Chapter 9

FILE HANDLING

9.1 TYPES OF FILE

The tape and disc filing systems on the BBC microcomputer are not restricted merely to storing BASIC programs. It is also possible to create files to contain data. The data could represent numerical information or text, and if required both types can be stored in a single file. This type of file is frequently called a textfile or data file and since, on some other computer systems, textfile is used to mean a file more like the ASCII file, we shall here refer to such files as data files.

It is worth pointing out straight away that the BBC computer operating system makes no intrinsic distinction between different types of file. They are all stored in the filing system, be it tape, disc or ROM, in exactly the same way.

Thus, any file can be loaded into memory (if there is room) by the command

```
*LOAD <filename> (<Load address>)
```

If the file happens to be a BASIC program, and the load address defaults to, or is given as, the current setting of PAGE, then it will be loaded and can be run just as if it had been loaded with the BASIC command LOAD "<filename>" .

Similarly, you can use the command *SAVE to save a BASIC program, if you know the right addresses to use.

What really creates an effective distinction between different types of file is the way that the information in the file is structured. This structuring will be carried out by the commands used to create the information or to store the information in the file. We can in this sense distinguish at least five separate file types

BASIC program
machine code program
binary file
data file
ASCII file

BASIC programs and machine code programs must have the information structured as the appropriate type of program *before* saving to a file.

Data files and ASCII files have their information structured (differently) by the commands which handle the respective types of file.

Binary files can contain any type of information, including the other four file types and also otherwise incomprehensible information such as graphics dumps.

We can deal fairly rapidly with the types of command associated with three of the file types.

BASIC programs

These are normally created by SAVE and loaded by LOAD or loaded and run in a single operation by CHAIN. These three commands are BASIC commands rather than operating system commands, so they are not preceded by a star, and the filename must be in between inverted commas.

Machine code programs

These can only be saved and loaded with the commands *SAVE, *LOAD and *RUN. The last command is equivalent to *LOAD followed by CALL, and plays a role similar to CHAIN for BASIC programs.

Binary files

Any section of memory, such as the area of a graphics display, can be saved and loaded using *SAVE and *LOAD (use of *SAVE and *LOAD are described in detail in Section 10.8.5). Thus to save a graphics picture in Mode 0, 1 or 2, the command would be

```
*SAVE GRDUMP 3000 8000
```

The picture could be loaded back simply by

```
*LOAD GRDUMP
```

The final two filetypes require special commands, and these are dealt with in the following sections.

Exercise 9.1

Enter a very short BASIC program such as

```
10 PRINT "*SAVE test"
20 PRINT "program compteted"
```

Save it to disc using the command

```
*SAVE TESTPG 1900 19FF
```

(assuming that you have a disc interface in your computer, so that &1900 is the normal program start position)

Type NEW to clear the program, then load it back again with

```
*LOAD TESTPG
```

LIST and RUN the program.

Exercise 9.2

Switch into Mode 1 and generate a simple graphics display, by commands such as

```
MODE 1
MOVE 1,1
DRAW 1000,1
DRAW 1000,1000
DRAW 1,1000
DRAW 1,1
DRAW 1000,1000
MOVE 1000,1
DRAW 1,1000
```

Use the commands given above to save the file with *SAVE, clear the screen with CLS, and then restore the display with *LOAD. (Note, however, that an interesting effect occurs if the screen display is scrolled between *SAVEing and *LOADing.)

9.2 HANDLING DATA FILES

9.2.1 Files and buffers

In BASIC, a set of commands is provided to facilitate handling of data files. Most of the commands can be used with cassette, single-user disc or level 2 Econet systems, and even the less common filing systems such as Prestel and IEEE.

One important difference between the handling of, say, a BASIC program file and a data file is that in the former case the saving or loading of a file is required to be carried out as far as possible as a continuous operation in the minimum of time, whereas for a data file these operations may extend over long periods. For example, in a data logging application, it may be necessary to record measured values of parameters in an experiment or industrial process at intervals of minutes or hours over a period of days. Similarly, where a file contains a large amount of data to be processed during execution of a BASIC program, it will be necessary to read the data in at irregular intervals as and when required by the program.

In order to minimize the number of disc operations, which relative to RAM operations are very slow, a section of memory is allocated for transfer purposes. This section is known as a buffer and has a capacity of 256 bytes, equivalent to one sector of the disc. All transfers between the computer and the disc must go through this memory. For writing to a file, the data in the buffer is transferred to the disc only when the buffer is full or at the end of file handling when the file is closed. In reading from a file, the buffer is filled and

no subsequent disc operation is needed until all the data in the buffer has been used by the program in INPUT or other statements.

The operation of setting up a buffer associated with a particular file is commonly referred to as *opening* the file. When operations on the file are completed, the buffer must be de-allocated which, not surprisingly, is called *closing* the file.

On the BBC computer, up to five files may be open at any one time; thus up to five buffers may be in use simultaneously.

This is one reason why you should always close a file as soon as transfer of data is finished. A much more important reason is that, when a file is being written to, the last, part-full buffer of information is not sent to the file until it is closed.

While a file is open, information is retrieved from or sent to the file by special versions of the normal input and output commands.

9.2.2 The BASIC commands to open, close and handle data files

The first step in handling a data file is to open the file for reading or writing. This is done by identifying and setting aside the buffer to be associated with a particular file by means of a *channel number*. The channel number is allocated by the operating system, and should be stored in a numeric variable (either integer or real) which will be referred to as the *channel variable*.

The channel number appears always to take the values 17 to 21 for the five files that may be open at any one time, but its actual value is immaterial to the user. The channel variable will always allow you to specify the file you want to work on simply and unequivocally.

OPENOUT

To open a file for writing (that is, to send output to the file), the function OPEN OUT is used. OPEN OUT returns the channel number to be associated with the file given as the argument to OPENOUT. The channel number should be assigned to a channel variable by a statement of the form

```
<channel variable>=OPENOUT(<filename>)
```

For example, if some data is to be written to a file called DFILE then the line

```
CH%=OPENOUT("DFILE")
```

can be used to get the system to allocate a buffer. (Note that as with most functions, brackets are conventionally included, but are actually unnecessary.)

