Chapter 10

DIRECT MEMORY AND SYSTEM ACCESS

10.1 COMPUTER MEMORY

10.1.1 Hexadecimal notation

The simplest counting system is binary, in which numbers are represented in terms of units, twos, fours, eights and so on, instead of the familiar decimal system where successive digits represent units, tens, hundreds etc. In binary the number nine would be represented as 1001 (1×8+0×4+0×2+1×1) and all numbers are written in binary as a simple string of zeros and ones − two symbols only, which is why it is the simplest, though most long-winded system. This is also why it is used in computers − the two possible values can easily be represented by two alternative states, ON and OFF, or two different voltage levels.

However, binary numbers are very long and inconvenient to work with. The hexadecimal system of numbers is frequently used as a more convenient alternative to binary numbers. This is a number system to base sixteen, or 2^4. The bits (Binary digiTS) are grouped into fours and each group of four is represented by a single hexadecimal digit. For instance, 10010011 becomes 1001 0011 which becomes 93 in hexadecimal (called nine-three because ninety-three implies nine tens and three). A problem occurs with numbers like 10011011 which splits into 9 and 11. We obviously cannot write this as 911 as that represents 100100010001. Conventionally the numbers ten to fifteen are replaced with the letters A to F, in order to represent them by a single ‘digit’. Thus the number 10011011 is written, not 911 but 9B in hexadecimal.

Care has to be taken to distinguish between hexadecimal and decimal numbers. 9B is fairly obviously hexadecimal, but the previous example of 93 is not. It is common to indicate that a number is hexadecimal by using a prefix. On the BBC computer the & sign is used for this. Thus our number should be written as &93, which makes it quite clearly hexadecimal. Its decimal equivalent can be calculated by remembering that the 9 represents the 16s column, so it is 16×9+3, or 147. Similarly, &9B is 16×9+11, or 155.

The microprocessor used in the BBC computer is the 6502A, which is an 8-bit microprocessor; that is, it deals with 8 bits of data at a time. This collection of bits is called a *byte*, and can have values from 0 to 255, or &00 to &FF. Thus the hexadecimal system is a very convenient notation when dealing with information in bytes, since any byte can be written as a two-digit hexadecimal number.

The BBC memory stores information one byte per memory location. In order to be able to locate an individual memory location, it is necessary for that location to be specified by a unique *address*. Hexadecimal numbers are used for addresses, and since 256 memory locations would be quite inadequate, 2 bytes are used to specify a memory address, giving a range of 256^2 or 65536. Memory addresses thus run from &0000 to &FFFF. 65536 is usually referred to as 64K, where $1K=1024=2^{10}$. (The decimal quantities have apparently strange values as they are the equivalents of ‘round’ binary numbers.)

Computer memory can be one of two types: read only memory (ROM) or read/write memory (RAM − from the less accurate name random access memory). ROM, as the name states, contains data permanently stored on a chip, which can be read but cannot be altered by the computer. The BASIC interpreter and the Machine Operating System are stored in ROM, from &8000 to &FFFF. RAM, on the other hand, is memory which can be read or altered (written to), and this is where BASIC programs, variables and graphics are stored.

10.1.2 Tilde and ampersand

The BBC computer can handle and print numbers in both decimal and hexadecimal form. A tilde (~) in front of a number or expression in a PRINT statement will cause it to be printed as a hexadecimal number (but without the & − since you put the tilde in, you presumably realize that the number is hexadecimal). A number may be given in hexadecimal form by preceding it with an ampersand, wherever an expression can be used. Thus you can PRINT (in decimal) a number given in hexadecimal, or assign it to a variable. You cannot directly INPUT a number in hexadecimal, but you can input it as a string and use EVAL to obtain its value. Try the following example to demonstrate this.

Example 10.1

```
10 REPEAT
20 INPUT "Number: "X$
30 X=EVAL(X$)
40 PRINT "Hexadecimal: ";~X
50 PRINT "Decimal    : ";X
60 UNTIL X=0
```

Run the program first by entering decimal values in the range 1 to 65535 and then in the form &1 to &FFFF. Predict the output for a few values such as 165, &23 and test your expectations using the program. Input a zero to finish.

Note that in Mode 7, the tilde will appear on the screen as a divide sign (remember that normal division is done with the solidus, /).

10.2 ORGANIZATION OF MEMORY IN THE BBC MICROCOMPUTER

It can sometimes be very useful to know how the memory of the BBC computer is organized. Figure 10.1 shows the allocation of the various parts of memory, drawn to scale, and a more detailed diagram of the first section.

In hexadecimal the addressing range of the computer (&0000 to &FFFF) is sometimes regarded as being a series of pages, each &100 (256) locations long, so that the highest two hexadecimal digits give the page number and the lowest two digits give the location within the page.

10.2.1 Memory usage by the system

Important and interesting areas of memory used by the system are as follows

- Zero page (&00 to &FF) A very important area for machine code programs, and used intensively by all parts of the system. &70 to &8F are reserved for the BASIC user, but note that this area may be used by other languages such as Pascal or *Wordwise*.
- Page 1 (&100 to &1FF) Another important area for machine code programmers, containing the 6502 stack.
- Pages 2 and 3 (&200 to &3FF) Used as workspace (places to store values in) by various parts of the system.

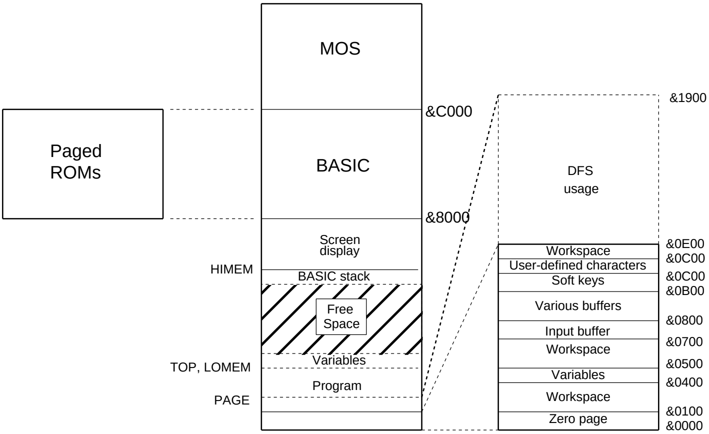

Figure 10.1 Organization of memory in the BBC microcomputer.

- Page 4 (&400 to &4FF) This page is used for two purposes. The first half stores the resident integer variables, and the second half stores the pointers to the linked chains of variables. An individual chain consists of all those variables that start with a particular letter.
- Pages 5 and 6 (&500 to &6FF) Further workspace.
- Page 7 (&700 to &7FF) The input buffer, used to store keyboard input until <RETURN> is pressed.
- Pages 8 to 10 (&800 to &AFF) Mostly used as buffer space for sound, printer, RS423 I/O, cassette I/O and speech. Pages 9 and 10 would be safe to use except when using the RS423 output port, a cassette recorder or speech.
- Page 11 (&B00 to &BFF) This is where the text loaded into the soft keys is stored.
- Page 12 (&C00 to &CFF) This page contains the definitions of the normal user defined characters (224 to 255). (If extra characters are user defined, the start of user space is raised to make room for the extra data.)
- Page 13 (&D00 to &DFF) Used for outside events and by the paged ROMs.
- Address &E00 is the start of user space for the cassette filing system.

10.2.2 Memory usage by the DFS

The disc filing system normally uses up a further 11 pages of memory

- Pages 14 and 15 (&E00 to &FFF) A copy of the current catalogue sectors (track 0, sectors 0 and 1) are stored in these two pages. Note that as long as a disc drive continues to run after a disc access, the DFS assumes that these pages are valid, so if you swap over a disc while the drive is still running, any command issued before the drive stops will use the wrong catalogue information, with potentially disastrous results.
- Pages 16 and 17 (&1000 to &11FF) These two pages are used as DFS workspace.
- Pages 18 to 22 (&1200 to &16FF) These five pages are used as buffers for the five possible files that may be open at any one time (for use with the data file commands or *SPOOL, for example). Note that buffers are *not* used for LOADing and SAVEing BASIC programs or binary files.
- Pages 23 and 24 (&1700 to &18FF) These two pages are used for further workspace.

If a user is desperate for more space for his program, it is possible to recover some of this memory by changing the value of the pseudo-variable PAGE (see Section 10.3). Those pages in excess of the maximum number of files he may want open simultaneously can be used, and if he is not using data or ASCII files at all it is possible to set PAGE as low as &1100, and still be able to load and save BASIC programs with reasonable safety.

