```
110 DCLOSE#1
200 REM NOW READ BACK FILE CONTENTS AND DISPLAY IT
210 DOPEN#1,"TESTDATA"
215 IF DS<>0 THEN PRINT DS$:STOP
220 FOR R=1 TO 10
230 PRINT "RECORD";R;
240 REM INPUT CONTENTS OF NEXT RECORD AND DISPLAY IT
250 FOR F=1 TO 10
260 INPUT#1,N
265 IF DS<>0 THEN PRINT DS$:STOP
270 PRINTN;
280 NEXT F
290 PRINT
300 NEXT R
310 DCLOSE#1
320 SCRATCH D0,"TESTDATA"
330 STOP
```

BASIC < 3.0 Version

```
10 REM PROGRAM "SEQ.NUM.B<3"
20 OPEN 15,8,15:REM COMMAND CHANNEL
21 INPUT#15,A$,B$,C$,D$
22 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$
23 PRINT#15,"I0"
24 OPEN 1,8,2,"0:TESTDATA<3,SEQ,W":REM DATA FILE
30 INPUT#15,A$,B$,C$,D$
31 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$
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 INPUT#15,A$,B$,C$,D$
86 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$
90 NEXT F
100 NEXT R
110 CLOSE 1
120 CLOSE 15
200 REM NOW READ BACK FILE CONTENTS AND DISPLAY IT
210 OPEN 15,8,15:REM COMMAND CHANNEL
211 INPUT#15,A$,B$,C$,D$
212 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$
213 OPEN 1,8,2,"0:TESTDATA<3,SEQ":REM DATA FILE
215 INPUT#15,A$,B$,C$,D$
216 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$
220 FOR R=1 TO 10
230 PRINT "RECORD";R;
240 REM INPUT CONTENTS OF NEXT RECORD AND DISPLAY IT
250 FOR F=1 TO 10
260 INPUT#1,N
265 INPUT#15,A$,B$,C$,D$
266 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$
270 PRINTN;
280 NEXT F
290 PRINT
300 NEXT R
310 CLOSE 1
320 SCRATCH D0,"TESTDATA<3"
330 CLOSE 15
340 STOP
```

Key in the version of the program that will work on your CBM computer, check it carefully for errors, save the program, then run it. You should get the display shown below when you run the program.

```
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
RECORD 6   501   502   503   504   505   506   507   508   509   510
RECORD 7   601   602   603   604   605   606   607   608   609   610
RECORD 8   701   702   703   704   705   706   707   708   709   710
RECORD 9   801   802   803   804   805   806   807   808   809   810
RECORD 10   901   902   903   904   905   906   907   908   909   910
```

Let us examine program logic.

Statements on lines 10 through 120 create the sequential data file and write ten records into it. Statements on lines 200 through 320 read the contents of the sequential data file, record by record, and display data as it is read.

Look at how files have been opened and closed.

In the BASIC 4.0 version sequential data file TESTDATA is opened for a write operation by the DOPEN# statement on line 20. Logical file number 1 is used by the DOPEN# statement. The file is closed on line 110 before being reopened for a read operation by the DOPEN# statement on line 210. Logical file number 1 is used again by the DOPEN# statement on line 210; reusing the same logical file number for the same data file is not necessary. Logical file 1 is closed finally on line 310.

The BASIC<3.0 version of the program opens its sequential data file TESTDATA<3 for a write operation using the OPEN statement on line 24. Logical file number 1 is used with secondary address 2. The file is closed on line 110 before being reopened for a write operation by the OPEN statement on line 213. TESTDATA<3 is finally closed on line 310. The BASIC<3.0 program also opens the command channel via the OPEN statement on line 20 using logical file number 15, which is optional, and secondary address 15, which is necessary. It is common practice to use logical file #15 for the command channel in BASIC<3 programs since the secondary address associates this number with the command channel. The command channel is closed on line 120, it is reopened on line 210, and closed finally on line 330. The command channel does not have to be closed and reopened. Lines 120 and 210 could be eliminated. But closing and reopening the command channel establishes the two halves of the program as separate modules which can be executed independently.

Notice that the BASIC<3.0 program initializes the diskette on line 23. Strictly speaking, the diskette should be re-initialized after the command channel is reopened on line 210 if the two halves of the program are to be treated as separate modules.

The BASIC 4.0 and BASIC<3.0 programs both scratch the data file at the end of the program (on line 320). If the data file were not scratched you would not be able to re-execute the program. Try eliminating statement 320 and running the program twice. On the second execution you will get a FILE ALREADY EXISTS error when the data file is opened for a write operation (on line 20 in the BASIC 4.0 version and on line 24 in the BASIC<3.0).

You should scratch temporary data files at the end of a program if the data held in the file does not need to be saved. If the temporary data file is not scratched it cannot be reused when the program is re-executed.

Next look at the diskette status logic in the two programs; this logic is missing from most programs written by programmers who are in a hurry. (BASIC 4.0 and BASIC<3.0 statements needed to test diskette status were described earlier in this chapter.)

The BASIC 4.0 program tests diskette status on lines 30, 85, 215, and 265. In each case the status string variable DS$ is displayed to identify a problem when status is not 0. Program execution is then stopped.

The BASIC<3.0 program executes the same logic by inputting status via string variables A$, B$, C$, and D$. If the numeric value of A$ is not zero, then the four variables are displayed. The disk status testing statements can be found on lines 21 and 22, 30 and 31, 85 and 86, 210 and 211, 215 and 216, 265 and 266.

The BASIC 4.0 and BASIC<3.0 programs contain identical statements to write records to the sequential data file, to read records back, and to display data.

Records are written to the sequential data file by statements on lines 50 through 100. The outer FOR-NEXT loop, indexed by R, counts records; the inner FOR-NEXT loop, indexed by F, counts fields within records. The PRINT# statement on line 80 writes each field to the sequential data file. Since there is only one variable in the PRINT# statement parameter list, a carriage return is forced; you do not have to force one. Remember, if the PRINT# statement rewritten as PRINT statements would display fields in a single vertical column, then the fields will be written correctly to the diskette data file.

Statements on lines 220 through 300 read data back from the sequential file and display the data. The outer FOR-NEXT loop indexed by R reads records; the PRINT statement on line 230 starts each record display with the record number. The inner FOR-NEXT loop indexed for F read fields one at a time using the INPUT# statement on line 260. Fields for a single record are displayed on one line by the PRINT statement on line 270. The PRINT statement on line 290 forces a carriage return after each record has been displayed.

Although we have described the sequential data file as consisting of ten records with ten fields in each record, on the diskette surface the sequential file consists of 100 fields separated by carriage return characters. If you were to look at the data as stored on the diskette surface, you would find nothing to identify the end of one record or the beginning of the next. Program logic must keep track of records and fields.

To demonstrate the lack of any real file structure on the diskette, **change the second half of the program so that it assumes 12 records, with 8 fields per record.** Statements on lines 220 and 250 must change as follows:

```
220 FOR R=1 TO 12
250 FOR F=1 TO 8
```

Make these changes in your program, then run it. The following display will appear:

```
RECORD 1   1   2   3   4   5   6   7   8
RECORD 2   9   10   101   102   103   104   105   106
RECORD 3   107   108   109   110   201   202   203   204
RECORD 4   205   206   207   208   209   210   301   302
RECORD 5   303   304   305   306   307   308   309   310
RECORD 6   401   402   403   404   405   406   407   408
RECORD 7   409   410   501   502   503   504   505   506
RECORD 8   507   508   509   510   601   602   603   604
RECORD 9   605   606   607   608   609   610   701   702
RECORD 10   703   704   705   706   707   708   709   710
RECORD 11   801   802   803   804   805   806   807   808
RECORD 12   809   810   901   902   903   904   905   906
```

Each record begins reading fields wherever the previous record left off. No attention was paid to the field/record organization used when the file was written.