The channel number to be associated with the file DFILE is allocated to the variable CH%, and CH% should be used with all subsequent file handling operations to inform the system that DFILE is the file to be written to.

A point to note is that with the disc filing system, the command OPEN OUT erases any existing version of DFILE, if it exists. If it does not exist, then a file named DFILE, 64 sectors long, is created. The filename given as the argument to OPENOUT may be either a string, such as "DFILE", or a string variable.

Strictly speaking a number could be used for the channel instead of a

variable, provided you were quite certain that you knew in advance what the correct number was for a particular channel, but it would be pointless as well as foolhardy since variables are actually more efficient than numbers within a program. Again it is possible, in principle, to open a file within one program, or even in direct mode, and use it within a subsequent program, but it is most unlikely that you will ever have reason to do this in practice.

PRINT#

The command to write information to a file has the form

```
PRINT#<channel>,variable,variable,...
```

For example

```
PRINT#CH%,B%,C$,"DATA"
```

Any variable, expression or value may be sent as output, just as with the normal PRINT command, but in other respects PRINT# differs. Items must only be separated by commas, and some commands such as SPC() and TAB() cannot be used.

CLOSE#

To empty and de-allocate the buffer, when writing is completed, the command required is

```
CLOSE#<channel>
```

For example

```
CLOSE#CH%
```

If instead of the channel variable the value 0 is used for the channel in the CLOSE command, then all open files will be closed. This is particularly valuable in an ON ERROR routine, to ensure that all files are closed if a program fails during execution. If you have not included such an error trap, the CLOSE command can also be used in direct mode for the same purpose, either specifying the actual file by its channel variable, or closing all files with a zero. It must be issued before the program is run again, or an error will occur when the computer attempts to reopen the still open file.

Example 9.1

The following procedure would write to a data file the results of a program which are stored in an array A(1) to A(20)

```
500 DEF PROC_writedata(F$)
510 CH%=0PENOUT(F$)
520 FOR I=1 TO 20
530 PRINT#CH%,A(I)
540 NEXT I
550 CLOSE#CH%
560 ENDPROC
```

A simple program to run this subroutine would be

```
10 DIM A(100)
20 FOR I=1 TO 20
30 A(I)=I*I: REM DATA IS FIRST 20 SQUARES
40 NEXT I
50 PROC_writedata("DFILE")
60 END
```

The procedure could equally well handle strings or integers if A(I) were replaced by A$(I) or A%(I).

Exercise 9.3

Write a program to store in a data file a list of the names of cars. Use the INPUT statement to read in the number of names and the names themselves. Run the program for, say, 5 names and use the *CAT command to verify that the data file has been created.

OPENIN

Once a file exists, it can be opened for reading with the OPENIN function, which is used in the same way as OPENOUT, as follows

```
CH%=OPENIN("DFILE")
```

Note that if DFILE does not exist, it will not be created. CH% will be set to zero and all will appear to be well, but when a command tries to use the channel number, it will fail with the error message

```
Channel at line...
```

This is particularly confusing because it is not really that line that is in error, but the line where the OPENIN was issued. It is useful in a situation like this to PRINT the value of CH% in immediate mode. If the value is zero it will wnfirm that a non-existent file is the source of error.

The fact that CH% is assigned a value of zero if the file does not exist could be used as a safety check, when a new file is to be created, to ensure that a file of the same name does not exist already before issuing an OPENOUT that would destroy an existing file.

INPUT#

Just as with output, the command to read data back from a file parallels the INPUT command

```
INPUT#CH%,A,B%,C$,D$
```

(Obviously, input has to be directly stored into variables, whereas values and expressions can be output to a file. Also, INPUT# is not a function, so it could not be used in an expression such as X=Y+INPUT#.)

Example 9.2

The equivalent to the procedure PROC_writedata, to read the data back, would be

```
500 DEF PROC_readdata(F$)
510 CH%=OPENIN(F$)
520 FOR I=1 TO 20
530 INPUT#CH%,A(I)
540 NEXT I
550 CLOSE#CH%
560 ENDPROC
```

It could be used with a program of the form

```
10 DIM A(100)
20 PRINT "The first 20 squares are"
30 PRINT '"      Number    square"'
40 PROC_readdata("DFILE")
50 FOR 1=1 TO 20
60 PRINT I,A(I)
70 NEXT I
80 END
```

End of File

It may well be that you do not know how many items of data there are in a file. There is a function, EOF#, which is designed for this purpose. It has the form

```
EOF#<channel>
```

and returns the value TRUE(-1) if the end of the file has been reached, and FALSE (0) otherwise. This is ideally suited to REPEAT...UNTIL loops, where it can be used as the UNTIL test.

Example 9.3

The procedure and program of Example 9.2 could be modified as follows to read back any number of values

```
 20 PROC_readdata("DFILE")
 25 I=I-1
 30 PRINT "The first ";I;" squares are"
 40 PRINT '"      Number    square"'
 50 FOR J=1 TO I
 60 PRINT J,A(J)
 70 NEXT
515 I=1
520 REPEAT
530 INPUT#CH%,A(I): I=I+1
540 UNTIL EOF#CH%
```

Exercise 9.4

Write a program to read the names of the cars stored in the data file from Exercise 9.3 and print out the list. Use the EOF# function to determine when the end of the file has been reached.

9.3 FURTHER ASPECTS OF DATA FILES

9.3.1 OPENUP

There are several reasons why you might want to open a file for writing without deleting the old version of the file. There is a third way to open a file, using OPENUP, which opens a file for updating (including both reading and writing) and does not delete the file first. The function is used in the usual way, for example

```
CH%=OPENUP("DFILE")
```

A serious complication occurs at this point between the two versions of BASIC that have been released by Acorn. These two versions are generally referred to as BASIC I (or just BASIC) and BASIC II respectively. To find out which version of BASIC your computer has, if you do not already know, press <BREAK> and then type

```
REPORT
```

The computer will respond either

```
(C)1981 Acorn
```

or

```
(C)1982 Acorn
```

The former message is issued by BASIC I, and the latter by BASIC II, which links the two dates very conveniently to the two versions of BASIC.