The Econet filing system uses just four pages of memory, up to &1200,

unless a DFS is also fitted, in which case the two systems together use memory up to &1B00.

10.2.3 VDU display space

The memory used for screen display depends on the mode, and is summarized in the following list

Modes 0 to 2	&3000 to &7FFF (20K)
Mode 3	&4000 to &7FFF (16K)
Modes 4 and 5	&5800 to &7FFF (10K)
Mode 6	&6000 to &7FFF (8K)
Mode 7	&7C00 to &7FFF (1K)

The top half of memory is allocated to ROMs. &8000 to &BFFF is used by all the paged or ‘sideways’ ROMs (so called because their memory sits side by side); this includes BASIC and the DFS ROMs. Most of &C000 to &FFFF is used by the Machine Operating System, but &FC00 to &FEFF is allocated for input/output. This memory is used for transferring information to or from the computer through the user port, the printer port and so on.

10.2.4 Program usage

The start of the available memory space depends on the filing system in use, and is referred to as the OSHWM (Operating System High Water Mark). For cassettes it is &E00, for discs &1900, and for Econet it is &1200 or &1B00 if a disc interface is also fitted.

BASIC programs usually start from this point, and build up in memory as required. Variables are stored immediately after the program itself, and BASIC requires some additional workspace, called the BASIC stack, which is normally situated immediately below the memory allocated to the displayed text or graphics. The extent of this memory will depend on the mode being used. Variables can thus build up as far as the bottom of the BASIC stack, which will normally occupy only a small amount of space. If a program, or its variables, tries to expand beyond this point, the error message ‘No room’ will occur. Trouble can also occur if a large program tries to change mode to one which requires more memory than is available, which will result in the ‘Bad Mode’ error.

10.3 THE MEMORY POINTERS

There are four important positions in memory for BASIC programs which are accessible from BASIC. The first, and most important, is PAGE, the position of the start of the program. The next is TOP, the position of the end, or top, of the program. The third is LOMEM (LOw MEMory), which is the start of the variable storage area. Finally, HIMEM (High MEMory) gives the last available location in memory for program usage. The BASIC stack builds down in memory from this point.

All of these pointers except TOP are pseudo-variables, and all can be altered as well as read except for TOP.

PAGE normally points to the OSHWM which, as pointed out above, is &E00 for the cassette filing system and &1900 for the disc filing system. However, it is possible to move PAGE either up or down. Note that PAGE (but not the other three pointers) must always be set to a page boundary, that is a value which in hexadecimal has the lowest two digits zero. If you try and set PAGE to any value other than a page boundary, the computer will ignore the lowest two hexadecimal digits. Thus if you tried to set it to &1B80 (7040), it would in fact become &1B00 (6912).

TOP always points to the first free memory location above the program. It is the one pointer which BASIC does not allow to be altered by the user, for fairly obvious reasons. The length of a program can always be calculated by (TOP-PAGE). Note that although you cannot alter TOP by a simple assignment, this restriction can be overridden using the indirection operators described in the next section. If you want to set the value of TOP to NEWTOP, then the following statements will accomplish this

```
?18=NEWTOP MOD 256
?19=NEWTOP DIV 256
```

Great care should be taken in doing this, since setting TOP too low will destroy the end of your program.

LOMEM normally has the same value as TOP, since the sensible place to start the storage of variables is immediately above the program. It can be altered to a different value if there is a reason to do so. You may alter it from within a program if you wish, but only before variables have been assigned, otherwise their values will be lost.

HIMEM points to the next address above the last memory location available to BASIC. It normally points to the bottom of the screen display area of memory, but can be altered, for example to create a safe area for a machine code program. The amount of free space above a program available for variables and the BASIC stack can easily be calculated by (HIMEM-LOMEM).

Exercise 10.1

Write a very short program to give the values of the four memory pointers, of the form

```
10 PRINT "PAGE = ";PAGE
   ...
   ...
```

Also print out the size of program (TOP- PAGE) and amount of free variable space (HIMEM-TOP). (The latter also tells you the amount of space available for program expansion.)

Switch into all the different modes, starting with Mode 7, and notice the dramatic reduction in available space when switching between, say, Mode 7 and Mode 1.

Exercise 10.2

Try altering PAGE to a value other than a page boundary, for example

```
PAGE=&2345
```

then PRINT ~PAGE to see what value has actually been assigned. Press the <BREAK> key to reset the machine, PRINT PAGE again, which will now give the default value, and make a note of what it is for your machine.

Note that whenever you press <BREAK>, PAGE is restored to its default value, so if you alter it deliberately and then load or type in a program, the program will be temporarily lost by a <BREAK>. It can be recovered, however, if you set PAGE once more and then type OLD.

There are two other pointers which it can be useful to read, but which are not accessible as pseudo-variables. These are the pointers to the top of the occupied variable space, and the bottom of the BASIC stack. The pointers are stored in zero page at addresses &02, &03 and &04, &05 respectively. To read them, simply type

```
PRINT "TOP OF VARIABLES: ";?2+256*?3
PRINT "BOTTOM OF STACK: ";?4+256*?5
```

(The use of the indirection operator ? above is explained in Section 10.4.1.) Of more interest than either individual pointer is the size of the gap between them, since this is the amount of free memory available. To calculate this, type

```
PRINT "FREE MEMORY: ";?4-?2+256*(?5-?3)
```

10.3.1 Loading more than one program into memory

One of the uses of PAGE is to allow more than one program to reside in memory simultaneously.

Provided that you set PAGE for a second program above the value of TOP for the first, they will not overlap, and you can move back and forth between them simply by altering PAGE (if the first program needs a lot of space for variable or array storage, you will have to allow for this as well when setting PAGE for the second program, or you could use LOMEM to force the variable space for the first program above the second program).

Exercise 10.3

Enter a short program which prints some message such as ‘First program’ and test that it runs correctly. Now set PAGE in immediate mode to, say, &3000 and type NEW. Then enter a second program to print ‘Second program’ and test that. Restore PAGE to the value that you noted down and RUN. The first message should be printed, since that program is still safely stored at that location. Finally, set PAGE to &3000 again and RUN, and the second message should be printed.

A number of short programs can be entered in this way, but note that NEW must be typed after PAGE is changed to a new location for the first time, to set various pointers needed by the interpreter, or a ‘Bad program’ error will occur. You must also always be careful that neither program nor variable ;torage areas overlap, and that you do not set PAGE incorrectly to the middle of a program.

It is also possible to change PAGE within one program, and then to CHAIN another program from within that program. The second program will be loaded at the new setting of PAGE (the new value of PAGE comes into effect when RUN, LIST or some similar command is encountered).

Exercise 10.4

Try the following series of programs

```
10 PAGE=&2000
20 PRINT "First program"
30 PRINT "Now we will run the second program"
40 RUN
```

Now enter two more programs by

```
PAGE=&2000
NEW

10 PRINT "Second program now"
20 PAGE=&3000
30 PRINT "PAGE does not come into effect yet"
40 RUN

PAGE=&3000
NEW

10 PAGE=&1900: REM ASSUMING YOU ARE USING A DISC SYSTEM
20 PRINT "Now back to the beginning"
30 RUN
```

Having entered the three programs, return to the first one and start execution by

```
PAGE=&1900      (or whatever your default value is)
RUN
```

If programs stacked up in memory in this fashion used substantial amounts of variable space, you could use LOMEM to force all the programs to share the memory space above the topmost program, to avoid the possibility of variables from one program corrupting the one above it.

10.3.2 Relocating a BASIC program

It can happen that memory constraints are so tight that a program does not have enough memory when loaded at the normal setting of PAGE. On a computer fitted with a disc or Econet interface, it is possible to relocate a program after it has been loaded and move it down in memory. To get the maximum amount of space will mean overwriting the disc and/or Econet workspace, but once the program has been loaded this need not matter. The most common reason for needing to do this is if you have a program working on tape, that will not work when transferred to disc. With the DFS the lowest address that you can use without interfering with program transfer is &1100. To retain one buffer for data transfer, you must keep above &1300, and so on. The lowest possible address is &E00, which would render the DFS unusable, but generally it would be sensible only to lower PAGE as far as is necessary for the program to run.