When a single PRINT# statement writes two or more numeric variables to a data file, you must force carriage return characters using the CHR$ function. Suppose on line 80 we output R, F and the computed expression. The PRINT# statement would have to be rewritten as follows:

```
80 PRINT#1,R,CHR$(13),F,CHR$(13),(R-1)*100+F
```

Usually the carriage return character is assigned to a string variable and the string variable is used in the PRINT# statement as follows:

```
15 C$=CHR$(13)
.
.
80 PRINT#1,R,C$,F,C$,(R-1)*100+F
```

There are now 30 numbers in each record, not 10. Therefore 30 numbers must be read and displayed for each record in the second half of the program. A simple (but inelegant) way of displaying 30 numbers would be to change the FOR statement on line 250, increasing the upper index of F from 10 to 30, as follows:

```
250 FOR F=1 TO 30
```

Make these changes, then run the program to assure yourself that three numbers were written out each time the PRINT# statement on line 80 was executed.

WRITING STRING DATA TO A SEQUENTIAL FILE

String variables can be separated using comma characters or carriage return characters. However, the use of comma character separators serves no useful purpose when string variables are stored in sequential files. Therefore **we will end all sequential file text variables using carriage return characters.**

There is no difference between program logic needed to write string variables or numeric variables to a sequential file.

We will write a simple mailing list program to illustrate string variables being stored in a sequential data file. Listings for BASIC 4.0 and BASIC<3.0 versions of this program are given below, followed by an illustration of program execution.

BASIC 4.0 Version

```
10 REM PROGRAM "SEQ.MAIL.B4"
20 REM MAILING LIST PROGRAM TO ILLUSTRATE DISKETTE FILE STRING HANDLING
30 DATA "   NAME: "," STREET: ","   CITY: ","  STATE: ","    ZIP: "
40 DOPEN#1,"SEQ.MAILDATA",W
50 IF DS<>0 THEN PRINT DS$:STOP
60 PRINT"□ ENTER NAME AND ADDRESS:■■"
70 FOR I=1 TO 5
80 READ F$
90 PRINTF$;:INPUT AD$(I)
100 NEXT I
110 RESTORE
120 PRINT"ENTER Y TO RECORD,N TO RE-ENTER";
130 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 130
135 PRINTY$
140 IF Y$="N" THEN 60
150 REM WRITE NAME AND ADDRESS TO SEQUENTIAL FILE
160 FOR I=1 TO 5
170 PRINT#1,AD$(I)
180 NEXT I
190 PRINT"ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO END";
200 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 200
205 PRINTY$
210 IF Y$="Y" THEN 60
220 DCLOSE#1
```

```
300 REM DISPLAY NAMES AND ADDRESSES ONE AT A TIME
310 DOPEN#1,"SEQ.MAILDATA"
330 IF DS<>0 THEN PRINT DS$:STOP
340 REM CLEAR SCREEN AND DISPLAY NAME AND ADDRESS
350 PRINT"□■■"
360 RESTORE
370 FOR I=1 TO 5
380 READ F$:PRINT F$;
390 INPUT#1,AD$
400 IF DS<>0 THEN PRINT DS$:STOP
410 PRINT AD$
420 NEXT I
430 PRINT"ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO END";
440 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 440
450 IF Y$="Y" THEN 350
460 DCLOSE#1
470 SCRATCH D0,"SEQ.MAILDATA"
480 STOP
```

BASIC<3.0 Version

```
10 REM PROGRAM "SEQ.MAIL.B<3"
20 REM MAILING LIST PROGRAM TO ILLUSTRATE DISKETTE FILE STRING HANDLING
30 DATA "    NAME: "," STREET: ","    CITY: ","  STATE: ","    ZIP: "
40 OPEN 15,8,15:REM COMMAND CHANNEL
41 INPUT#15,A$,B$,C$,D$
42 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$
43 PRINT#15,"I0"
44 OPEN 1,8,2,"0:MAILDATA<3,SEQ,W"
50 INPUT#15,A$,B$,C$,D$
51 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$
60 PRINT"□ ENTER NAME AND ADDRESS:■■"
70 FOR I=1 TO 5
80 READ F$
90 PRINTF$;:INPUT AD$(I)
100 NEXT I
110 RESTORE
120 PRINT"ENTER Y TO RECORD,N TO RE-ENTER";
130 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 130
135 PRINTY$
140 IF Y$="N" THEN 60
150 REM WRITE NAME AND ADDRESS TO SEQUENTIAL FILE
160 FOR I=1 TO 5
170 PRINT#1,AD$(I)
180 NEXT I
190 PRINT"ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO END";
200 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 200
205 PRINTY$
210 IF Y$="Y" THEN 60
220 CLOSE 1
300 REM DISPLAY NAMES AND ADDRESSES ONE AT A TIME
310 OPEN 1,8,2,"0:MAILDATA<3,SEQ"
320 INPUT#15,A$,B$,C$,D$
321 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$
330 IF DS<>0 THEN PRINT DS$:STOP
340 REM CLEAR SCREEN AND DISPLAY NAME AND ADDRESS
350 PRINT"□■■"
360 RESTORE
370 FOR I=1 TO 5
380 READ F$:PRINT F$;
390 INPUT#1,AD$
400 INPUT#15,A$,B$,C$,D$
401 IF VAL(A$)<>0 THEN PRINT A$,B$,C$,D$
410 PRINT AD$
420 NEXT I
430 PRINT"ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO END";
440 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 440
450 IF Y$="Y" THEN 350
460 CLOSE 1
470 SCRATCH D0,"SEQ.MAILDATA"
480 STOP
```

```
ENTER NAME AND ADDRESS:

   NAME: JO BLOW
 STREET: 125 5TH. AVE
   CITY: NEW YORK
  STATE: NY
    ZIP: 10010
ENTER Y TO RECORD,N TO RE-ENTERY
ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO ENDY
 ENTER NAME AND ADDRESS:

   NAME: FRED SMITH
 STREET: 23 ROYAL RD.
   CITY: BERKELEY
  STATE: CA
    ZIP: 94708
ENTER Y TO RECORD,N TO RE-ENTERY
ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO ENDN

   NAME: JO BLOW
 STREET: 125 5TH. AVE
   CITY: NEW YORK
  STATE: NY
    ZIP: 10010
ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO END


   NAME: FRED SMITH
 STREET: 23 ROYAL RD.
   CITY: BERKELEY
  STATE: CA
    ZIP: 94708
```

Let us examine program logic.

Statements on lines 40 through 220 input names and addresses from the keyboard, then output the names and addresses to a sequential data file. Statements on lines 300 through 460 read names and addresses from the sequential data file and display them.

The sequential data file is named SEQ.MAILDATA in **the BASIC 4.0 program**. This sequential file is opened on line 40 for a write operation; it is closed on line 220. The file is reopened on line 310 for a read operation, and finally closed on line 460. In **the BASIC<3.0 version** of the program the sequential data file is named MAILDATA<3. The file is opened on line 44 for a write operation; it is closed on line 220. The file is reopened on line 310 for a read operation, and finally closed on line 460.

Both programs scratch the sequential data file on line 470 so that the program can be rerun. A real mailing list program would not scratch the file; mailing lists need to be preserved. Instead, additional names and addresses would be appended to the file. Appending data to sequential files is described next.

File status is tested in the BASIC 4.0 version of the program by statements on line 50, 330, and 400. In the BASIC<3.0 version file status is tested on lines 41 and 42, 50 and 51, 320 and 321, and 400 and 401. File status statement logic was described earlier in this chapter.

Notice that SEQ.MAIL.B<3 opens a command channel at the beginning of the program on line 40. The STOP statement on line 480 is allowed to close the command channel; this is not good programming practice, but it will work.

Identical statements are used by the BASIC 4.0 and BASIC<3.0 versions of the mailing list program to read data from the keyboard, write data to the sequential file, read data from the sequential file, and display data on the screen.

Statements on lines 60 through 140 input names and addresses from the keyboard. Names and addresses are input as five fields by the FOR-NEXT loop on lines 70 through 100. Notice that the operator's prompt message is identified by string variable F$ which is read from the DATA statement on line 30. The five fields of the name and address are input to string array AD$(I). The RESTORE statement on line 110 restores the data pointer to select the first string variable of the DATA statement.