The above description of OPENOUT, OPENIN and OPENUP refers to BASIC II. Only OPENOUT is the same for BASIC I.

The other two commands are replaced by a single command, OPENIN, but worse is to follow. OPENIN in BASIC I has the same effect as OPENUP in BASIC II. There is no equivalent to the effect produced by BASIC II's OPENIN. Furthermore, a program containing OPENIN written with BASIC I will list the command as OPENUP on BASIC II. A program with OPENIN written with BASIC II will list nothing on BASIC I, and the command will cause an error.

If this seems totally confusing, Table 9.1 may help.

Table 9.1 Summary of commands to open files in BASIC I and BASIC II.

BASIC I	BASIC II	Token	Action
`OPENOUT`	`OPENOUT`	`&AE`	Delete existing file and open file for writing only
N/A	`OPENIN`	`&8E`	Open file for reading only
`OPENIN`	`OPENUP`	`&AD`	Open file for reading, writing or updating

(For the technically minded, what happens is that BASIC I and BASIC II use the same token − &AD − for OPENIN and OPENUP respectively. This is very sensible, since the action is the same in both cases, though the BASIC name is different. BASIC II uses a different token − &8E − that BASIC I does not recognize, for OPENIN.)

To make maximum use of the difference between the commands, and to minimize confusion, OPENUP will be used whenever a file is to be opened for both reading and writing. BASIC I users should remember that they must use OPENIN instead.

Programmers writing programs in BASIC II that may be used with either version of BASIC should *always* use OPENUP, even when opening a file for reading only.

9.3.2 Appending to a file: EXT# and PTR#

It is possible, but not very likely, that you may want to change the contents of a simple data file. Random access files, as discussed in Section 9.4, are really needed for this purpose. What you might well want to do, however, is to add further information to the end of a file, or *append* to a file.

Two further file handling commands will enable you to achieve this. The first of these is the function PTR#, which behaves like a pseudo-variable. As you write to, or read from, a data file, the computer keeps track of where you are up to by using a *pointer*, a variable which contains the current position in the file, measured in terms of the number of bytes from the start. This variable is accessible to you as PTR#<channeI>, and can be both read and, if you wish, altered.

The second command needed is EXT# (short for EXTent). EXT# <channel> is a function that returns the current size of the file associated with the specified channel.

We can now append to a file, by opening it with OPENUP and setting PTR# to EXT#. (Note that appending is not possible, and the commands EXT# and PTR# will not work, with the cassette filing system.)

Example 9.4

The following program will append the next ten square numbers to the file DFILE, created in Example 9.1.

```
100 CH%=OPENUP("DFILE"): REM USE OPENIN WITH BASIC I
110 PTR#CH%=EXT#CH%
120 FOR I=21 TO 30
130 PRINT#CH%,I*I
140 NEXT I
150 CLOSE#CH%
```

You can test the success of this program by reading back DFILE again with the program from Example 9.3.

9.3.3 Problems with data files on disc

There are two potential problems that can occur when using data files on disc, which is in many other ways much the most satisfactory filing system.

The first is associated with the fact that OPENOUT allocates 64 sectors to a file when it is first created. It may be that a disc has less than 64 sectors free, but ample room for the needs of your particular data file. The program will still not allow you to create the file, failing with a 'Disc full' error.

There is a simple way round this problem, associated with the fact that the operating system does not distinguish between different types of file, as explained in Section 9.1. You can create a 'dummy' file of a different type, and then reopen it with OPENOUT and overwrite it. The 64 sector reservation only applies to a brand new file.

The simplest way to create a dummy file is to use a command such as

```
*SAVE DFILE 0 10
```

that will create a tiny file. It will contain rubbish, but this is immaterial since it is to be overwritten. (You could equally well create a small BASIC program with SAVE.)

The second problem is more serious, and stems from the inefficient way in which the DFS stores its files in strict sequential order on the sectors of a disc. If, as is likely, a new file is saved to a disc following a data file, it will butt up against it on the disc, as shown in Figure 9.1.

The problem now arises that if you wish to append to the data file, there is no room to do so and the program will fail with

```
Can't extend
```

This is an infuriating message when you know that there is plenty of room left on the disc, and there is no foolproof way of avoiding it, except by making the data file the only file on the disc.

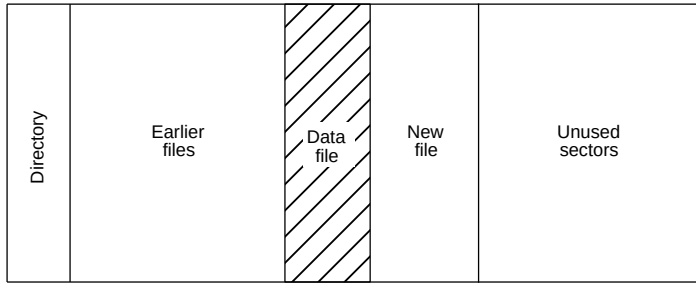

Figure 9.1 The problem with extending a data file.

A reasonably safe, but wasteful, way of ensuring that there is enough room is initially to create a file sufficiently large for all possibilities, by the same *SAVE method a described before. A file of 100 sectors could be created by

```
*SAVE LARGEFL 0 6400
```
(&64=100 decimal)

after which it should only be accessed by OPENUP or OPENIN.
However, you can no longer append to the file by the simple technique described above, since EXT# will point to the end of the 100 sectors, not the end of the genuine data.

Until Acorn change the DFS away from the strict sequential storage method there will be no really satisfactory solution to this 'Can't extend' problem.

9.4 RANDOM ACCESS DATA FILES

Data files can be further subdivided into two subcategories: sequential files and random access files. Up to now we have only discussed sequential or serial access to data files.

As ever on the BBC computer, random access files are simply data files where a very systematic method of organizing the data has been used, so that random access becomes possible. The key to random access is the function PTR# that has already been discussed. Note that random access is only possible with disc-based filing systems.