Example 10.2

The following lines, added to the beginning and end of your program, will cause the program to relocate itself automatically when run.

```
    0 N%=&1100: O%=PAGE: REM NEW VALUE FOR PAGE. CHANGE AS
      DESIRED
    1 IF PAGE>N% THEN GOTO 32000
    2 LOMEM=TOP
      ...
      ...
      ...
31990 PAGE=O%: END
32000 A%=TOP-PAGE: REM LENGTH OF PROGRAM
32010 FOR B%=0 TO A% STEP 4: B%!N%=B%!PAGE: NEXT
32020 ?19=?19-(PAGE-N%) DIV 256: REM RESET TOP
32030 PAGE=N%
32040 GOTO 1
```

The main relocation loop must be at the end of the program, because as the program is moved down in memory, the early part will be overwritten. For maximum speed the loop is written on a single line, and uses the resident integer variables, as described in Section 10.6. It also uses the ‘pling’ indirection operator, which is described in the next section. This is a case where it is also necessary to change the value of TOP.

These program lines could very conveniently be captured into an ASCII file with *SPOOL, and then added to any program requiring relocation with *EXEC.

10.4 MEMORY ACCESS

10.4.1 The indirection operators

Most versions of BASIC provide a means of directly accessing memory by a pair of commands PEEK and POKE for reading and writing respectively. On the BBC microcomputer, these have been superseded by a set of *indirection operators*, ? (query), ! (pling) and $ (dollar). They act on different numbers of bytes, but otherwise are all identical in operation. Their operation can best be understood by interpreting them as meaning ‘the contents of memory location(s)’, and they are used for both reading and writing. To read a byte of memory at location M and print out the result, we can use the command

```
PRINT ?M
```

or if you want the result in hexadecimal

```
PRINT ~?M
```

To understand the meaning of this more clearly, remember the meaning to be associated with ?, and read this command as

`PRINT` the contents of memory location M.

Alternatively, the contents of a memory location can be loaded into a variable. For instance, to read the contents of location &FE61 into the variable RD, the command needed is

```
RD=?&FE61
```

Writing to memory is just the reverse of this command; to write a &FF to location &FE61, the command is

```
?&FE61=&FF
```

This time we can understand the purpose of ? as ‘the contents of memory location &FE61 (becomes)=&FF’. Variables can be used for either address or contents. For example

```
?M=WR
```

would write the value of variable WR into location M.

Thus we see that ? serves the purpose of both PEEK and POKE, according to which side of the assignment it is on.

The ! (pling) operator works in exactly the same way, except that it transfers not one but *four* bytes at a time to or from memory.

Note that the value is transferred starting from the *lowest* byte, and that the split between the four bytes will be difficult to work out unless the number is in hexadecimal, when two digits will go into each byte.

Exercise 10.5

Try moving values to and from memory with the ? operator. Memory locations around &2000 would be suitable.

Transfer four bytes into memory locations &2000 to &2003 with the command

```
!&2000=&11223344
```

and then read them back one at a time with ?. Try the same thing with decimal rather than hexadecimal values.

The third indirection operator is the string operator $ (dollar). This transfers a string of up to 255 characters to or from memory. When writing to memory, a string or string variable can be transferred. A <RETURN> character is written at the end of the string to mark its end.

When reading from memory, all the characters up to the next occurrence of a <RETURN> character are read back. Thus a string written by $ can be read back in the same way.

Note that $ is used here in quite a different way from its usual purpose of denoting a string variable. $M means the string contents of memory starting at address M, whereas A$ is a variable which contains a string.

Exercise 10.6

What is the meaning of

1. `$M=A$`
2. `M$=$A`

Exercise 10.7

Write a string into memory by

```
$&2000="123456789"
```

Read it back first by

```
PRINT $&2000
```

and then using the ? operator by

```
FOR J=0 TO 9: PRINT ~?(&2000+J): NEXT J
```

In the latter case you will see the ASCII codes &30 to &39 corresponding to 0 to 9, followed by &0D which is the ASCII code for the <RETURN> character.

10.4.2 Indexed addressing

Both ? and ! have a useful variation which will immediately be recognized by machine code programmers as *indexed addressing*. If either indirection operator is *preceded* by a variable (a number will not do in this context) and *followed* immediately by another variable or a number, then the location addressed is the sum of the two quantities. For example

```
ADDR?10
```

means ‘the contents of location (ADDR+ 10)’.

Example 10.3

At first sight this use seems to be confusing and pointless, but consider the following small program

```
10 ADDR=&2000
20 FOR J=0 TO 9
30 PRINT "Location ";~ADDR+J;" contains ";~ADDR?J
40 NEXT J
```

In line 30, ADDR?J is exactly equivalent to ?(ADDR+J).

Exercise 10.8

Write a program, using indexed addressing, that prompts for a starting address and a string, and then writes the string into memory as a series of ASCII codes, ending with &0D for the <RETURN> character.

Test the operation of your program by reading the string back with the $ operator.

10.5 A MEMZAP AND DISCZAP PROGRAM

The indirection operators can be used in a MEMZAP program to give easy access to memory, both for reading and modifying. There are a number of possible uses of such a program: the study of how BASIC programs and variable storage are carried out; loading files of any sort with *LOAD and modifying them in memory, after which they can be *SAVEd; or examining the low areas of memory used as buffers. Because the program is very close to what is required for a disc editing program, the same program also provides a DISCZAP facility. Use of this latter application is discussed in Appendix M.

