two or more files you simply add the drive number and file name to the parameter string. For example, you can modify the statements illustrated above and delete two files as follows:

```
OPEN 15,8,15                                  Delete file REL. NUM. B4 on drive 0 and file
PRINT#15, "S0:REL.NUM.B4,1:REL.NUM.B<3"         REL.NUM.B<3 on drive 1
CLOSE 15
```

If you place an asterisk after one or more letters of a file name, then any file whose name has the letters preceding the asterisk will be deleted. Consider the following example:

```
OPEN 15,8,15
PRINT#15, "S0:NUM*"
CLOSE 15
```

Any file on drive 0 whose name begins with the three letters NUM will be deleted.

If you replace a character in a file name with a question mark, then the name of the file to be scratched can have any character in that position.

For example the following statements delete a file whose name begins with NUM, ends with .SEQ, and has four characters in between.

```
OPEN 15,8,15
PRINT#15, "S0:NUM????.SEQ"
CLOSE 15
```

Replace a File (BASIC<3.0)

Although BASIC<3.0 does not allow you to write into an old file, it does allow the contents of an old file to be replaced. The old file should be opened for a write operation with an @ sign appearing as the first character in the parameter list string variable. For example, the MAIL file opened on line 50 below could be an old file:

```
50 OPEN 1,8,7,"@1:MAIL,SEQ,WRITE"
```

SEQUENTIAL DATA FILES

BASIC 4.0 and BASIC<3.0 both support sequential data files.

A sequential data file is opened either for a read access or for a write access, never for both. When a new sequential file is opened, the process of opening the file also creates it. The new sequential file must be opened for a write operation; it cannot be opened for a read operation. An existing sequential file must be opened for a read operation; it cannot be opened for a write operation.

SEQUENTIAL FILE FIELD SEPARATORS

Numeric variables in a sequential data file must be terminated by carriage return characters. String variables may be terminated by comma characters or by carriage return characters.

We recommend that you **use carriage return characters to separate all fields in sequential data files.** Using comma characters to separate string variables offers no identifiable advantage and can cause unnecessary programming problems.

If all fields are terminated with a carriage return, then rules for writing to sequential data files are very simple: use the PRINT# statement with a parameter list which in a PRINT statement would display variables on the screen as a single vertical column. The data is read back using INPUT# or GET# statements. Using BASIC 4.0 with DOS 2.0, PRINT# statements automatically add a carriage return character at the end of a line if the logical file number is 128 or higher. No terminating carriage return character is output if the file number is 127 or less.

WRITING NUMERIC DATA TO A SEQUENTIAL FILE

Beginning with a very simple example, we will write a program that opens a sequential file, then writes ten records to the file, with ten numbers in each record, as follows:

Record 1:	1	2	3	4	5	6	7	8	9	10
Record 2:	101	102	103	104	105	106	107	108	109	110
Record 3:	201	202	203	204	205	206	207	208	209	210
Record 4:	301	302	303	304	305	306	307	308	309	310
Record 5:	401	402	403	404	405	406	407	408	409	410
etc.										

The program will read the records back and display them. Listings for BASIC 4.0 and BASIC<3.0 versions of this program are given below. The programs are named SEQ.NUM.B4 and SEQ.NUM.B3.

BASIC 4.0 Version

```
10 REM PROGRAM "SEQ.NUM.B4"
20 DOPEN#1,"TESTDATA",W
30 IF DS<>0 THEN PRINT DS$:STOP
40 REM WRITE TEN RECORDS
50 FOR R=1 TO 10
60 REM WRITE TEN FIELDS PER RECORD
70 FOR F=1 TO 10
80 PRINT#1,(R-1)*100+F
85 IF DS<>0 THEN PRINT DS$:STOP
90 NEXT F
100 NEXT R
```