Statements on lines 120 through 140 are standard operator dialogue which allow the operator to re-enter the entire name and address, or record it. This type of dialogue was described frequently in Chapter 5. Note that very primitive error recovery logic is provided since our goal is to demonstrate file handling; we are not trying to illustrate good data entry programming practice.

The name and address is written to the sequential data file by the FOR-NEXT loop on lines 160 through 180. Since one string variable is output each time the PRINT# statement on line 170 is executed, a carriage return is forced. We could replace statements on lines 160 through 180 with these two statements:

```
160 C$=CHR$(13)
170 PRINT#1,AD$(1),C$,AD$(2),C$,AD$(3),C$,AD$(4),C$,AD$(5)
```

The following INPUT# statement can be used optionally to read the data back:

```
200 INPUT#1,AD$(1),AD$(2),AD$(3),AD$(4),AD$(5)
```

Statements on lines 190 through 210 allow the operator to enter another name and address, or proceed to the display portion of the program.

The FOR-NEXT loop on lines 370 through 420 reads the five fields of each name and address from the sequential data file, then displays the name and address. Once again the DATA statement on line 30 is used to provide labels for each field that is displayed. On line 380 the READ statement takes the next string value from the DATA statement on line 30 and assigns it to F$; the PRINT statement then displays this string variable as a label. The INPUT# statement on line 390 reads the corresponding field from the sequential data file and the PRINT statement on line 410 displays it.

Operator dialogue on lines 430 through 450 allow the operator to display the next name and address, or terminate program execution.

Note that we have provided no protection against the operator asking for another name and address to be displayed when the end of file has been reached. We could solve this problem by adding the following statements on a new line 405:

```
405 IF DS=64 THEN PRINT "END OF FILE": I=5: GOTO 420
```

Mixed Sequential Data Files

No special program logic is needed in order to write numeric and string variables to the same sequential data file. However your program logic must keep track of field types. If a statement attempts to read a field from a sequential data file using a variable name of the wrong type, then an error will be reported.

Here is an example of a statement that writes two numeric variables and three string variables to a sequential data file:

```
10 DOPEN#1,"DATA",W
20 C$=CHR$(13)
30 PRINT#1,P$,C$,X,C$,Q$,C$,Y,C$,R$
```

These five variables would be read back correctly by the following INPUT# statement:

```
100 INPUT#1,A$(1),A$(2),A$(3),X(1),X(2),
```

The following INPUT# statement would not execute correctly since the variable types in its parameter list do not correspond with the variable types recorded in the sequential data file:

```
100 INPUT#1,A$(1),A$(2),A$(3),X(1),X(2)
```

There are now 30 numbers in each record, not 10. Therefore 30 numbers must be read and displayed for each record in the second half of the program. A simple (but inelegant) way of displaying 30 numbers would be to change the FOR statement on line 250, increasing the upper index of F from 10 to 30, as follows:

```
250 FOR F=1 TO 30
```

Make these changes, then run the program to assure yourself that three numbers were written out each time the PRINT# statement on line 80 was executed.

ADDING DATA TO SEQUENTIAL FILES

BASIC 4.0 allows you to add data to an existing sequential file using the APPEND# and CONCAT statements. The APPEND statement will write fields to the end of the existing file; the CONCAT statement will concatenate two files.

Appending Data To Sequential Files (BASIC 4.0)

To illustrate the APPEND# statement, we will modify program "SEQ.NUM.B4". The modified program, named "SEQ.NUMAPPEND", is listed below, with changed statements shaded.

```
10 REM PROGRAM "SEQ.NUMAPPEND"
20 DOPEN#1,"TESTDATA",W
30 IF DS<>0 THEN PRINT DS$:STOP
35 FOR J=1 TO 3
40 REM WRITE TEN RECORDS
50 FOR R=1 TO 10
60 REM WRITE TEN FIELDS PER RECORD
70 FOR F=1 TO 10
80 PRINT#1,(R-1)*100+F*J
85 IF DS<>0 THEN PRINT DS$:STOP
90 NEXT F
100 NEXT R
110 DCLOSE#1
200 REM NOW READ BACK FILE CONTENTS AND DISPLAY IT
210 DOPEN#1,"TESTDATA"
215 IF DS<>0 THEN PRINT DS$:STOP
220 FOR R=1 TO 10*J
230 PRINT "RECORD";R;
240 REM INPUT CONTENTS OF NEXT RECORD AND DISPLAY IT
250 FOR F=1 TO 10
260 INPUT#1,N
265 IF DS<>0 THEN PRINT DS$:STOP
270 PRINTN;
280 NEXT F
290 PRINT
300 NEXT R
310 DCLOSE#1
315 APPEND#1,"TESTDATA"
316 NEXT J
320 SCRATCH D0,"TESTDATA"
330 STOP
```

"SEQ.NUMAPPEND" is equivalent to three executions of "SEQ.NUM.B4". On the first execution 100 numeric fields are written to "TESTDATA". On each re-execution 100 fields are added to sequential data file TESTDATA. Therefore after the second execution "TESTDATA" will hold 200 numbers, and after the third execution "TESTDATA" will hold 300 numbers.

The three executions are enabled by a FOR-NEXT loop which uses the index J. The FOR statement is on line 35. The NEXT statement is on line 316.

In order to identify appended numbers, the field counter F is multiplied by the execution counter J on line 80. On line 220 the upper bound for the record counter R becomes 10 * J, since the number of records will increase by 10 on each re-execution.

You cannot APPEND to a file that does not exist. Therefore you cannot simply replace the DOPEN# statement on line 20 with an APPEND# statement, and open "TESTDATA" within the FOR-NEXT loop indexed by J. The DOPEN# statement on line 20 creates sequential file "TESTDATA" and opens it for a write operation. Ten records are written to "TESTDATA" on the first execution of statements 40 through 315; these ten records are read from the file and displayed. At the end of the first execution the APPEND# statement on line 315 reopens TESTDATA for the second execution of statements on lines 40 through 315. Ten additional records are added to TESTDATA. Similarly on the third execution of statements on lines 40 through 315, ten more records are added to TESTDATA, bring the total to 30 records.

Now run the program. On the first execution you will see exactly the same display that program SEQ.NUM.B4 created. There will be a pause, then on the second execution 20 records will be displayed; you will be able to identify the second set of ten records by the fact that the last digit of each number has been doubled. After another short pause you will see 30 records displayed when the program is executed a third time. You will be able to differentiate the first, second, and third set of ten records by the last digit of each number, which is doubled for the second set of ten records, and tripled for the third set of ten records.

Concatenating Sequential Data Files (BASIC 4.0)

BASIC 4.0 with DOS 2.0 allows you to concatenate files using the CONCAT statement. Program CONCATEST, listed below, provides a simple demonstration of file concatenation.

```
5 REM PROGRAM "CONCATEST", DEMONSTRATES CONCAT STATEMENT
10 DOPEN#1,"DATA1",W
20 DOPEN#2,"DATA2",W
30 FOR I=1 TO 20
40 PRINT#1,I
50 PRINT#2,I+10
60 NEXT I
80 DCLOSE
90 DOPEN#1,"DATA1"
100 DOPEN#2,"DATA2"
110 PRINT"□"
120 FOR I=1 TO 20
130 INPUT#1,X:PRINT X;
140 NEXT
145 PRINT
150 FOR I=1 TO 20
160 INPUT#2,X:PRINT X;
170 NEXT
175 PRINT
180 DCLOSE
190 CONCAT "DATA2" TO "DATA1"
```

```
200 DOPEN#1,"DATA1"
210 FOR I=1 TO 40
220 INPUT#1,X:PRINT X;
230 NEXT
235 PRINT
240 DCLOSE
250 STOP
```

This very simple program writes 20 numbers into sequential data files DATA1 and DATA2, then concatenates DATA2 to DATA1. Contents of DATA1 and DATA2 are displayed separately, then the contents of DATA1 are displayed after concatenation.