Program ZAP is listed below

```
 10 MODE 7
 20 HIMEM=&7B00
 30 ON ERROR GOTO 30000
 40 DIM BT$(40)
 50 BELL$=CHR$(7)
 60 DIM KEY(12),BYTE$(40)
 70 DATA 136,137,139,138,72,68,65,73,80,82,87,86
 80 REM L-ARROW,R-ARROW,UP-ARROW,DN-ARROW,H,D,A,I,P,R,W,V
 90 FOR J=1 TO 12
100 READ X
```

```
 110 KEY(J)=X
 120 NEXT J
 130 *FX 4,1
 140 X=0: Y=0: PG=0: MD%=1
 150 DRIVE=0
 160 ?871=0: ?&72=&7B: ?&73=0: ?874=0: ?875=3: ?879=&21
 170 BA=PAGE
 180 DZAP=0
 190 CLS
 200 HD$=CHR$(141)+CHR$(130)+ "MEMZAP/DISCZAP PROGRAM"
 210 PRINT TAB(4,5);HD$
 220 PRINT SPC(4);HD$
 230 PRINT TAB(10,10);"Do you want:"
 240 PRINT TAB(10,13);"1) MEMZAP"
 250 PRINT TAB(10,15);"2) DISCZAP"
 260 PRINT TAB(10,18);"Type 1 or 2: ";
 270 INPUT ""DZAP
 280 IF DZAP<1 OR DZAP>2 THEN VDU 7: GOTO 260
 290 DZAP=DZAP-1
 300 CLS
 310 IF DZAP THEN BA=&7B00
 320 PRINT TAB(0,21);CHR$(130);"Move cursor:" ;CHR$(135);
     "[]ˆv(+SHIFT) ";: IF DZAP=0 THEN PRINT CHR$(130); "New
     Page:";CHR$(135);"P" ELSE PRINT
 330 PRINT CHR$(130);"Input:";CHR$(135);"I ";CHR$(131);
     "Input mode:";CHR$(135);"A,H,D ";CHR$(133);"Quit:";
     CHR$(135);"ESC"
 340 IF DZAP THEN PRINT CHR$(134);"Read:";CHR$(135);
     "R t,s  ";CHR$(134);"Write:";CHR$(135);"W t,s  ";
     CHR$(132);"Drive:";CHR$(135);"Vn"
 350 KEY=5: GOSUB4000
 360 IF DZAP THEN TRACK=0: SECTOR=0: GOSUB 6420
 370 IF DZAP=0 THEN PROC_printdisplay(BA)
 380 PROC_inverse(X,Y)
1000 REM *** HANDLE KEYS ***
1010 PRINT TAB(0,20);"COMMAND:";SPC(32);
1020 PRINT TAB(9,20);
1030 KEY=0
1040 *FX15,1
1050 REPEAT
1060 KEY$=GET$
1070 SHIFT=0
1080 FOR J=1 TO 9+3*DZAP
1090 IF KEY(J)=ASC(KEY$) AND (DZAP=0 OR J<>9) THEN KEY=J:
     PRINT KEY$; " ";
1100 IF INKEY(-1) THEN SHIFT=1
1110 NEXT J
1120 IF KEY=0 THEN VDU 7
```

```
1130 UNTIL KEY
1140 PRINT TAB(0,1);SPC(30): PRINT TAB(11,20);
1150 ON KEY GOSUB 2000,2000,3000,3000,4000,4000,4000,5000,
     6000,6400,6600,6800
1160 GOTO 1000
2000 REM *** L/R ARROW KEYS ***
2010 PROC_normal(X,Y)
2030 IF SHIFT THEN X=7*KEY-7 ELSE X=X+2*KEY-3
2040 IF X>7 THEN X=X-8: KEY=4: GOTO 3030
2050 IF X<0 THEN X=X+8: KEY=3: GOTO 3030
2060 PROC_inverse(X,Y)
2070 RETURN
3000 REM *** DN/UP ARROW KEYS ***
3010 PROC_normal(X,Y)
3020 IF SHIFT THEN GOTO 3500
3030 Y=Y+2*KEY-7
3040 IF Y<0 OR Y>15 THEN Y=Y-16*(2*KEY-7): GOTO 3500
3050 PROC_inverse(X,Y)
3060 RETURN
3500 REM *** CHANGE PAGE ***
3510 PG=PG+2*KEY-7
3520 IF FN_newsector THEN GOTO 6430
3530 PROC_printdisplay(BA+128*PG)
3540 PROC_inverse(X,Y)
3550 RETURN
4000 REM *** SET INPUT MODE ***
4010 MD%=KEY-4
4020 PRINT TAB(33,0);CHR$(131);
4030 IF MD%=1 THEN PRINT"HEX  ";
4040 IF MD%=2 THEN PRINT"DEC  ";
4050 IF MD%=3 THEN PRINT"ASCII";
4060 PRINT CHR$(135);
4070 RETURN
5000 REM *** INPUT DATA ***
5010 INPUT ""IP$
5020 CH=0: BT$=""
5030 FOR J=1 TO LEN(IP$)
5040 CH$=MID$(IP$,J,1)
5050 IF CH$<>" " THEN BT$=BT$+CH$: IF MD%<3 THEN GOTO 5070
5060 IF BT$>"" THEN CH=CH+1: BT$(CH)=BT$: BT$=""
5070 NEXT J
5080 IF BT$>"" THEN CH=CH+1: BT$(CH)=BT$
5090 FOR J=1 TO CH
5100 IF MD%=3 THEN BT=ASC(BT$(J))
5110 IF MD%=2 THEN BT=VAL(BT$(J))
5120 IF MD%=1 THEN BT=EVAL("&"+BT$(J))
5130 PROC_rewrite(BT,BA+128*PG,X,Y)
5140 BT$=CHR$(46): REM DOT
```

```
5150 IF BT MOD 128>31 THEN BT$=CHR$(BT)
5160 PRINT TAB(32+X,Y+3);BT$;
5170 KEY=2: GOSUB 2000
5180 NEXT J
5190 RETURN
6000 REM *** NEW PAGE ***
6010 IF DZAP THEN VDU 7: RETURN
6020 IF MD%>2 THEN PRINT BELL$;CHR$(129);"WRONG MODE";:
     TIME=0: REPEAT: UNTIL TIME=500: RETURN
6030 INPUT ""IP$
6040 PROC_normal(X,Y)
6050 IF MD%=2 THEN BA=VAL(IP$)*256 ELSE BA=EVAL("&"+IP$)*256
6055 IF BA>&FFFF THEN PRINT BELL$;TAB(11,20);: GOTO 6020
6060 PG=0: X=0: Y=0
6070 PROC_printdisplay(BA+128*PG)
6080 PROC_inverse(X,Y)
6090 RETURN
6400 REM *** READ SECTOR ***
6410 INPUT ""TRACK,SECTOR
6420 PROC_normal(X,Y)
6430 IF PG<0 OR PG>1 THEN PG=PG-2*SGN(PG) ELSE PG=0: X=0: Y=0
6440 IF FN_validate(TRACK,SECTOR) THEN PROC_inverse(X,Y):
     RETURN
6450 ?&70=DRIVE: ?&76=&53: ?&77=TRACK: ?&78=SECTOR: ?&79=&21
6460 TRIES=0
6470 REPEAT
6480 TRIES=TRIES+1
6490 A%=&7F: X%=&70: Y%=0
6500 CALL &FFF1
6510 UNTIL ?&7A=0 OR TRIES=10
6520 IF ?&7A>0 THEN PROC_discerror: RETURN
6530 PROC_printsecno
6540 PROC_printdisplay(BA+128*PG)
6550 PROC_inverse(X,Y)
6560 RETURN
6600 REM *** WRITE SECTOR ***
6610 INPUT ""TRACK,SECTOR
6620 PRINT TAB(17,20);BELL$;CHR$(129); "ARE YOU SURE
     (Y/N)?";CHR$(135);
6630 YN$=GET$
6640 IF YN$<>"Y" AND YN$<>"N" THEN VDU 7: GOTO 6630
6650 IF YN$="N" THEN RETURN
6660 PRINT YN$;
6670 IF FN_validate(TRACK,SECTOR) THEN RETURN
6680 ?&70=DRIVE: ?&76=&48: ?&77=TRACK: ?&78=SECTOR: ?&79=&21
6690 TRIES=0
6700 REPEAT
6710 TRIES=TRIES+1
```

```
 6720 A%=&7F: X%=&70: Y%=0
 6730 CALL &FFF1
 6740 UNTIL ?&?A=0 OR TRIES=10
 6750 IF ?&7A>0 THEN PROC_discerror: RETURN
 6760 PROC_printsecno
 6770 RETURN
 6800 REM *** CHANGE DRIVE ***
 6810 INPUT ""DRIVE
 6820 PRINT TAB(18,0);CHR$(132);"DRIVE ";DRIVE
 6830 $&AE0="DRIVE "+STR$(DRIVE)
 6840 X%=&E0: Y%=&0A
 6850 CALL &FFF7: REM CALL CLI
 6860 RETURN
 7000 DEF PROC_rewrite(BT,BASEADD,X,Y)
 7010 BASEADD?(X+8*Y)=BT
 7020 PRINT TAB(3*X+7,Y+3);
 7030 IF BT<16 THEN PRINT ;0;
 7040 PRINT ;~BT;
 7050 ENDPROC
 8000 DEF PROC_printdisplay(BASEADD)
 8010 LOCAL HORIZ,VERT,BT,BT$
 8020 FOR VERT=0 TO 15
 8030 PRINT TAB(0,VERT+3);
 8040 IF BASEADD<&1000 THEN PRINT ;0;
 8045 IF BASEADD<&100 THEN PRINT ;0;: IF BASEADD+8*VERT<&10
      THEN PRINT ;0;
 8050 PRINT ;~BASEADD+8*VERT;
 8055 IF DZAP THEN PRINT TAB(0,VERT+3);"  ";
 8060 FOR HORIZ=0 TO 7
 8070 PRINT TAB(3*HORIZ+7,VERT+3);
 8080 BT=BASEADD?(8*VERT+HORIZ)
 8090 IF BT<16 THEN PRINT ;0;
 8100 PRINT ;~BT;
 8110 PRINT TAB(HORIZ+32,VERT+3);
 8120 BT$=CHR$(46)
 8130 IF BT>127 AND BT<255 THEN BT=BT-128
 8140 IF (BT>31 AND BT<127) OR BT=255 THEN BT$=CHR$(BT)
 8150 PRINT BT$;
 8160 NEXT HORIZ
 8170 NEXT VERT
 8180 ENDPROC
 9000 DEF PROC_inverse(X,Y)
 9010 PRINT TAB(3*X+6,Y+3);CHR$(130);
 9020 PRINT TAB(3*X+9,Y+3);CHR$(135);
 9030 ENDPROC
10000 DEF PROC_normal(X,Y)
10010 PRINT TAB(3*X+6,Y+3);CHR$(135);
10020 ENDPROC
```

```
11000 DEF PR0C_discerror
11010 PRINT BELL$;TAB(0,1) ;CHR$(136);CHR$(133); "DISC ERROR
      NO.";?&7A
11020 ENDPROC
12000 DEF PR0C_printsecno
12010 PRINT TAB(0,0);CHR$(134);"TRK ";TRACK;", SECT ";
      SECTOR;"    "
12020 ENDPROC
13000 DEF FN_validate(TRACK,SECTOR)
13010 LOCAL FAIL%
13020 FAIL%=0
13030 IF TRACK<0 OR TRACK>79 OR SECTOR<0 OR SECTOR>9 THEN
      VDU 7: FAIL%=1
13040 =FAIL%
14000 DEF FN_newsector
14010 LOCAL FAIL%
14020 IF DZAP=0 OR (PG>=0 AND PG<2) THEN =FALSE: RETURN ELSE
      FAIL%=1
14030 SECTOR=SECTOR+SGN(PG)
14040 IF SECTOR<0 OR SECTOR>9 THEN TRACK=TRACK+SGN(SECTOR):
      SECTOR=SECTOR-10*SGN(SECTOR)
14050 =FAIL%
30000 *FX 4,0
30010 PRINT TAB(0,23)
30020 END
```

The key lines are 7010, where a modified byte is rewritten to memory, and 8080 where a byte at a time is read from within a double loop over VERT and HORIZ.