The simplest way to achieve random access is to organize the data into *records*, which are single or multiple items of data occupying a fixed number of bytes − the record length. Thus if a record length of 40 is chosen, then the Nth record can be found by setting

```
PTR#CH%=40*(N-1)
```

which immediately skips over the previous (N-) records.

There is a wide range of situations where data can be conveniently subdivided into records − phone lists, names and addresses, journal references, library catalogues, club membership lists, subscription accounts, stock lists, patient records and many similar record-keeping requirements. Quite clearly, anything that can be stored in a card index is ideally suited for a random access textfile.

In most cases, including those listed above, each record will contain more than one item, or *field*. These fields will be accessed sequentially, and the record acts in many respects like a very small sequential data file. In principle, each record could contain a different number of fields, though handling the file would be very difficult unless the records had some systematic structure, such as alternate records with the same numbers of fields.

9.4.1 Creating a random access file

Random access files are handled with the usual data file commands, plus PTR#. However, the record structure must be carefully planned. In particular, the record length must be made sufficiently large for all eventualities.

Consider as an example a simple inventory list, to contain three fields: item name, type and number in stock. The last field is quite simple, because it is numeric, and the internal format used to store numbers is such that integers occupy 5 bytes, and real numbers occupy 6 bytes. (It follows from this that files containing only real or only integer variables could be randomly accessed automatically, since the record length is always 6 or 5 respectively .)

String variables occupy the number of characters of the string plus 2 bytes; thus if the item name and type are considered to need up to 15 and 8 characters respectively, then assuming the stock number to be an integer, the record length required is (15+2)+ (8+2)+5=32 bytes. The record does not need to be full for every item; it must contain two strings and one integer but the strings could be shorter than 15 and 8 characters.

It is the standard length of each record that makes random access simple.

Example 9.5

The following program will set up such an inventory list.

```
 10 CH%=OPENOUT("INVENT")
 20 RECNO=0
 30 REPEAT
 40 INPUT "Item name: "ITNM$
 50 INPUT "Type:      "TPE$
 60 INPUT "Quantity:  "QU%
 70 PTR#CH%=RECNO*32
 80 PRINT#CH%,ITNM$,TPE$,QU%
 90 RECNO=RECNO+1
100 UNTIL ITNM$=""
110 CLOSE#CH%
```

The program will loop until <RETURN> is pressed with no other input in response to the item name. (It will also be necessary to press <RETURN> in response to the type and quantity, and this last empty record will be written into the file.)

The key lines are line 70, where PTR# is set to the start of the next record, and line 90 where the record number is incremented.

Exercise 9.5

A danger with the program above is that a name and type longer than the limits set may be entered, which will cause records to overlap. Modify the program to check the length of the fields before writing them to the file.

9.4.2 Reading a random access file

To read records back we need a program which is effectively the mirror image of that in Example 9.5, but it is necessary to know when to stop. As with Example 9.3, the EOF# function can be used for this purpose.

Example 9.6

The following program will read back the inventory produced in Example 9.5

```
 10 CH%=OPENIN("INVENT")
 20 RECNO=0
 30 REPEAT
 40 PTR#CH%=RECNO*32
 50 INPUT#CH%,ITNM$,TPE$,QU%
 60 PRINT "Item name: ";ITNM$
 70 PRINT "Type:      ";TPE$
 80 PRINT "Quantity:  ";QU%
 90 PRINT
100 RECNO=RECNO+1
110 UNTIL EOF#CH%
120 CLOSE#CH%
```

However, the whole point of random access files is to be able to read an individual record without first searching through the whole file and, even more importantly, to be able to modify or edit a record. For instance, with an inventory it must be possible to change the quantity, as items are drawn from stock or reordered. All that is necessary is to set PTR# to the desired record, as explained earlier.

Example 9.7

The following program allows the user to read, or write to, a specified record.

```
  10 PRINT "Do you want to read or wite (R/W)? ";
  20 INPUT " "RW$
  30 INPUT "Type the record number that you want: "RECNO
  40 IF RW$="R" THEN PROC_read(RECNO-1): REM SINCE RECORDS
     START AT 0
  50 IF RW$="W" THEN PROC_write(RECNO–1)
  60 END
1000 DEF PROC_read(RECNO)
1010 CH%=OPENIN("INVENT")
1020 PTR#CH%=RECNO*32
1030 INPUT#CH%,ITNM$, TPE$,QU%
1040 PRINT "Item name: ";ITNM$
1050 PRINT "Type:      ";TPE$
1060 PRINT "Quantity:  ";QU%
1070 PRINT
1080 CL0SE#CH%
```

```
1090 ENDPROC
2000 DEF PROC_wite(RECNO)
2010 CH%=OPENUP("INVENT"): REM USE OPENIN WITH BASIC I
2020 INPUT "Item name: "ITNM$
2030 INPUT "Type:      "TPE$
2040 INPUT "Quantity:  "QU%
2050 PTR#CH%=RECNO*32
2060 PRINT#CH% ,ITNM$,TPE$,QU%
2080 CLOSE#CH%
2090 ENDPROC
```

Exercise 9.6

Modify the above program so that when reading a record, it checks that the end of the file has not been reached.

Exercise 9.7

Modify the program further to allow it to read a record, then change the quantity field for that record.

9.5 SINGLE BYTE ACCESS TO A DATA FILE

9.5.1 The BGET# and BPUT# commands

For most purposes the normal data handling commands INPUT# and PRINT# are quite adequate, but there are circumstances when single byte access to a file is necessary. The commands to read or write a single byte are BGET# and BPUT# respectively. Each operates on the byte pointed to by the current value of PTR#, and PTR# is then incremented to point at the next byte in the file. The format of BGET# and BPUT# are rather different. Bytes are written by

```
BPUT#<channel> ,<number/variable>
```

For example

```
BPUT#CH%,10
```

or

```
BPUT#CH%,A%
```

Note that if any value exceeds 255, then only the least significant byte will be written. BGET# on the other hand returns a value, and must be PRINTed or assigned to a variable

BGET# on the other hand returns a value, and must be PRINTed or assigned to a variable

```
PRINT BGET#CH%
```

or

```
BT%=BGET#CH%
```

In the case of both BPUT# and BGET#, the pseudo-variable PTR# is incremented after the operation.