The two sequential data files DATA1 and DATA2 are opened on lines 10 and 20. The FOR-NEXT loop on lines 30 through 60 writes 20 numeric fields to each of the two files. Numbers one through 20 are written to DATA1. Numbers 11 through 31 are written to DATA2 so that the two numeric sequences can be distinguished, one from the other.

The two data files are closed by the single DCLOSE statement on line 80 so that they can be reopened for read accesses by the DOPEN# statements on lines 90 and 100. Two FOR-NEXT loops on lines 120 through 140 and 150 through 170 display the contents of DATA1 and DATA2 respectively. The PRINT statements on lines 145 and 175 force carriage returns.

DATA1 and DATA2 are both closed on line 180. DATA2 is concatenated to DATA1 by the CONCAT statement on line 190. DATA1 is then opened so that its contents can be displayed by the FOR-NEXT loop on lines 210 through 230. DATA1 is closed on line 240.

Note that CONCATEST does not scratch DATA1 and DATA2 at the end of the program. Before re-executing the program you must SCRATCH files DATA1 and DATA2 in immediate mode, or you must SCRATCH statements to the end of the program as follows:

```
245 SCRATCH "DATA1":SCRATCH "DATA2"
```

It is easy to misuse the CONCAT statement and get into a lot of trouble.

The two concatenated files must both contain data, and must both be closed when the CONCAT statement is executed.

If you concatenate data to an empty file, the computer will "hang up." You must turn power off at the computer, then turn power on again and restart whatever you were doing.

If you attempt to concatenate files that are open, or improperly closed, the computer may start appending a file to the diskette directory. If this happens, you will see diskette activity continue for a very long time after the CONCAT statement has been executed. It is possible to stop the diskette operation by pressing the STOP key at the keyboard. If you display the directory you will see a lot of garbage appear after the valid file names. In order to remove this garbage execute the COLLECT statement in immediate mode.

Appending Data to Sequential Files (BASIC<3.0)

In order to append data to an existing sequential file using BASIC<3.0, you need two sequential files, which we will arbitrarily name DATA1 and DATA2. Suppose DATA1 contains data. In order to add data to DATA1 you must create a new file DATA2 using these steps:

1. If DATA2 exists scratch it.
2. OPEN DATA2 for a write access.
3. OPEN DATA1 for a read access.
4. Read records sequentially from DATA1 and write them sequentially to DATA2.
5. On detecting the end of the DATA1 file, start writing new records to DATA2.
6. Close DATA1.
7. Scratch DATA1.
8. Rename DATA2, giving it the new name DATA1.

The next time you wish to update the file, repeat the steps described above, switching DATA1 with DATA2.

END OF FILE

You can test for an end of file by looking for a value of 64 in ST. The following statement will stop program execution on detecting an end-of-file:

```
200 IF ST=64 THEN PRINT "END OF FILE":STOP
```

RELATIVE DATA FILES (BASIC 4.0)

Only BASIC 4.0 supports relative data files.

An open relative file can be read from or written to. However, you cannot read from an empty relative file; until you have written into the file, you cannot read from it.

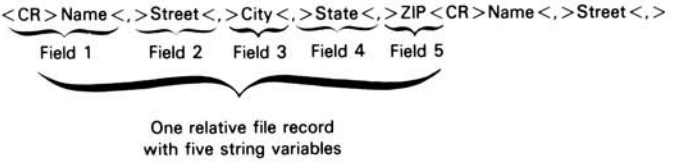
RELATIVE FILE FIELD SEPARATORS

Comma and carriage return characters have different meanings as field separators in relative files; **the record length specified in a relative file DOPEN# statement identifies the number of characters (bytes) separating carriage return characters.** If all fields are separated using carriage return characters, then the relative file record length becomes a field length. Remember, BASIC 4.0 PRINT# statements do not transmit an automatic carriage return character at the end of a line if the file number is 127 or less.

Relative File Record Length

All numeric fields must be terminated with carriage return characters, therefore if a relative file holds numeric data, the record length specified for the relative file is also the field length. The number appearing after the L parameter in the relative file DOPEN# statement identifies the number of characters (bytes) that will be set aside for every numeric field in the file.

Since string variables can be terminated by comma characters or carriage return characters, you can place a number of string variables within a single relative file record. A name and address, for example, could have the following five fields:



The record length specified for the relative file in its DOPEN# statement now applies to all five fields of the name and address record. This is useful since it accommodates records that have one or two very long fields. This may be illustrated as follows:

	Number of Characters					
	Name Field 1	Street Field 2	City Field 3	State Field 4	ZIP Field 5	Total
Address 1	9	14	16	2	5	46
Address 2	13	12	8	2	5	40
Address 3	12	11	12	2	5	42
Address 4	17	8	11	2	5	43
Address 5	10	12	13	2	5	42
etc.						

If all five fields are stored in a single record, a record length of 50 characters (bytes) would probably be adequate.

If every string variable field ended with a carriage return, then the record length specified in the DOPEN# statement would apply to each field of the name and address. Every field would have to be long enough to accomodate the longest expected entry in any one of the five fields. To be safe we would probably select a 20-character (byte) field length. Now every field, including state and ZIP, will be allocated 20 characters. The total allocation for the name and address becomes 100 characters (bytes), since there are five fields with 20 characters per field. Therefore each name and address requires twice as much disk space as it would need if data were stored five fields per record.

Reading Relative File Records

INPUT# and GET# statements can be used to read fields from a relative file.

If commas are used to separate string variables, and INPUT# statements are used to read data from the relative file, then each INPUT# statement must read all of the variables occurring between two carriage return characters. We will illustrate this with programming examples on the following pages.

If a relative file has numeric and string variables, selecting a record length becomes more complicated. You can select a record length that allows a number of string variables separated by commas to be stored in each record, but numeric fields will still have to be stored one per long record. And that can prove very costly in terms of wasted diskette space. There are two solutions to this problem:

1. Select a record length based on the numeric variables. Store string variables one field per record, breaking up any long strings into smaller pieces.
2. Convert numeric variables into strings using the STR$ function, then store a number of numeric strings in each record.

WRITING NUMERIC DATA TO RELATIVE FILES

To explore numeric relative files we will modify program SEQ.NUM.B4, creating REL.NUM.B4, which is listed below.

```
10 REM PROGRAM "REL.NUM.B4"
20 DOPEN#1,"RELDATA",L10
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
110 DCLOSE#1
200 REM NOW READ BACK FILE CONTENTS AND DISPLAY IT
210 DOPEN#1,"RELDATA",L10
215 IF DS<>0 THEN PRINT DS$:STOP
220 FOR R=1 TO 10
230 PRINT "RECORD";R;
240 REM INPUT CONTENTS OF NEXT RECORD AND DISPLAY IT
250 FOR F=1 TO 10
260 INPUT#1,N
265 IF DS<>0 THEN PRINT DS$:STOP
270 PRINTN;
280 NEXT F
290 PRINT
300 NEXT R
310 DCLOSE#1
320 SCRATCH D0,"RELDATA"
330 STOP
```

Load program SEQ.NUM.B4 into memory, then create program REL.NUM.B4 by making appropriate changes to statements on lines 10, 20, and 210. Run program REL.NUM.B4. If it executes correctly you will get the same display that program SEQ.NUM.B4 generated. Save program REL.NUM.B4 when it has executed correctly.

Record Length

Note the short record length of ten characters (bytes) specified by the REL.NUM.B4 program's DOPEN# statements. Since numeric data is written to relative file RELDATA, one field is written per record. This is because record length is always interpreted as the number of characters (bytes) separating carriage return characters; and every numeric variable must be terminated with a carriage return. Therefore just one numeric variable can be stored per record. Ten characters (bytes) is enough space for one numeric field.

There is no need to close relative file RELDATA on line 110 and then reopen it on line 210. We do so in order to separate the program into two modules, and examine how the two halves of the program interact.