The program’s output is used as follows: 128 bytes of memory are displayed in the main portion of the screen, in rows of 8 bytes. On the left is the address of the first byte in the row, then the 8 bytes are listed in hexadecimal and finally, where possible, the ASCII character corresponding to each byte is listed. Where there is no printable ASCII character, a dot is shown, and note that the high bit is ignored for this purpose.

This last listing is in many ways the most useful. It enables you to pick out the contents of data files; find your way through BASIC programs; and to determine which parts of memory directly represent meaningful combinations of characters.

Near the bottom is the command line, and below that a brief reminder of the active keys and their uses.

One of the bytes is highlighted in green (on a monochrome screen this will appear as a paler shade), and the highlight can be moved around the screen using the four cursor keys. If the highlight moves off the top or bottom of the screen, a new ‘page’ of 128 bytes is displayed. The highlight can be moved by a complete page by holding down the SHIFT key while pressing the up or down arrow keys, and to the left or right edge by SHIFT and left or right arrow.

Larger movements can be made by the letter P (for Page), followed by the new page number wanted. For example, to read the memory starting from address &E00, type

```
PE <RETURN>
```

The highlighted byte can be altered by typing I (for Input), followed by the new contents of one or more bytes. Input can be in one of three modes: hexadecimal, decimal or ASCII. The current input mode is displayed at the top right of the screen, and the chosen mode is selected by typing H, D or A. Multiple bytes, starting from the one highlighted, can be input in hexadecimal or decimal by separating the bytes by a single space (not a comma, since the INPUT command is used to collect the string of bytes). In the ASCII mode, the ASCII codes of single characters are written to each byte, and spaces between bytes are not necessary. In the case of hexadecimal and decimal modes, the mode also controls the way in which the P command works (ASCII is clearly inappropriate in this context, and is not allowed). The program is terminated by pressing <ESCAPE>.

If you want to use this program to examine another BASIC program, this can be accomplished quite easily by setting PAGE to a different value before CHAINing ZAP. Any value of PAGE up to about &6800 is possible, since ZAP operates in Mode 7.

Use of the program ZAP for disc editing is described in Appendix M.

Exercise 10.9

Use the MEMZAP facility of ZAP to examine the various areas of memory described in Section 10.2.

10.6 THE WAY BASIC PROGRAMS ARE STORED

With an understanding of how BASIC programs are stored in memory, and the MEMZAP program, you can do a variety of things not otherwise possible. For instance, it is possible in some cases to recover from ‘Bad program’ situations, recover the existing portion of partly overwritten programs, perform some special tricks and so on. It is not possible in this book to go into great detail about all these features, but we can give the essential background information to enable the adventurous programmer to explore these possibilities.

Program lines are stored as a mixture of ASCII codes and ‘tokens’. All BASIC keywords are stored in a special form, as tokens. These are simply single byte codes, distinguishable from ASCII codes because they are greater than &80. MEMZAP will show them in the ASCII display as dots. Line numbers following GOTO statements are stored in a special format, starting with &8D (this format is explained in the book *The BBC Revealed* by Jeremy Ruston, published by Interface). The remainder of a line, except for the initial line number − formulae, strings, variable names and so on − is stored as ASCII codes (this includes all spaces in the line, including any inadvertently added to the end of a line, as can easily happen when screen editing). Each line ends with &0D, the <RETURN> code.

Each line starts with two items of ‘housekeeping’ information. First is the line number, stored as a two-byte hexadecimal number, high byte first (in contrast to the way in which addresses are normally stored, which is low byte first). Second comes a single byte containing, in hexadecimal, the length of the line, or more accurately, an offset from the start of one line to the start of the next.

The end of a program is marked by &FF in the high byte position of the line number, which would be the first byte of that line. (This shows up distinctively with the MEMZAP program, as a solid white rectangle in the ASCII listing.) When you type NEW to delete a program, all that happens is that &FF is written over the high byte of the first line number (this will be at address PAGE+1 − programs start with a <RETURN> character &0D at PAGE). Thus to restore a program erased by NEW, all that is necessary is to replace the &FF marker with the proper high byte of the first line number − probably 0 − and then type END in immediate mode to get the BASIC interpreter to sort out other internal pointers. This is, of course, what is done automatically by the OLD command, and it is why there is an apparent bug if you use it with a program starting with a line number greater than 255 − the high byte of the line number has been overwritten and is reset to zero by OLD.

Of course, there is no point in recovering a program erased by NEW with MEMZAP when you can do it more simply by the OLD command; but if you accidentally enter a program line, however trivial, before typing OLD, recovery is no longer possible. With MEMZAP, however, you can delete the end of program marker, sort out the pointer to the first line not overwritten, and thereby recover the remainder of the program.

Exercise 10.10

Type in a short program such as

```
10 SUM=0
20 FOR J=1 TO 10
30 SUM=SUM+J
40 NEXT J
```

Change the value of PAGE, run MEMZAP and examine how the program has been stored in memory. Restore PAGE to normal, delete the program with NEW, switch back to MEMZAP and recover the program by replacing the end of program marker with a zero. Return to normal PAGE once more, type END, and check by LISTing and RUNning that you have recovered the program. You could also see whether you can sort out the problem of recovering the program after typing in a new line after NEW (type in − say − 10 REM).

10.6.1 Protecting a program against LISTing

Another example of the useful jobs that can be done with MEMZAP is protecting a program against LISTing. The essence of the technique is to bury the special character 21 (&15) at the end of a REM, or on its own after a colon. This character will switch off the screen display when listing reaches that point. It cannot be inserted in a REM as CTRL-U, because in direct mode it serves a different purpose, as the command to delete the current line. However, if you put in a dummy character after a REM, you can use MEMZAP to change the character to &15.

If you do this and then try and LIST the program, you will see that the remainder of the program is indeed invisible, but the protection mechanism is less than perfect, because it is quite obvious to any inquisitive user where the program stops listing, and the offending line can be deleted or the REM edited out.

The complete solution is to precede the code &15 with enough delete characters (code &7F) to backspace over the whole line. Again these will have to be entered with MEMZAP over a dummy line such as REM123456789, with the &15 in last place. To complete the process, you should switch display back on again after listing is complete with code 6 in a final REM.

Regretfully this protection mechanism is far from foolproof, because a user can just LIST that part of your program beyond the REM. A better ploy for real protection is to conceal the fact that you have ever used the technique. Identify some key lines in your program, insert a code &15 before and code 6 after, and renumber the lines so that they are packed together between a pair of lines in the normal sequence of 10s. This type of insertion will be almost impossible to track down (without the user having a MEMZAP program of his own).

10.6.2 Variable storage

There are two factors to be considered regarding the storage of variables: the way in which a single variable is represented, and the way in which the table of variables is organized.

Real and integer variables are stored in the same format as in data files (see Section 9.5.2) and strings are stored as ASCII codes, together with a byte giving the string length.

The storage of integers is quite simple. All integers are four bytes long, giving a range of about ±2147 million, and the bytes are stored in sequence, low byte first, following the normal 6502 convention. The high bit of the highest byte is set for negative numbers.

Real variables, on the other hand, are stored in a much more complex fashion. As with the display of numbers in exponent form or scientific notation, the numbers consist of a mantissa and exponent, and they are stored in five bytes, exponent first.

The mantissa for real numbers is a four byte number, stored high byte first (in contrast to integers where the low byte is stored first). The exponent is adjusted to be such that the most significant bit of the mantissa (the first bit to the right of the point) is always a one. Thus the ‘decimal’ point is to the left of the leftmost digit and the number is a binary fraction. The final complication is that because the first digit is always a one it is therefore redundant, and it is used instead to represent the sign of the number, being zero for a positive number and one for a negative number. The remainder of the mantissa simply gives the magnitude of the number; it is not a proper two’s complement negative number.