Exercise 9.8

Write a program to display the hex value of each separate byte in the data file from the previous exercises. Run the program and observe the format. An explanation of this format is given below.

9.5.2 Format of data files

Each item of data stored in a data file by the PRINT# command is preceded by a byte which indicates the type of data. A value of &40 indicates integer type and is followed by a further 4 bytes of data, which is the internal representation of the integer number. A value of &FF indicates real type and is followed by 5 bytes of data, the internal representation of a real number. A value of 0 indicates string type and is followed by a byte to indicate the length (L) of the string. A further L bytes follow which are the ASCII values of the characters in reverse order.

In order to position the pointer correctly in a data file containing, say, only integer variables, it is necessary to move the pointer 5 places for each item of data. Similarly for real variables the pointer must be moved 6 places per item. For string variables, however, the situation is more complicated since the lengths in general may not be the same. One solution to this problem is to pad out the strings to the same length by means of spaces at the end of each variable. The following instructions will pad out a string to 10 characters by using the STRING$ function to generate as many spaces as required.

```
10 INPUT "NAME",NAME$
20 L%=LEN(NAME$)
30 IF L%>10 THEN PRINT "Not more than 10 characters": GOTO 10
40 NAME$=NAME$+STRING$(10-L%," ")
```

Exercise 9.9

Write a program to write a list of names of cars to a data file with each string padded out to the same length. Use the program from Exercise 9.8 to verify that it has worked correctly.

Exercise 9.10

Write a program to read from the data file created in the previous exercise any specified item of data. Use an INPUT statement of the form "which car do you require (1 2 3...)?" and use the PTR# function to locate the required item.

9.6 A GENERAL PROGRAM TO READ OR WRITE A SEQUENTIAL DATA FILE

We are now in a position to write a general purpose program to read or write a sequential data file.

Writing is quite straightforward, using the techniques described in Sections 9.2 and 9.3. For mixed variables, it will be necessary to specify whether each is real, integer or string. Where all are of the same type, this need only be ascertained once.

Reading the file is where a problem arises, because we must cater for any type of data item. This is why this discussion needed to be postponed until this point. We can now use BGET# to interrogate each starting byte of an item to determine which type it is, so that the appropriate type of variable can be used to collect the input. (After the BGET#, the PTR# must be decremented to move it back to the start of the data item.)

Example 9.8

```
 10 ON ERROR GOTO 5000
 20 CLS
 30 PRINT "Do you want to WRITE or READ a data file?"
 40 INPUT "Please answer R or W: "A$
 50 PRINT
 60 IF A$<>"R" AND A$<>"W" THEN GOTO 40
 70 INPUT "Please type filename: "F$
 80 IFA$="R" THEN GOTO 500
 90 REM
100 REM ******** WRITE DATA FILE ********
110 REM
120 INPUT '"Do you want to append to the file? "YN$
130 YN$=LEFT$(YN$,1): IF YN$<>"Y" AND YN$<>"N" THEN VDU 7:
    GOTO 120
140 CLS
150 PRINT"Do you want to write:"
160 PRINT'"1) All real numbers"
170 PRINT'"2) All integer numbers"
180 PRINT'"3) All strings"
190 PRINT'"4) A mixture of data types"
200 INPUT'"Type the number for your choice: "DTYPE%
210 IF DTYPE%<1 OR DTYPE%>4 THEN VDU 7:GOTO 200
220 MODE 7
230 PRINT CHR$(136);"TYPE <ESC> TO TERMINATE INPUT"
240 PRINT'
250 *FX229,1
```

```
 260 REM CANCEL ESCAPE
 270 IF YN$="Y" THEN CH%=OPENUP(F$): PTR#CH%=EXT#CH% ELSE
     CH%=OPENOUT(F$)
 275 REM *** USE OPENIN FOR BASIC I ***
 280 REM APPEND IF YN$="Y"
 290 DONE=0
 300 REPEAT
 310 INP$=""
 320 IF DTYPE%=1 OR DTYPE%=2 THEN PROC_number(DTYPE%)
 330 IF DTYPE%=3 THEN PROC_string
 340 IF DTYPE%=4 THEN PROC_mixture
 350 UNTIL DONE
 360 CLOSE#CH%
 370 *FX 229,0
 380 REM RESTORE ESCAPE
 390 CLS
 400 PRINT "Writing completed"
 410 END
 470 REM
 480 REM ******** READ DATAFILE ********
 490 REM
 500 PRINT'"Do you want to pause after each"
 510 INPUT "field? (Y/N):"YN$
 520 YN$=LEFT$(YN$,1)
 530 IF YN$<>"Y" AND YN$<>"N" THEN VDU 7:GOTO 500
 540 CLS
 550 CH%=OPENIN(F$)
 560 REPEAT
 570 DTYPE%=BGET#CH%
 580 PTR#CH%=PTR#CH%-1: REM BACK TO START OF DATA ITEM
 590 IF DTYPE%=0 THEN INPUT #CH%,INP$:PRINT INP$
 600 IF DTYPE%=64 THEN INPUT#CH%,INP%:PRINT INP%
 610 IF DTYPE%=255 THEN INPUT#CH%,INP:PRINT INP
 620 IF YN$="Y" THEN A$=GET$:PRINT
 630 UNTIL EOF#CH%
 640 PRINT''"THAT IS THE END OF THE FILE"
 650 CLOSE#CH%
 660 END
1000 DEFPROC_number(DTYPE%)
1010 PROC_input
1020 IF DONE THEN ENDPROC
1030 NUM=VAL(INP$)
1040 IF DTYPE%<>2 THEN ELSE NUM%=INT(NUM):IF STR$(NUM%)<>INP$
     THEN INP$="":VDU 7:GOTO 1010
1050 IF DTYPE%=1 THEN PRINT#CH%,NUM ELSE PRINT#CH%,NUM%
1060 ENDPROC
1500 DEFPROC_string
1510 PROC_input
```

```
1520 IF DONE THEN ENDPROC
1530 PRINT#CH%,INP$
1540 ENDPROC
2000 DEF PROC_mixture
2010 PRINT "Which data type? (I-R-S): ";
2020 DTYPE$=GET$: PRINT DTYPE$;
2030 IF DTYPE$=CHR$(27) THEN DONE=1: ENDPROC
2040 REPEAT
2050 UNTIL GET$=CHR$(13)
2060 PRINT
2070 IF DTYPE$="I" THEN PROC_number(2): ENDPROC
2080 IF DTYPE$="R" THEN PROC_number(1):ENDPROC
2090 IFDTYPE$="S"THENPROC_string:ENDPROC
2100 VDU7:GOTO 2010
2500 DEF PROC_input
2510 CH$=GET$
2520 IF CH$=CHR$(13) THEN PRINT: ENDPROC
2530 IF CH$=CHR$(127) THEN PROC_cancel: ENDPROC
2540 IF CH$=CHR$(27) THEN PROCescape: ENDPROC
2550 PRINT CH$;
2560 INP$=INP$+CH$
2570 PROC_input
2580 ENDPROC
3000 DEF PROC_cancel
3010 IF LEN(INP$)=0 THEN VDU 7: PROC_input: ENDPROC
3020 PRINT CH$;: INP$=LEFT$(INP$,LEN(INP$)-1)
3030 PROC_input
3040 ENDPROC
3500 DEF PROCescape
3510 IF LEN(INP$)>0 THEN VDU7:PROC_input: ENDPROC
3520 DONE=1:ENDPROC
4970 REM
4980 REM ******** ERROR ROUTINE ********
4990 REM
5000 *FX 229,0
5010 REM RESTORE ESCAPE
5020 CLOSE#0
5030 PRINT "UNEXPECTED ERROR, NUMBER ";ERR;"AT LINE ";ERL
5040 END
```