Next change the record length in the DOPEN# statement on line 210 from L10 to L8. Now re-execute the program. The program will not execute; the following message will appear:

```
50,RECORD NOT PRESENT, 00,00
BREAK IN 215
READY
```

The wrong record length in the DOPEN# statement on line 210 has caused the problem. **BASIC 4.0 does not allow a relative file to be reopened with the wrong record length.**

WRITING STRING DATA TO RELATIVE FILES

When writing string variables to relative files you can end each variable with a comma or a carriage return character. If you end each field with a carriage return character, there will be one string variable field per record. You can include a number of string variables within a single record by using a comma character to separate fields within the record. The last field of the record must end with a carriage return character.

For our first example of writing string variables to a relative file, we will modify the sequential mailing list program SEQ.MAIL.B4. The modified program generates a relative file with the five fields of each name and address stored as a single record. This new program (named REL.MAIL.B4) is listed below; statements that differ from SEQ.MAIL.B4 are shaded.

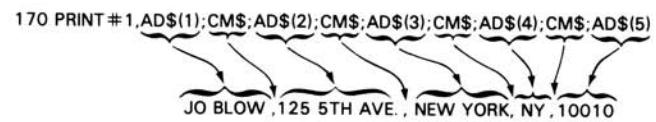
```
10 REM PROGRAM "REL.MAIL.B4"
20 REM MAILING LIST PROGRAM TO ILLUSTRATE DISKETTE FILE STRING HANDLING
25 REM FOR RELATIVE FILES
30 DATA "   NAME: "," STREET: ","   CITY: ","  STATE: ","    ZIP: "
40 DOPEN#1,"REL.MAILDATA",L50
50 IF DS<>0 THEN PRINT DS$:STOP
60 PRINT"□ ENTER NAME AND ADDRESS:▮▮"
70 FOR I=1 TO 5
80 READ F$
90 PRINTF$;:INPUT AD$(I)
100 NEXT I
110 RESTORE
120 PRINT"ENTER Y TO RECORD,N TO RE-ENTER";
130 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 130
135 PRINTY$
140 IF Y$="N" THEN 60
150 REM WRITE NAME AND ADDRESS TO SEQUENTIAL FILE
160 CM$=CHR$(44)
170 PRINT#1,AD$(1);CM$;AD$(2);CM$;AD$(3);CM$;AD$(4);CM$;AD$(5)
171 IF DS<>0 THEN PRINT DS$:STOP
190 PRINT"ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO END";
200 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 200
205 PRINTY$
210 IF Y$="Y" THEN 60
220 DCLOSE#1
224 IF DS<>0 THEN PRINT DS$:STOP
300 REM DISPLAY NAMES AND ADDRESSES ONE AT A TIME
310 DOPEN#1,"REL.MAILDATA",L50
330 IF DS<>0 THEN PRINT DS$:STOP
340 REM CLEAR SCREEN AND DISPLAY NAME AND ADDRESS
350 PRINT"□▮▮"
360 RESTORE
365 INPUT#1,AD$(1),AD$(2),AD$(3),AD$(4),AD$(5)
366 IF DS<>0 THEN PRINT DS$:STOP
370 FOR I=1 TO 5
380 READ F$:PRINT F$;
410 PRINT AD$(I)
420 NEXT I
430 PRINT"ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO END";
440 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 440
450 IF Y$="Y" THEN 350
460 DCLOSE#1
470 SCRATCH D0,"REL.MAILDATA"
480 STOP
```

Load program SEQ.MAIL.B4 from diskette, change statements on the shaded lines, then run the program. If you have entered the program correctly, it will execute exactly as described for SEQ.MAIL.B4. When program REL.MAIL.B4 is free of errors, save it.

Let us examine the changed statements in program REL.MAIL.B4.

The DOPEN# statements on lines 40 and 310 have been changed to specify a relative file with a 50-character record length and the name REL.MAIL.DATA.

Data is input from the keyboard and displayed on the screen as described for SEQ.MAIL.B4, but statements that write each name and address to the data file are completely different. The PRINT# statement on line 170 outputs a single record. CM$ has been assigned the numeric value of the comma character (CHR$(44)) by the assignment statement on line 160. Note the semicolons separating each variable in the PRINT# statement parameter list. The combination of semicolons separating parameters in the PRINT# statement and CM$ occurring between each field of the name and address will cause a relative file record to be created as follows:



This illustration assumes that AD$(1)="JO BLOW", AD$(2)="125 5TH AVE.", AD$(3)="NEW YORK", AD$(4)="NY" and AD$(5)="10010".

Note the statements on line 171, which test for diskette status after each record is written to the relative file. Strictly speaking, program SEQ.MAIL.B4 should have had statements to test status at this point; for SEQ.MAIL.B4 it would have represented good programming practice. But **it is vitally important after writing a record to a relative file, since you must check for record overflow.** Without the status testing statements on line 171, any name and address that did not fit into the allowed record length would be stored inaccurately; if your eye were fast you might notice the error indicator on the diskette drive flash red while the long record is written to the relative file. Otherwise you would have no idea that an overflow had occurred until a program read data back from the file, and found one or more fields of the record missing.

To demonstrate the need for the status testing logic on line 171, eliminate this line, then change the semicolons on line 170 to commas. Now re-execute the program. If you watch carefully you will see the error indicator at the diskette drive flash red when records are written to the diskette. When names and addresses are subsequently displayed, the first two or three fields of each name and address will be present; remaining fields will be absent.

What happened?

The commas in the parameter list of the PRINT# statement on line 170 have the same effect on display fields and relative file fields; each new field is written or displayed beginning at the next 10th character boundary. The PRINT# statement on line 170 has 9 variables in its parameter list (you must count the four CM$ variables). Therefore the record will require at least 90 characters. More characters will be needed if any of the five name and address fields has more than ten characters. You can see the effect of commas by adding the following statement to program REL.MAIL.B4:

```
PRINT AD$(1),CM$,AD$(2),CM$,AD$(3),CM$,AD$(4),CM$,AD$(5)
```

Each record will be displayed exactly as it will be written to file REL.MAIL.DATA. You can then count characters for yourself, and see where the name and address gets truncated by a 50 character record length.

Statements that read the name and address back from the relative data file are shown on lines 365 and 366. Statements that read names and addresses back for program SEQ.MAIL.B4 have been removed; hence the absence of lines 390 and 400.

An INPUT# statement reads one record from a diskette file. This is true for all INPUT# statements, reading from any type of diskette file. In other words, each INPUT# statement reads data from one carriage return character to the next. In program REL.MAIL.B4 there are five fields between each pair of carriage return characters, therefore the INPUT# statement on line 365 will read five fields each time the statement is executed. This is true whatever number of variables there may be in the INPUT# statement parameter list.

The INPUT# statement on line 365 has five string variables in its parameter list. If any variable in the parameter list were not a string variable, you would get a syntax error and the program would stop executing.

If there were less than five string variables in the parameter list, some variables at the end of the relative file records would not be read. You can demonstrate this for yourself by removing AD$(4) and AD$(5) from the INPUT# statement on line 365. When you re-execute the program, names and addresses read from the relative file will have their first three fields displayed correctly, with nothing in the last two fields.

Next add an additional variable to the INPUT# statement on line 365 by appending ,AD$(6) to the end of the INPUT# statement. When you execute the program, you will find that the presence of this additional variable in the INPUT# statement has no effect. Unlike sequential files, the additional variable has no data assigned to it, since the record has run out of fields.

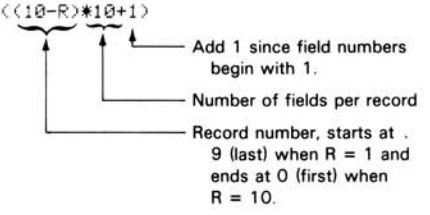
POSITIONING TO RECORDS OF RELATIVE FILE

The RECORD# statement allows you to position to any character (byte) of any record in a relative file. To demonstrate the use of the RECORD# statement, add the following line to program REL.NUM.B4:

```
23 RECORD#1,((10-R)*10+1)
```

You will see ten records displayed, with 901 through 910 in the first record and 1 through 10 in the last record. This is the exact inverse of the record display given by REL.NUM.B4.

The record positioning factor is derived as follows:



Whether a relative record contains numeric or string data has no effect on the way the RECORD# statement works. Prove this to yourself by adding RECORD# statements to the REL.MAIL.B4 program to select names and addresses in any sequence.