For internal storage the exponent is the power of two stored in a single byte in ‘excess 80’ form; that is, &8^0 represents 20 &8^1 represents 2^1 and &7E represents 2^{-2}. Another way of looking at this format is as a signed one-byte number with the sign bit reversed, or with an offset of 2^{-2}. Table 10.1 gives a few examples to help to clarify the format.

Table 10.1 The internal storage format for real numbers.

Decimal number	*Binary form*	*Internal storage*
+1	0.1×2^1	**`81 00 00 00 00`**
-1	-0.1×2^1	**`81 80 00 00 00`**
+15	0.1111×2^4	**`84 70 00 00 00`**
-0.1875	-0.11×2^{-2}	**`7E C0 00 00 00`**

Now let us consider how the table of variables is organized. The storage of variables comprises three parts: first is a two byte pointer which is the address of the next variable; next comes the variable name in ASCII form, including the % or $ for integer and string variables and followed by a null byte; and lastly the value of the variable itself as described above. The last variable in the list contains a null byte in place of the high byte of the pointer to the next variable. For example, if with no program present there are two direct mode variables, ALPHA1 and ALPHA2% having values 0.5 and 5, these would appear in memory starting at &1902 (assuming a disc-based system), as

```
0F 19     4C 50 48 41 31   00 80 00 00 00 00

pointer    L  P  H  A  1   null
                           byte

00 00     4C 50 48 41 32 25    00 05 00 00 00

pointer    L  P  H  A  2  %    null
                               byte
```

In the case of strings a complication arises because the string may change in length, possibly becoming longer so that it does not fit into the space initially allocated for it. What happens then is that the original string and its storage space is discarded, and the new string is added to the end of the variable space. Because of this, two extra items of information are stored immediately after the string name. The first is a two-byte pointer to the location af the actual string, and the second is a single byte containing the number af bytes allocated to the string. The actual length of the string, mentioned earlier, is also stored in a single byte after these other two items, so initially it will be at the start of the string proper, but after reassignment it may be separated. The reason for storing the number of bytes allocated is so that, on reassignment of the string variable, the BASIC interpreter can decide whether the new string will fit into the existing space, or whether that space must be discarded and the string appended to the end of the variable space.

In order to speed up access to a variable, they are not all stored in a single list. instead, a whole series of linked lists is used, one for each letter of the alphabet, so that all variables starting with the same initial letter form a single chain, and the two-byte pointer at the beginning is the address of the next variable with the same initial. The initial letter therefore becomes redundant, and is omitted. Only subsequent characters of the variable name are stored. The first pointer in the chain is stored on Page 4 at address (&400+2*ASC(<letter>)), following the resident integer variables.

The organization of arrays is even more complicated. When an array is DIMensioned, enough storage space is set aside for the whole array, including obviously the appropriate amount of space for each array element (or in the case of string arrays, the array of pointers). For example a real array dimensioned as DIM A(5,10) would have 66 elements needing 66×5 or 330 bytes. In addition, immediately after the array name (which includes the opening bracket to identify it as an array) the offset to the start of the storage space proper is stored, and then the dimension of each index of the array, two bytes each stored in the usual lowlhigh format. (The offset is thus a double byte having a value of twice the number of dimensions plus 1.)

As an example, a real array declared by DIM ARR(5,10) would appear in memory as

`00 00`	**`52 52 28`**	**`00`**	**`05`**	**`06 00`**	**`0B 00`**	**`81 00 00 00 00`**	...
pointer	**`R  R  (`**	null byte	offset dim.	first dim.	second	first value	...

Here the offset is 5 (twice the number of dimensions − 2 in this case − plus 1) and the dimensions are 6 and 11 respectively (remember that array indexes start from zero).

In addition to variables, addresses of procedures and functions are also stored in the space above a program, in a further pair of linked lists. The entry for each routine, whether procedure or function, simply contains the pointer to the next item in the list, the routine name (including the initial letter) and the address of the point in the program where the routine is to be found. Just as with variables, the entry in the list is made on the first occasion that the routine is accessed when the program is executed. The start of the chain of pointers for procedures is at &4F6, and for functions is at &4F8. Readers wanting to gain a thorough familiarity with variable storage will find it useful to use the MEMZAP program to study some actual examples that can be set up by making assignments in direct mode. (This will entail setting PAGE to a different value − say &6000 − to set up some variables, and switching back to the value of PAGE where ZAP is stored to use the program again.)

From this description of the method of variable storage we can note three points which are relevant to the efficient execution of a BASIC program

1. Use different initial letters as far as possible, since this keeps the chains of variables short and enables the interpreter to find the required variable more quickly.
2. If the string stored in a string variable is increased in length, the initial storage space allocated is insufficient. That space is therefore discarded and the new string added to the end of the variable space. This can eventually waste a great deal of memory in certain circumstances. If strings may grow in this way, they should be ‘initialized’ first, at their maximum eventual length, by setting them equal to dummy strings. This is particularly important in situations such as sorting, where array elements are likely to be repeatedly reassigned.
3. Procedure and function names are stored along with other variable names. This enables the interpreter to find the appropriate procedure or function very quickly, without searching right through the program line by line, as has to be done for every GOTO or GOSUB. For maximum speed of execution it is therefore advisable to use procedures and functions wherever possible.

10.6.3 The resident integer variables

The integer variables @% (the PRINT format variable) and A % to Z% have special properties and purposes. Instead of being stored in the normal variable space above a program, they are located, four bytes each, from addresses &400 to &46B, as was mentioned earlier. This gives these particular variables two unique features.

First, because their positions in memory are fixed, they can be accessed extremely quickly, and these variables should be used preferentially wherever speed of operation is critical. Second, and much more important, because they are not stored in the usual variable space but in the reserved section of memory below the OSHWM, their values are not cleared by any of the Operations, such as loading a new program, that clears all other variables and their values (even <BREAK> does not destroy them). This makes the variables ideal for carrying values from one program to another.

Certain of the resident integer variables also have a third property. Their current values are used to initialize certain 6502 registers whenever machine code programs are executed with the CALL or USR commands. These variables, and their purposes, are

A% − transferred to the Accumulator
C% − transferred to the Carry flag
P% − transferred to the Program counter (this is used with the assembler)
X% − transferred to the X register
Y% − transferred to the Y register

(When the values of the variables are greater than 255, it is the low byte of the value that is transferred. In the case of C%, only the lowest bit is transferred.)

Note that you need not be inhibited from using A% etc. in normal BASIC programs, providing that you are not using CALL or USR.

Exercise 10.11

In direct mode, set variables A%=12, A1%=34 (or any other values that you choose). Press <BREAK>, then try and PRINT the values of A% and A1%.

10.7 HANDLING THE FILING AND OPERATING SYSTEMS FROM WITHIN PROGRAMS

10.7.1 Issuing operating system commands from within programs

In Example 9.9 we saw a case where we would have liked to issue an aperating system command with parameters not known in advance. In that particular example, we wanted to issue a *SPOOL command with a filename supplied by the user, but other situations can occur; for instance you might want to issue a *SAVE command with addresses calculated from within the program, or issue a *FX command such as *FX 138 with a character determined by the program.

It has already been explained that all commands starting with a star are mt handled by the BASIC interpreter but are passed straight to the operating system where they are handled by the Command Line Interpreter (CLI). This is why the problem occurs, because variables or expressions that we might want to use as parameters to the commands can only be handled by the BASIC interpreter (it is also why no further BASIC commands can be included on the line).

The solution to the problem is to pass your command directly to the CLI, instead of letting the leading star force the BASIC interpreter to do this. There are two ways to send commands straight to the CLI, but the easier one is only available with BASIC II, in the form of a new command, OSCLI. With OSCLI you simply form your OS command without the leading star as a string expression following the OSCLI command.

Under BASIC I the OSCLI command is not available. It is still possible to make a direct call to the CLI but you must first load the desired command directly into memory as a string (again without the leading star), then set the resident integer variables X% and Y% to point to the address of the string, and finally CALL the CLI at address &FFF7.

In both cases the fundamental requirement is thus to form the command into a string, and for BASIC I the string can be loaded into memory using the $ indirection operator, as explained in Section 10.4. The exercise below shows how direct access to the CLI is achieved for both versions of BASIC for the particular problem encountered in Example 9.9.