This program consists of two main parts: the WRITE section starting at line 120, and the READ section starting at line 500. When writing, four data input options are offered: real numbers only; integers only; strings only; or a mixture of data types. Procedures PROC number and PROC string handle the actual input of a data item, and for the mixed option, PROC_mixture determines which data type is wanted next and then calls either PROC_number or PROC_string. Typing <ESCAPE> at the start of a data item (but not in the middle) terminates input to the file. There is also an option to

append to an existing file, which opens the file with OPENUP (or OPENIN for BASIC I) instead of OPENOUT, and uses EXT# to move the file pointer to the end of an existing file (line 270).

For reading files, a single option is offered, to pause after each field, so that a long file can be read (an alternative would be to use <CTRL-N>). The first character of each data item is read with BGET# at line 570 to determine the data type. The file pointer is then decremented back to the statt of the item at line 580, and the item read into the appropriate type of variable on lines 590, 600 or 610.

9.7 ASCII FILES

A more primitive form of data file is the ASCII file. Whereas items (basically the contents of integer, real or string variables) are stored as distinguishable, structured units in a proper data file, an ASCII file consists simply of a continuous stream of characters stored by means of the ASCII codes of the characters.

Because of the lack of structure, such files cannot be read with INPUT#. However, they can be created and read by certain other system commands, and even by BPUT# and BGET#.

9.7.1 Creating an ASCII file

The simplest way to create an ASCII file is by means of the *BUILD command. This has the form

```
*BUILD <filename>
```

Items are entered one ‘line’ at a time, where line here means, as with a BASIC program line, the entry up to the point the <RETURN> key is pressed, not a screen line. The computer prompts with a number for each line, but these are not stored in the file. Entry to the file is terminated, and the file closed, by pressing <ESCAPE>.

We shall see shortly that ASCII files are most commonly used as *EXEC files, but in principle *BUILD can be used to produce an ASCII file for any purpose.

An ASCII file can be read quite simply using one of three commands: *TYPE <filename> and *LIST <filename> are almost equivalent. They list back the ASCII file line by line, just as it was created by *BUILD, the only difference being that *LIST adds line numbers, in the same way that *BUILD does, whereas *TYPE does not. The third command is

```
*DUMP <filename>
```

Its effect is quite different. It produces a hexadecimal dump of the file contents, 8 bytes per line, together with the ASCII representation of each byte.

*DUMP can be used with any type of file, not just ASCII files. For instance, it could be used with a data file and would quite conveniently display the format used to store integers, real variables and strings. A specimen *DUMP listing of a data file is shown in Figure 9.2.

```
0000 00 09 53 52 4F 54 53 49 ..SROTSI
0008 53 45 52 00 04 4B 59 2E SER..K9.
0010 33 40 00 00 00 7D 00 00 3@...}..
0018 00 00 00 00 00 00 00 00 ........
0020 00 0A 53 52 4F 54 49 43 ..SROTIC
0028 41 50 41 43 00 05 46 6E APAC..Fn
0030 30 30 32 40 00 00 00 4B 002@...K
0038 00 00 00 00 00 00 00 00 ........
0040 00 0A 53 52 4F 54 49 43 ..SROTIC
0048 41 50 41 43 00 03 46 75 APAC..Fu
0050 31 40 00 00 00 19 00 00 1@).....
0058 00 00 00 00 00 00 00 00 ........
0060 00 00 00 00 00 40 00 00 00 ....@...
0068 00 ** ** ** ** ** ** ** .•..•.....
```

Figure 9.2 A specimen *DUMP listing.

9.7.2 *EXEC

*EXEC One of the main reasons, other than applications such as word processing, for wanting ASCII files is for use with the *EXEC command, which again has the simple form

```
*EXEC <filename>
```

The action of *EXEC is to issue the contents of the specified ASCII file to the computer *exactly as if the contents had been typed at the keyboard*.

There are three common uses for *EXEC.