Changing Records in a Relative File

Having positioned to any record in a relative file, you can write a single record. No special programming techniques are required. The same PRINT# statement that creates a record can be used to overwrite a record, once you have positioned to the record.

USING GET# WITH DISKETTE FILES

The GET# statement reads one character from a diskette data file, just as the GET statement reads one character from the keyboard buffer. **The character read by the GET# statement is taken from the 256 byte diskette buffer. Characters are taken sequentially, beginning with the first character in the buffer. Blanks, punctuation characters and anything occupying a character position will be read.**

When using the GET# statement to read from sequential files, you must read characters sequentially, beginning with the first character of the file. However, when reading from relative files you can use the RECORD# statement to select any character in any record; the GET# statement will then start reading at the selected character.

Avoid using GET# to read numeric data from disk files. Remember, GET# returns 0 for a null numeric character. Therefore you cannot distinguish between a true 0 digit and a null character.

We will demonstrate use of the GET# statement by modifying programs SEQ.MAIL.B4 and REL.MAIL.B4, substituting a GET# statement for the INPUT# statement that reads back name and address fields. Changes apply also to SEQ.MAIL.B<3.

Using GET# with Sequential Files

First we will modify program SEQ.MAIL.B4 substituting GET# for the INPUT# statement on line 390. The GET# statement follows standard GET statement logic (which you should understand by now). Here is the new line 390:

```
390 GET#1,AD$:IF AD$="" THEN 390
```

The PRINT statement on line 410 will now print just one character; we must therefore add a semicolon to the end of the PRINT statement in order to suppress a carriage return.

We must test for a carriage return by adding this extra statement on a new line 415:

```
415 IF AD$<>CHR$(13) THEN 390
```

The IF statement on line 415 branches back to the GET# statement until a carriage return is detected. Then the FOR-NEXT loop is allowed to iterate once more. Since carriage return characters mark the end of each word, the carriage return is displayed by the PRINT statement on line 410 before the IF statement on line 415 causes program logic to move on to the next word.

Load program SEQ.MAIL.B4 into memory. Make the changes described and run the program. Execution should be identical.

In order to experiment with the GET# statement, try modifying your program to detect and change specific characters. For example, you could display a graphics character wherever a carriage return is detected.

Using GET# with Relative Files

Program REL.MAIL.GET#, listed below, shows program REL.MAIL.B4 modified to use the GET# statement; in addition, some characters have been modified so that we can examine the organization of relative file records.

```
10 REM PROGRAM "REL.MAIL.GET#"
20 REM MAILING LIST PROGRAM TO ILLUSTRATE DISKETTE FILE STRING HANDLING
25 REM FOR RELATIVE FILES
30 DATA "   NAME: "," STREET: "," CITY: "," STATE: "," ZIP: "
40 DOPEN#1,"REL.MAILDATA",L50
50 IF DS<>0 THEN PRINT DS$:STOP
60 PRINT"□ ENTER NAME AND ADDRESS:▮▮"
70 FOR I=1 TO 5
80 READ F$
90 PRINTF$;:INPUT AD$(I)
100 NEXT I
110 RESTORE
120 PRINT"ENTER Y TO RECORD,N TO RE-ENTER";
130 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 130
135 PRINTY$
140 IF Y$="N" THEN 60
150 REM WRITE NAME AND ADDRESS TO SEQUENTIAL FILE
160 CM$=CHR$(44)
170 PRINT#1,AD$(1);CM$;AD$(2);CM$;AD$(3);CM$;AD$(4);CM$;AD$(5)
171 IF DS<>0 THEN PRINT DS$:STOP
190 PRINT"ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO END";
200 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 200
205 PRINTY$
210 IF Y$="Y" THEN 60
220 DCLOSE#1
224 IF DS<>0 THEN PRINT DS$:STOP
300 REM DISPLAY NAMES AND ADDRESSES ONE AT A TIME
310 DOPEN#1,"REL.MAILDATA",L50
330 IF DS<>0 THEN PRINT DS$:STOP
340 REM CLEAR SCREEN AND DISPLAY NAME AND ADDRESS
350 PRINT"□▮▮"
360 RESTORE
370 FOR I=1 TO 5
380 READ F$:PRINT F$;
390 GET#1,AD$:IF AD$="" THEN 390
395 IF DS<>0 THEN PRINT DS$:STOP
400 IF AD$=CHR$(32) THEN AD$="•"
405 IF AD$=CHR$(44) THEN PRINT AD$;:AD$=CHR$(13)
410 PRINT AD$;
415 IF AD$<>CHR$(13) THEN 390
420 NEXT I
430 PRINT"ENTER Y FOR ANOTHER NAME AND ADDRESS,N TO END";
440 GET Y$:IF Y$<>"Y" AND Y$<>"N" THEN 440
450 IF Y$="Y" THEN 350
460 DCLOSE#1
470 SCRATCH D0,"REL.MAILDATA"
480 STOP
```

Since the GET# statement reads characters one at a time, we do not need to worry about the different punctuation separating fields and records. The GET# statement will read punctuation like any other character, and carry on reading. Therefore the INPUT# and status test instructions on lines 365 and 366 of program REL.MAIL.B4 have been removed. A standard GET# statement has been added on line 390; status for this file access is tested by the IF statement on line 395.

In order to detect space codes, on line 400 space code characters are replaced by the more visible • character.

Line 405 checks for a comma. Commas are displayed, then replaced with a carriage return character.

On line 410 a semicolon has been added to the end of the PRINT statement since this statement will now display just one character. On line 415 logic branches back for the next character, unless the carriage return has been detected, at which point the next field is input. Remember, on line 405 commas have been converted to carriage returns, therefore on line 415 commas and carriage returns will both cause an advance to the next field.

Enter program REL.MAIL.B4 and make the modifications shown. Now run the program. You will find that REL.MAIL.GET# and REL.MAIL.B4 create identical displays, apart from asterisks appearing instead of blanks. Notice that no asterisks appear after the zip code. Therefore **a carriage return character must appear directly after the zip code, with unused disk space separating this record from the beginning of the next.**

Using the GET# and RECORD# Statements With Relative Files

The RECORD# statement will position to any character in any record of a relative file. To demonstrate the character positioning ability of the RECORD# statement add the following line to program REL.MAIL.GET#:

```
365 RECORD#1,2,5
```

The second half of program REL.MAIL.GET# will now start displaying names and addresses at the fifth character of the second record. Re-execute the program (making sure that you enter at least two names and addresses). You will find that the second name and address is displayed, beginning at its fifth character.

PROGRAM FILES

CBM computers handle program and data files in totally different ways. Each has its own set of file handling statements.

Loading and Saving Program Files

Program files are loaded into memory using the LOAD (for BASIC<3.0) or the DLOAD (for BASIC 4.0) statements.

Program files are written to diskette using the SAVE (for BASIC<3.0) or the DSAVE (for BASIC 4.0) statements.

Loading and saving programs is described first in Chapter 2.

Accessing Program Files as Data Files

You can OPEN and CLOSE program files as you would data files, and you can execute GET#, INPUT#, and PRINT# statements accessing program files as though they were data files. But until you have an intimate understanding of CBM computer system software, you will get results that are highly unpredictable;

moreover you will achieve nothing that could not be done more easily using standard program file statements and screen editing capabilities.

When using BASIC<3.0, remember that secondary address 0 is used to LOAD program files into memory, while secondary address 1 is used to SAVE program files on diskette. By specifying these secondary addresses in OPEN statements you get to access program files as though they were data.

Backup Program Files

It is imperative that you always have one or more copies of every program file. Wherever possible, at least one copy of the program file should be held on a different diskette. Having two copies of the same program file on one diskette serves no purpose if by some mischance the entire diskette is erased.

Use the BASIC 4.0 COPY statement to copy a single file. Use the COPY or the BACKUP statement to copy an entire diskette.

With BASIC<3.0 you must copy files and diskettes using a variation of the PRINT# statement, as described earlier in this chapter.