Exercise 10.12

Make the following modifications to the program from Example 9.9 to allow the user to specify the name of the filename for the spooled output.

```
 100 PROC_spool
 220 END
1000 DEF PROC_spool
1010 LOCAL FILENAME$
1020 PRINT "Type filename for the spooled output:": INPUT
     ""FILENAME$
1030 CMD$="SPOOL "+FILENAME$
1040 OSCLI CMD$
1050 ENDPROC
```

or for BASIC I

```
1040 $&AE0=CMD$
1050 X%=&E0: Y%=&0A
1060 CALL &FFF7: REM CLI CALL ADDRESS
1070 ENDPROC
```

Line 1030 forms the required command as the string variable CMD$ (note the space after SPOOL). For BASIC II all that is then needed is line 1040 to issue the OSCLI command.

For BASIC I line 1040 uses the dollar indirection operator to place the string into memory starting at &AE0, which is a fairly safe area of memory. Line 1050 sets the variables X% and Y% to point at this address (low byte in X%), and line 1060 performs the actual CLI call.

You could write a similar procedure for *SAVE or any other command Where the parameters are not known in advance. Note that *SAVE requires addresses to be in hexadecimal, whereas you would normally turn a number into a decimal string with STR$. However, you can get a hexadecimal string form by preceding the brackets after STR$ by a tilde, so you might form the string by a line such as

```
CMD$="SAVE "+F$+" " +STR$˜(START)+" +" +STR$˜(LGTH)
```

10.7.2 Using commands in the function keys from within a program

There are certain occasions when you may wish to issue commands from within a program which either are not allowed or would be overwritten before completion. For instance, you might wish to load a new program in two sections using the method of merging described in the *User Guide*, and then run the resultant program. You could not do this in the normal way from within a program, because the LOAD command may not be issued from within a program and because, even if you could get round this problem, one of the program sections might overwrite the old program before it could finish the series of instructions.

The solution is to load the necessary series of instructions into one of the function keys, where they are safe from overwriting, and from where they will be issued as direct mode instructions. You could then have the program tell the user to press the appropriate function key to cause the instructions to be issued, but even this is not necessary. The command *FX 138 allows you to load characters directly into the keyboard buffer, which would have the same effect as if the user pressed the key.

As an example, consider how we would carry out the task mentioned above. The original program could program a function key with the *KEY command, then load that function key into the keyboard buffer, and finally END, at which point the keyboard buffer would be polled and the function key found there and executed.

As a sophistication, you could also disable the screen with VDU 21 before this operation, to suppress the potentially confusing display on the screen of the commands being issued from the function key.

The process of merging requires you to LOAD the first program section, then *LOAD the second section at address (TOP-2) for the first section, and finally sort out the internal pointers with a command such as OLD. It would be simplest to work out the *LOAD address for the second section beforehand by issuing the command

```
PRINT ˜(TOP-2)
```

with the first section loaded. The resultant value can then be used with the *LOAD command. Obviously the second section of program must have line numbers after those of the first section.

If we suppose that the two program sections to be loaded are called PARTI and PART2, and the load address for the second is &3456, the following lines in the original program would achieve the desired effect.

```
2000 VDU 21: REM DISABLE SCREEN DISPLAY
2010 *KEY 0 LOAD"PART1"|M *LOAD PART2 3456|M OLD |M RUN |F|M
2020 *FX 138,0,128
2030 END
```

The parameter 128 to *FX 138 is the code for function key f0. Subsequent keys go up in sequence, to 137 for f9. The |F switches the screen display back on after issuing the commands.

Another occasion where it is necessary to load commands into a function key is to achieve true chaining of programs.

10.7.3 The CHAIN command

There are occasions when it may be desirable, or necessary, to *chain* one program from another. Chaining here means running one program from another, with the variables from the first being retained for use by the second program.

The chief use of this is where a program is too big to fit into memory complete. It must then be broken up into modules, and each module may need to use variables set up by an earlier module. In some instances the modules will form a linear chain, control passing from one to the next until the program is ended. In other cases, where the path of execution through the program may vary, the chain may be haphazard, or even circular if the program forms a giant loop.

The command CHAIN on the BBC computer is actually a misnomer, because it does not provide a true chaining facility. When a new program is CHAINed from a previous one, all variables except the resident integer variables are cleared at the start of the new program, whereas chaining normally implies one program being chained to the end of a previous one, with variable values being preserved during the process.

There are at least three ways in which true chaining can be effected. The least satisfactory is to write all the variables that need to be preserved to a data file before passing execution to the next module, and then reading the values back.

The best method, where it can be achieved, is to use the resident integer variables. These were described in Section 10.6, and an important use of them is in preserving the value of variables when chaining programs. If you also need to preserve strings, these could be stored in a safe area of memory (below PAGE or above HIMEM) with the dollar indirection operator ($), and restored by the next program module. The relevant addresses could be transferred by means of the resident integer variables.

10.7.4 Preserving real variables during chaining

If you need to preserve real variables, or a large number of integer variables or arrays, only the method of storing to a data file on disc would work. There is, however, a third method by which variables can be preserved when CHAINing a new program module, but it is ‘unofficial’, and great care must be taken since the normal protection mechanisms and checks of the BASIC interpreter must be overridden.

All that happens when you CHAIN a new program is that the computer carries out an automatic CLEAR so that the pointers to the variable chains, stored in the upper half of Page 4 of memory, are destroyed and the pointer to the end of the variable space is reset. Provided that the new program is not larger than the old one, or an existing program is not enlarged by editing, the variables themselves remain in memory undisturbed.

The simplest method of preserving access to these variables is not to CHAIN the new program, but to *LOAD it and then use GOTO to jump into the program without CLEARing variables first. The procedure to adopt is as follows

1. To prevent a longer program module overwriting some of the variables stored at the end of a shorter module, first load the largest of the modules, and PRINT the value of LOMEM. The first module in the chain must start with a command setting LOMEM to this value (or a little larger still might be advisable).
2. Each module must chain in the next by loading one of the function keys with a *LOAD for the next module, followed by a GOTO to the first program line (it would be possible to jump to a later line if you wished, say to skip a DIM statement if the arrays already existed). Note that *LOAD must be used rather than LOAD because the latter, like CHAIN, clears all variables.

Example 10.4

The following pair of programs show a simple example of chaining in operation, with a real variable and a succession of array elements being carried from one program to the next.

Program P1

```
10 LOMEM=10000
20 DIM RV(50),ST$(50)
30 J=0
40 VDU 21: REM DISABLE SCREEN DISPLAY
50 *KEY 0 *LOAD P2|M GOTO 10 |M
60 *FX 138,0,128
```

Program P2

```
 10 REM THIS LINE must EXIST
 20 VDU 6: REM REACTIVATE SCREEN DISPLAY
 30 J=J+1
 40 IF J>1 THEN PRINT "NEW VARIABLE IS ";XYZ
 50 IF J>50 THEN END
 60 INPUT "TYPE A NUMBER",RV(J)
 70 INPUT "TYPE A STRING",ST$(J)
 80 FOR I=1 TO J
 90 PRINT ST$(I),RV(I)
100 NEXT I
110 PRINT: INPUT "NEW VARIABLE",XYZ
120 VDU 21
130 *KEY 0 *LOAD P2 |M GOTO 10 |M
140 *FX 138,0,128
```

10.8 USING MACHINE CODE PROGRAMS

It is not the purpose of this book to teach machine code programming. However, it can be useful for the non-machine code programmer to have some understanding of what machine code is, and how he can use it without needing to understand it in any detail. For instance, program listings that you wish to copy may include a section of machine code. Nevertheless, some readers may not be interested in machine code at all, and this section may safely be skipped if you wish, apart perhaps for Section 10.8.5.

Machine code is a very primitive language, but it is understood directly by the 6502 microprocessor, whereas BASIC has to be decoded into machine code by the BASIC interpreter which is resident in ROM. This makes BASIC extremely slow compared with machine code for simple operations − typically machine code may operate 100 times faster than BASIC. For some applications only machine code can give the speed necessary. Arcade games are an obvious example, though by no means the only one. (An example of a more serious application where machine code is used is in routines to produce copies of graphics screen displays on a printer, of which several have been published.)

One other instance where machine code has to be used, is to carry out operations which simply cannot be done in BASIC because of the limitations of the BASIC language or interpreter.