1 Issuing a series of commands automatically

If you find that you repeatedly issue a particular set of commands, such as programming the red function keys, then these commands can be stored in an ASCII file by *BUILD. Issuing *EXEC for that file will then program the keys automatically.

Exercise 9.11

Use *BUILD to program the function keys to issue a series of commands, such as RUN<RETURN>, LIST<RETURN>, AUTO<RETURN>, RENUMBER<RETURN>, *RENAME, *DELETE. Test the file by *EXECing it. (You should see the commands appear on the screen as they are issued.)

Another example where it can be useful to issue not only commands but program input as well, is to run an interactive program automatically. This

could be useful to set up a file to control a program that takes a very long time to run, or to keep testing a program during debugging.

Exercise 9.12

Set up a file to run the program from Example 4.12 (the class list sort) automatically. You will need to start with

```
CHAIN "EX4.12"
```

then enter the marks one per line.

2 !BOOT files

As was discussed in Chapter 8, it is possible to make a disc into a turnkey system by adding a file named !BOOT to the disc.

The boot option can be set to *EXEC by

```
*OPT 4,3
```

and then, when <SHIFT-BREAK> is pressed, the file !BOOT is automatically *EXECed.

!BOOT can as usual be created by *BUILD. Quite commonly it will only be one or two lines long, say to CHAIN a BASIC program (unfortunately there is no direct boot option to CHAIN a program).

3 Issuing program lines

If a line of the ASCII file starts with a number, then when it is issued by *EXEC it will behave as it would if typed in; that is, it would be stored in memory as a program line. This gives a way of adding procedures or functions to a program, or merging programs, because unlike LOADing a new program section, the existing program remains in memory.

The required program lines could, if you wish, be created by the *BUILD command, but if you already have the program or procedure written, there is an easier way, using *SPOOL.

9.7.3 *SPOOL

The second normal method of creating an ASCII file is by using the *SPOOL command. This has two forms

```
*SPOOL <filename>
```

creates or opens the specified file, and thereafter any output to the screen from a BASIC command is also written into the file.

```
*SPOOL
```

with no filename closes the file or files. (The same effect can also be obtained with CLOSE#0.) Thus to capture into an ASCII file a program in memory (or part thereof), you would issue the commands

```
*SPOOL TXTPROG
LIST              (or LIST 100,200)
*SPOOL
```

If you subsequently issue *EXEC TXTPROG, all the listed lines will be reissued. Note that the ASCII file contains *everything* that appeared on the screen after *SPOOL TXTPROG, including the commands and BASIC prompts (>), so as well as the BASIC lines you will see the extra lines

```
>LIST
```
and
```
>*SPOOL
```

When *EXECed these will generate ‘Syntax error’ messages just as they would if you typed them (complete with leading >, that is), but this will not affect the program lines that you want.

Merging programs

The best method of merging two programs, or program sections, is by using *SPOOL and *EXEC. The procedure is as follows.

Load the shorter program into memory and capture it into an ASCII file by *SPOOL. Load the longer program into memory, then *EXEC the ASCII file to input the first program back into memory.

Note that if there are any duplicate line numbers the *EXECed lines will overwrite those already in memory. Interleaving lines would be all right in principle, but you are unlikely to want this. One or both programs should be RENUMBERed before merging to avoid this, if necessary.

The most common reason for merging program sections is probably to add to a program favourite procedures or functions, such as error checking or validation procedures. These can be kept as ASCII files and *EXECed into as many programs as you wish.

Editing programs with a word processor

The ability to *SPOOL a program into an ASCII file, and subsequently restore it with *EXEC, gives the possibility of using a word processor to edit BASIC programs. Word processors such as *Wordwise* operate with standard ASCII files, and once a program has been captured into an ASCII file it can be accessed by the word processor. All the powerful word processing facilities such as scrolling back and forth at will, global search and replace, and text movement or duplication can then be used to make program editing much easier. When editing is complete the program can be saved in ASCII form, and then *EXECed to retrieve it as a BASIC program.

Capturing program output in an ASCII file

Although *SPOOL is most widely used for capturing programs, it is not restricted to this. Another possible use is to capture program output.

As an example, consider the way in which short machine code subroutines are sometimes loaded into memory from within a BASIC program, by the ? indirection operator and DATA statements (the indirection operators are described in Section 10.4). The sort of program section required would be

```
1000 FOR J=0 TO 19
1010 READ X
1020 ?(BASEADD+J)=X
1030 NEXT J
1040 DATA 1,2,3,4,5,... 19,20
```

The numbers in the DATA statement(s) need to be worked out from the machine code, and if the code is already in memory, this could be done by a program, and the resulting values could be generated as a set of BASIC lines of DATA. *SPOOL can be issued from within a program to create an EXEC file. The following program shows how to do this.

Example 9.9

```
 10 INPUT "Start address of machine code: "SA$
 20 INPUT "Number of bytes: "NB$
 30 SA=EVAL(SA$): REM SA CAN BE IN DECIMAL OR HEX
 40 NB=EVAL(NB$)
 50 INPUT "Line number for start of DATA lines: "LI%
 60 DIM BT%(NB-1)
 70 FOR J=0 TO (NB-1)
 80 BT%(J)=?(SA+J)
 90 NEXT J
100 *SPOOL DATALNS
110 FOR K=0 TO (NB-1)/10
120 PRINT LI%+10*K;" DATA ";
130 J=0
140 REPEAT
150 PRINT ;BT%(J+10*K);
160 IF J<9 AND J+10*K<(NB-1) THEN PRINT ",";
170 J=J+1
180 UNTIL J>9 OR J+10*K=NB
190 PRINT
200 NEXT K
210 *SPOOL
```

Note that in contrast to BASIC commands, the name of the file following *SPOOL cannot be a string variable. This is because *SPOOL, like any

command starting with a star, is an operating system command which is sent directly to the Command Line Interpreter. The CLI cannot interpret BASIC variables, since it operates at a lower level quite independently of whether BASIC is even fitted to the computer. There is a way round this problem, however, as we shall see in Section 10.7.1.