Program File Update Sequence

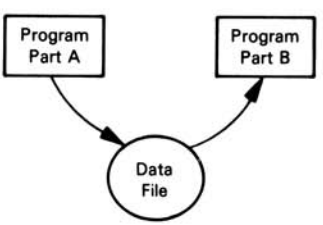
Programs constantly change as you make corrections or improvements. The safest way of changing a program is to keep a copy of the present version, and the two most recent versions, generally referred to as the "father" and "grandfather." When you change a program follow these steps:

1. LOAD the present "current" version into memory and make appropriate changes.
2. SCRATCH the current grandfather program.
3. RENAME the father program as the grandfather.
4. RENAME the current program as the father.
5. SAVE the new version as the new current program.

JOB QUEUING

Programmed use of the LOAD command allows you to execute very long programs and to perform various types of job queuing.

Suppose you have an application whose program will not fit in available memory. Try resolving the problem by splitting the program into two pieces. The two pieces must be completely independent, except for data which one piece can transmit to the other via an external data file. This may be illustrated as follows:



For this scheme to work, the original program must be divisible into two or more independent steps.

Let us call the two parts of the program Part A and Part B. **The entire program is loaded,** using the following steps:

1. Load Part A into memory via an immediate mode LOAD command.
2. Execute Part A via an immediate mode RUN command.
3. When Part A finishes, it loads part B.
4. Part B executes automatically.

Here is a BASIC 4.0 statement which will transfer from Part A to Part B:

```
60030 DLOAD D0,"PART B"
```

Part A must terminate execution by writing out a data file that contains all of the data needed by Part B. Part B must begin execution by loading the data file which Part A wrote out.

PROGRAMMING THE LINE PRINTER

Up to this point we have made very little use of the CBM computer system's line printer. All we have done is list programs; and that takes no programming effort. But most programs generate results in the form of printed reports. The format of a report is very important; reports get used if they are easy to read. A badly formatted report is discarded. Fortunately it is easy to program CBM line printers to generate well formatted reports.

Two printers are available with CBM computer systems: the Model 2022 and the Model 2023. Both printers contain their own internal microprocessors, which is why well formatted printouts are so easy to generate.

The Model 2022 and 2023 printers both print the PET keyboard character set, not the CBM keyboard character set.

Printers are accessed by opening a logical file specifying physical unit #4, and a secondary address whose value must range between 0 and 7. If no secondary address is specified, then 0 is assumed. As summarized in Table 6-4, **secondary addresses provide these printer options**:

1. Print data exactly as received.
2. Print output to a previously specified format.
3. Define the number of lines to be printed per page.
4. Specify the space separating printed lines (Model 2022 Printer only).
5. Print characters that are not part of the standard character set.
6. Enable special diagnostic messages to be printed.

Additional formatting can be specified using the special control characters summarized in Table 6-5.

PRINTING DATA EXACTLY AS RECEIVED

To print data exactly as received you must open a logical file specifying physical unit #4 and no secondary address, or a secondary address of 0. Then print data using PRINT# and/or CMD statements.

Printing with the PRINT# Statement

The PRINT# statement outputs data to the printer just as it would to a cassette or diskette file. For example, to print the word "MESSAGE", enter the following program and run it:

```
10 OPEN 2,4
20 PRINT#2,"MESSAGE"
30 CLOSE 2
40 STOP
```

Each time you run this program the single word "MESSAGE" is printed; then the following display appears:

```
BREAK IN 40
READY
```

This display is generated by the STOP statement on line 40.

You cannot use BASIC 4.0 DOPEN# and DCLOSE# statements to access the line printer. These statements will only work with diskette files.

Printing with the CMD Statement

Instead of using the PRINT# statement you can transmit data to the printer using the CMD statement. But **the CMD statement must be followed by at least one PRINT# statement before the printer logical file is closed.** To demonstrate the use of the CMD statement, enter and run the following program:

```
10 OPEN 2,4
20 CMD 2,"MESSAGE"
25 PRINT#2
30 CLOSE 2
40 STOP
```

Table 6-4. Printer Control Chaacters, used in Text Printed to Secondary Address 0 or 1

	Printer Function	C Code	Keyboard Entry	Comment
Necessary	End string field	CHR$ (29)	CRSR →	Every string field must be followed by CHR$ (29) except for the last variable in a PRINT # statement parameter list.
	Leading blanks in string	CHR$ (160)	SHIFT + SPACE	To insert leading space codes in a string variable you must use shifted space codes. Unshifted space characters will be deleted.
Optional	Enhance	CHR$ (1)	NONE	Each occurance of CHR$ (1) in the parameter list of a PRINT # statement will double the width of subsequent string characters (initially 6 dots) printed on the same line. A carriage return nullifies character enhancement. CHR$ (1) characters have a cumulative effect on characters printed on a single line.
	Line Feed	CHR$ (10)	NONE	The printer executes a carriage return and line feed on encountering this character.
	Carriage Return	CHR$ (13)	RETURN	
	Reverse on	CHR$ (18)	OFF RVS	Start printing reverse field characters. Reverse field is cancelled by a carriage return.
	Lower-case	CHR$ (17)	CRSR ↓	Start printing lower-case letters. Lower-case is cancelled by a carriage return or an upper-case specification.
	Paging off	CHR$ (19)	HOME	Execute a top of form if paging is on.
	Quote character	CHR$ (34)	NONE	Print all control characters.
	Unenhance	CHR$ (129)	NONE	Cancel character enhancement specified by preceding CHR$ (1) character(s)
	Carriage return with no line feed	CHR$ (141)	NONE	The printer executes a carriage return without a line feed, overprinting prior text, if any.
	Upper-case	CHR$ (145)	CRSR ↑	The printer returns to printing upper-case letters if it is printing lower-case letters following a CHR$ (17) control. CHR$ (145) has no effect if the printer is already printing upper-case letters.
	Reverse off	CHR$ (146)	SHIFT + OFF RVS	Cancel reverse character printing, if in effect.
	Paging on	CHR$ (147)	CLR	Print 66 lines per page until a subsequent output to secondary address 3 changes the lines per page specification.

Table 6-5. Printer Format Characters, Used in Format Statements Printed in Secondary Address 2

Data Type	Code	Specification	Use	Examples		
				Data	Format	Printed Result
Numeric Fields	9	A numeric digit with leading zeros suppressed.	Specify all characters to the left and/or right of a decimal point (if present).	23 124.756 124.756	999.9 9999.99 999	23.0 124.75 124
	Z	A numeric digit with leading zeros printed.	Specify all characters to the left of a decimal point (if present).	23 124.756 124.756	ZZZ.9 ZZZZ.99 ZZZ	023.0 0124.75 124
	.	Decimal point	Numbers are aligned on the decimal point, if present. Numbers are right justified otherwise		See below	
	$	Number is a dollar amount	Print $ in front of first non-blank character, or in first character of the field	23 124.756	$999.9 $$$$$.99	$23.0 $ 124.75
	S	A signed number. S must be first format character.	Sign (+ or −) is printed as first non-blank character	124.756 −23 −475.2	S$999.99 S$$$$.99 S9999.9	+$124.75 −$ 23.00 +475.2
	−	A signed number. − must be last format character	Negative numbers are identified by a − printed as the last non-blank character	124.756 −23 −475.2	$999.99− $$$$.99− 9999.9−	$124.75 $ 23.00− 475.2−
String Fields	A	Any string field character position	Use an A to specify each string field character position String variables are left adjusted within the string field	ABCDE ABC ABCDE	AAAAAA AAAAAA AA	ABCDE ABC AB
Any	b	Character spaces between fields	Use one blank for every character position separating fields			
	<RVS ON>	Print the next character literally	The character in the format specification is printed		<RVS ON> − <RVS ON> −A	− −

When you run this program the word "MESSAGE" will be printed followed by two carriage returns. The PRINT# statement on line 25 generates the second carriage return.

Printing With CMD and PRINT Statements

After a CMD statement has been executed, PRINT statements will output data to the printer rather than the display until the next PRINT# statement is executed. To demonstrate this, change the printer program as shown below and run it:

```
10 OPEN 2,4
20 CMD 2
21 PRINT "MESSAGE"
21 PRINT#2
26 PRINT "MESSAGE"
30 CLOSE 2
40 STOP
```

When you run this program, the printer will execute a carriage return, then it will print the word "MESSAGE" followed by two carriage returns, then the word "MESSAGE" is displayed. The CMD statement on line 20 generates the first carriage return; the PRINT statement on line 21 causes the word "MESSAGE" to be printed by a carriage return. The PRINT# statement on line 25 generates the additional carriage return. The PRINT statement on line 26 displays the word "MESSAGE".

Now remove the PRINT statement on line 21. When you run the program again, the printer will execute two carriage returns, but the word "MESSAGE" is displayed; it is not printed.

A Comparison of CMD and PRINT# Statements

To understand what happened we must examine the slight difference between the effect of a CMD statement, as compared to a PRINT# statement.

Visualize the printer as a substitute for the display. A single output channel goes from the CBM computer either to the display, or to the printer. When an OPEN statement is executed specifying physical unit 4, the CBM computer is told that a printer is present, but the single output channel still selects the display.

When a PRINT# statement is executed subsequently, the output channel is deflected from the display to the printer; data in the PRINT# statement parameter list is transmitted to the printer, then the output channel selects the display again.

When a CMD statement is executed, the output channel is deflected from the display to the printer, data in the CMD statement parameter list is transmitted to the printer, but the output channel is left selecting the printer; the display no longer has an output channel.

When a PRINT statement is executed after a CMD statement, data is printed, not displayed, because the CMD statement has deflected the output channel from the display to the printer. But as soon as a PRINT# statement is executed, the output channel is deflected back to the display at the end of the PRINT# statement's execution. A PRINT statement executed after the PRINT# statement will again cause data to be displayed.

The printer must be closed, like any other logical file. When the CLOSE statement is executed, the CBM computer is told that the printer is no longer present.

If the output channel is left selecting the printer rather than the display when the printer is closed, then subsequent PRINT statements will continue to select the printer. To demonstrate this, enter and run the following program:

```
10 OPEN 2,4
20 CMD 2
30 CLOSE 2
35 PRINT "MESSAGE"
40 STOP
```

When you run this program you will see the following printout:

```
MESSAGE
BREAK IN 40
READY
```

The BREAK and READY lines which were previously displayed are now printed since the output channel was left selecting the printer.

FORMATTED PRINTER OUTPUT

CBM computer system printers will automatically format output for you.

First you must specify the printer format. You do this by transmitting an appropriate text string to the printer, using secondary address 2. Text string characters used to specify printer format are summarized in Table 6-5.

Data which is to be printed using the specified format must be output via secondary address 1. Data output in this fashion is printed using the most recently transmitted format specification. If no format has been specified, then data output using secondary address 1 is printed as transmitted — as it would be if output via secondary address 0.

To program formatted printer output, OPEN two logical files: one file selects physical unit 4 with secondary address 2; the other file selects physical unit 4 with secondary address 1. Then transmit format specifications and data using the appropriate logical file numbers.

Printing Formatted Numeric Data

We will begin by examining how the printer can format numeric data.

Character positions for each numeric field are specified using the digit 9, the letter Z, and optionally, a decimal point.

The decimal point, if included, will be printed wherever it appears in the numeric field. Numbers are aligned on the decimal point.

The digit 9 and the letter Z both specify numeric character positions. However the letter Z forces all zeros to be printed, whereas the digit 9 prints blanks for leading zeros. Here are some examples:

Number	Format Specification	Result
123.456	999999.99	123.45
6457		6457.00
−128.1		128.10
123.456	ZZZZZ.9	00123.4
6457		00123.4
−128.1		00128.1

A number can be printed with a preceding sign, or a trailing sign.

The letter S appearing at the beginning of the number field specification will cause a + or − sign to be printed at the beginning of the numeric field.

A minus sign (−) appearing as the last character of the numeric field specification will cause negative numbers to be represented by a trailing minus sign; no trailing plus sign is printed.

When a number is to be treated as a $ value then the $ sign can directly precede the number, or it can be aligned at the beginning of the allotted number field. The sign can precede the $ sign, it can follow the number, or the number can be unsigned.

For the simplest specification, add a $ character at the beginning of the numeric field format. This will cause a $ to be printed in the first (leftmost) character position of the numeric field. If the $ amount is to be printed with a + or − sign preceding the number, then the format must begin with S$; this will cause a + or − sign, and then a $ character, to be printed in the first two character positions of the numeric field.

You can also print $ amounts with leading zeros suppressed and a $ character appearing in front of the first numeric digit. For this specification specify all digit positions preceding the decimal point using $ characters; add one more $ character to specify the $ sign. Once again you have the option of putting an S at the beginning of the format in which case a + or − sign will precede the $ character.

Here are some examples of formats that include a sign and/or $ specification:

Number	Format Specification	Printed Result
123.456 6457 −128.1	S9999	123 6457 −128
123.456 6457 −128.1	S$9999.99	$0123.45 $6457.00 −$0128.10
123.456 6457 −128.1	S$9999.99	$123.45 $6457.00 −128.10
123.456 6457 −128.1	$$$$$.99−	$123.45 $6457.00 $ 128.10−
123.456 6457 −128.1	$$$$$.99−	$0123.45 $6457.00 $0128.10−

Later we will describe how you can substitute any other character or symbol for the $ sign if you are programming in a country that does not use $'s.

In order to demonstrate formatted numeric printout, key in program NUM.FORM.PRINT as listed below. This program reads eight miscellaneous numbers from the DATA statement on line 30, then prints them using the format specified by the PRINT# statement on line 100. When you run the program, a single column of numbers will be printed, as shown below the listing.

```
10 REM PROGRAM "NUM.FORM.PRINT"
20 REM DEMONSTRATE FORMATTED NUMERIC PRINTOUT
30 DATA 1.75,-12300,0.74682,12,-456.832,23456.78,-100.798,4789326
70 OPEN 1,4,1:REM OUTPUT DATA VIA LOGICAL FILE 1
80 OPEN 2,4,2:REM OUTPUT DATA FORMATS VIA LOGICAL FILE 2
90 REM OUTPUT DATA FORMAT
100 PRINT#2,"999999.99"
110 FOR I=1 TO 8
```

```
120 READ N
130 PRINT#1,N
140 NEXT I
150 CLOSE 1
155 CLOSE 2
160 STOP
```

```
     1.75
 12300.00
      .74
    12.00
   456.83
 23456.78
   100.79
******.**
```

Notice that numbers have been aligned on the decimal point. The eighth number will not fit within the specified numeric field. **Asterisks are printed in all digit positions when a number is too large for the specified format.**

Now change the PRINT# statement on line 100, substituting Z's for the 9's preceding the decimal point; re-run the program. Numbers are printed as follows:

```
000001.75
012300.00
000000.74
000012.00
000456.83
023456.78
000100.79
******.**
```

Notice that the Z's cause leading zeros to be printed. The eighth number still overflows the numeric field and is printed as asterisks. Add one more numeric digit position preceding the decimal point and the eighth number will be printed. Try it and see it for yourself.

You cannot mix Z's and 9's in the pre-decimal point field specification. If you do the printer will stop interpreting the field specification at the character change. For example, change the PRINT# statement on line 100 as follows:

```
100 PRINT#2,"ZZZ999.99"
```

Now run the program. Numbers will be printed as though the field specification were "ZZZZ". Now try changing the PRINT# statement on line 100 as follows:

```
100 PRINT#2,"9999ZZZ.99"
```

When you run the program, numbers are printed as though the specification were "9999".

Numbers have been printed unsigned. **In order to print a leading sign, change the PRINT# statement on line 100 as follows:**

```
100 PRINT#2,"S9999999.99"
```

Now run the program. Numbers are printed with a leading sign and suppressed leading zeros as follows:

```
+      1.75
-  12300.00
+       .74
+     12.00
-    456.83
+  23456.78
-    100.79
+4789326.00
```