9.7.4 Single byte access to ASCII files

The point has already been stressed that the only difference between different file types on the BBC microcomputer is their internal structure. The general approach to handling data files and ASCII files is very similar. Both operate through buffers, transferring information to or from the filing system a sector at a time. The main difference lies in the special way in which the PRINT# command writes information to the file, so that it can only readily be interpreted by the INPUT# command.

However, this is not true of the single character commands BPUT# and BGET#, which simply transfer a single character to or from the filing system. They can therefore be used to create or read an ASCII file as easily as a data file. This is particularly relevant for reading an ASCII file, since the three normal commands, *LIST, *TYPE and *DUMP all have limitations. *LIST and *TYPE may hang up if they try and read certain control characters, for instance CTRL-B which tries to turn on a printer. *DUMP will certainly list any type of file, but the hexadecimal dump makes it tedious to use with files containing text that you would like to read.

Example 9.10

The following program will write an ASCII type of file, or more accurately any file of characters that can be typed in from the keyboard, including control characters and Mode 7 special effect codes. More importantly it will read them back.

```
 10 ON ERROR GOTO 900
 20 CLS
 30 PRINT "Do you want to WRITE or READ an ASCII"
 40 INPUT "file? Please anwer W or R: "A$
 50 PRINT
 60 IF A$<>"R" AND A$<>"W" THEN GOTO 40
 70 INPUT "Please type filename: "F$
 80 IF A$="R" THEN GOTO 500
 90 REM
100 REM ******** WRITE ASCII FILE *******
110 REM
120 INPUT '"Do you want to append to the file? "YN$
130 YN$= LEFT$(YN$,1): IF YN$<> "Y" AND YN$<>"N" THEN VDU 7:
    GOTO 120
140 MODE 7
150 PRINT CHR$(136);"TYPE <ESC> TO TERMINATE INPUT"
160 PRINT'
170 *FX 229,1
```

```
180 REM CANCEL ESCAPE
190 IF YN$="Y" THEN CH%=OPENUP(F$): PTR#CH%=EXT#CH% ELSE
    CH%=OPENOUT(F$): REM USE OPENIN WITH BASIC I
200 REM APPEND IF YN$="Y"
210 REPEAT
220 CH=GET
230 IF CH<>27 AND CH<>127 THEN PRINT CHR$(CH);: BPUT#CH%,CH
240 IF CH<>127 THEN ELSE IF PTR#CH%>0 THEN PRINT CHR$(CH);:
    PTR#CH%=PTR#CH%-1 ELSE VDU 7
250 IF CH=13 THEN PRINT CHR$(10);: REM CR NEEDS LF ALSO
260 UNTIL CH=27
270 CLOSE#CH%
280 *FX 229,0
290 REM RESTORE ESCAPE
300 CLS
310 PRINT "Writing compteted"
320 END
500 REM
510 REM ******** READ ASCII FILE ********
520 REM
530 CLS
540 PRINT TAB(0,4);"Option List:"
550 PRINT '"1) Suppress high bit of each byte"
560 PRINT '"2) Suppress CONTROL codes"
570 PRINT '"3) Send output to printer"
580 PRINT '"4) Pause after each paragraph"
590 PRINT '"5) Start listing"
600 OP%=0
610 REPEAT
620 INPUT '"Type the number for your choice: "OP
630 IF OP=1 THEN OP%=(OP% OR 1): PRINT '"HIGH BITS WILL BE
    SUPPRESSED"
640 IF OP=2 THEN OP%=(OP% OR 2): PRINT '"CONTROL CODES WILL BE
    SUPPRESSED"
650 IF OP=3 THEN OP%=(OP% OR 4): PRINT '"PRINTER ENABLED"
660 IF OP=4 THEN OP%=(OP% OR 8): PRINT '"PRESS A KEY AFTER
    EACH PARAGRAPH"
670 IF OP<>INT(OP) OR 0P>5 THEN VDU 7
680 UNTIL OP=5
690 CLS
700 IF (OP% AND 4)=4 THEN VDU 2
710 CH%=OPENIN(F$)
720 REPEAT
730 CH$=CHR$(BGET#CH%)
740 IF (OP% AND 1)=1 AND ASC(CH$)>128 THEN CH$=CHR$(ASC(CH$)-128)
750 IF (OP% AND 2)=2 AND ASC(CH$)<32 AND CH$<>CHR$(13) THEN
    CH$="."
760 PRINT CH$;
```

```
770 IF CH$=CHR$(13) THEN PRINT CHR$(10);: IF (OP% AND 8) = 8
    THEN A$=GET$
780 UNTIL EOF#CH%
790 VDU 3
800 PRINT ''"THAT IS THE END OF THE FILE"
810 CLOSE#CH%
820 END
900 REM
910 REM ******** ERROR ROUTINE ********
920 REM
925 REPORT:PRINT " AT ";ERL:END
930 *FX 229,0
940 REM RESTORE ESCAPE
950 CLOSE#0
960 PRINT "UNEXPECTED ERROR, NUMBER ";ERR;" AT LINE ";ERL:
    VDU 3
970 END
```

When reading a file the program offers a number of options. The first is to suppress the high bit of any byte. Some files may contain characters in a modified ASCII form, having 128 added to the normal code, and this option will strip off the 128.

The second option allows you to suppress any control characters (except the <RETURN> character), printing a full stop instead. The third and fourth options are self-explanatory, allowing you to list to a printer or pause after each paragraph (or more accurately, after each <RETURN>) until any key is pressed. Note that if your printer requires setting up by commands such as *FX 6,0 these will have to be issued before running the program.

One particular use for this program is to read word processor files if you have access to them but do not yourself possess a word processor. Word processor files often contain control codes as well as simple text, and you can use the option to suppress these in order to read the text. The popular word processor Wordwise in particular can create files containing CTRL-B codes that will hang the computer unless a printer is connected.

Exercise 9.13

One problem when reading long strings, such as those from word processor files, is that words are broken at the end of each line. Use the type of technique introduced in Example 4.10 to modify Example 9.10 so that when long strings are read, words too long to fit on the end of a line of screen display are wrapped round onto the next line.