True machine code is a series of one byte numbers that are the operating instructions for the 6502 microprocessor. Many of these are followed by oneor two-byte addresses giving the location where data is to be fetched from, or written to. Since these ‘op codes’ are exceedingly difficult to remember, programmers invariably use mnemonic code, with three letter mnemonics for the op codes. An assembler is used to translate these, in essence by looking up the corresponding numbers from a table. (An assembler also has a number of sophistications to make life a little easier for the programmer, in particular by the provision of labels.)

The operations that are possible are very simple, mainly reading from and writing to memory, adding or subtracting bytes, unconditional jumps or conditional branches, and a variety of operations on individual bits of a byte.

A machine code program is executed by specifying the memory address of the first instruction. This can be done from BASIC, by the command CALL followed by the starting address. The operation codes (with their accompanying data addresses where appropriate) are then executed in order, unless a jump or branch instruction is encountered.

10.8.1 The BBC assembler

One of the many unusual features of the BBC microcomputer is the built-in assembler that works directly from BASIC. In particular, the assembler can handle BASIC variables at any point in the assembly code, substituting into the machine code the value of the variables at the time of assembly.

The assembler is invoked, either in direct mode or more commonly from a program, by an opening square bracket − [(note that in Mode 7, square brackets are printed as left and right arrows). In direct mode, each assembler line must start with a square bracket, but in a program assembly continues until a closing square bracket is encountered (this must be at the start of a line). Within the square brackets, the usual mnemonic forms are used, together with labels which are denoted by a preceding full stop. Both labels *and* BASIC variables can also be used for addresses.

Very few of the usual pseudo-op codes are provided, because there are other ways of achieving the same operations.

The resident integer variable P% is used to set the program counter from BASIC before entering the assembler, thus defining the origin for the assembly. Most assemblers do this with an ORIGIN pseudo-operation. Similarly, in place of a DATA or DFB pseudo-op, it is possible at any time to leave the assembler temporarily and enter data into memory with the indirection operators. P% can again be used to give the position in memory, provided that it is incremented for every byte entered. Labels can be defined in the same way as ordinary BASIC variables, removing the need for an EQU or = pseudo-operation. (In BASIC II only, there is a set of pseudo-ops EQUB, EQUW, EQUD, EQUS to enter respectively, 1, 2 or 4 bytes of data or a string into memory. These are equivalent to the usual DFB rather than EQU.)

Comments can be included in a statement at the end, if they are preceded by a backslash.

Example 10.5

Enter the following short program

```
10 start%=&C80
20 P%=start%
30 [
40 OPT 1
50 LDA #5
60 ADC #7
70 RTS
80 ]
```

Line 20 sets the start of the assembly to &C80, which is a safe position unless you have a large set of user defined characters. P% could have been used in place of start% in line 10, eliminating line 20, but this form has been used in order to be consistent with a later version of the program. RUN the program and look at the display. OPT is not a true mnemonic but a ‘directive’ or pseudo-operation telling the assembler what listing OPTion to use. OPT 1 lists everything as the program is assembled for the first time. Working from left to right the listing shows the address, the byte(s) forming the instruction, the mnemonic notation for that instruction and lastly any comments. Instructions need from one to three bytes. It is beyond the scope of this book to describe in detail the instructions executed by the processor − those used in this program are

LoaD Accumulator with the number 5
ADd with Carry the number 7
ReTurn from Subroutine

Note the use of P% to determine the position in memory at which the machine code will be assembled. As was pointed out in the previous section, certain of the resident integer variables have special applications for assembler programs. A%, C%, X% and Y% are copied to the appropriate registers when the machine code routine is called or used. P% is used during the assembly process to set the program counter to the desired position. Note that at the end of the routine the registers are not returned to the resident integer variables, but they are accessible through the USR function, as explained below.

10.8.2 CALL and USR()

There are two ways of running user-defined machine code routines − CALL <address> and USR(<address>), where <address> may be either a number such as &1234 or a variable. CALL updates the 6502 from the relevant resident variables and then jumps to the user routine. The RTS instruction returns to the BASIC system. The changed values of the registers are lost. CALL is the machine code equivalent to PROC which simply does something without returning values to the program.

USR calls up a machine code subroutine in a similar way to CALL, but it is a function rather than a command and it also returns the values of the processor status register, accumulator, and X and Y index registers, as a single 32-bit number.

Exercise 10.13

Run the machine code program created by Example 10.5 by typing

```
PRINT ~USR(start%)
```

(Note that Example 10.5 did not run the machine code program. It simply created it.)

You should see an eight character (four-byte) number. The left-hand byte represents the status register which stores ‘flags’ such as carry, zero and overflow; the next two bytes are the X and Y index registers and the last (right-hand) byte represents the accumulator. This should have the value 0C Or 0D depending on whether the carry bit was set (ADC is ADd with Carry). Type C%=1 first then PRINT USR(start%) again. The accumulator should definitely be 0D this time. Set X% and Y% to different numbers in immediate mode and print USR again and notice which byte represents which index register. To prove that A% is left unchanged set A% (the accumulator variable) to an arbitrary value and print A% after using USR again.

Exercise 10.14

It is inconvenient to use numbers in CALL and USR statements and it is not necessary when labels have been included in the mnemonic listing. Insert

```
 35 .test
```

in the program from Example 10.5 and RUN it again, then try PRINT ~USR(test). The full stop in front of ‘test’ tells the assembler that a label rather than a mnemonic follows − the dot is not part of the label name. Hence when the label is used as an operand it is written just as ‘test’, with no full stop in front.

Exercise 10.15

Type NEW and enter the following

```
 10 start%=&C80
 20 P%=start%
 30 [ OPT 1
 40 .test
 50 LDA #7
 60 .loop
 70 ADC #5
 80 JMP loop
 90 RTS
100 ]
110 END
```

RUN the program and look at the code generated for the JMP (JuMP) instruction − ‘loop’ has been replaced by the actual address of the ADC instruction. (Do *not* try and CALL the machine code program − it is a closed loop and will run until interrupted.)

Interchange lines 60 and 80 so that the jump instruction points ‘forward’, and run the program to assemble again. Change OPT 1 to OPT 3 and note the error message. When the assembler encounters JMP loop now it does not know what to do as ‘loop’ has not yet been defined. We must force the assembler to work through (pass) the code twice − once to discover all the labels and then again to assemble properly.

We can make two passes of the assembler by using a FOR...NEXT loop around the code. We are not interested in error messages on the first pass as we will expect problems with labels so we need to use different OPTions for each pass, which is achieved by using a variable as the parameter for OPT. The OPTion number is actually a two bit number and so has the value 0 to 3. The least significant bit enables (1) or disables (0) listing and the most significant bit should be cleared or set according to whether it is the first or second pass to suppress some of the error messages. We will use OPT 0 (no listing) on the first pass and OPT 3 (listing and error messages) on the second pass.

Exercise 10.16

Modify the program from Exercise 10.15 by adding extra lines 15 and 105. The resultant program will be as follows

```
 10 start%=&C80
 15 FOR Z%=0 TO 3 STEP 3
 20 P%=start%
 30 [ OPT Z%
 40 .test
 50 LDA #7
 60 JMP loop
 70 ADC #5
 80 .loop
 90 RTS
100 ]
105 NEXT Z%
110 END
```

Run the program and observe that the listing (which comes from the second pass) now has the forward address for the JMP worked out correctly.

Note that there is nothing special about Z%; any variable would do. Again do not attempt to execute the resultant machine code program. It is a meaningless program simply designed to explore the problems of forward jumps.

Exercise 10.17

Why has P%=start% been included inside the FOR...NEXT loop? Test your answer by moving it to line 12 after modifying line 15 to select OPTions 1 and 3 to list on both passes.

10.8.3 Allocating space for a machine code program

For machine code programs needing larger amounts of space than can be safely found in areas such as &C80, it is possible to set aside memory in the variable storage area for the machine code.

```
DIM start% 300
```

would allocate 300 bytes of memory for a routine, the start of which is stored in the variable start%. In the above programs, line 10 could be replaced by

```
10 DIM start% 20
```

Note that this use of DIM is at first sight quite different from the usual form, Where a variable name is followed by a value in brackets. In fact the use is not SO illogical: when an array is defined, enough memory is set aside for all the elements of the array, so in both cases, DIM has the effect of reserving memory in the variable storage area for later use.

10.8.4 System access

It would be difficult, from within machine code, to carry out such tasks as printing a character to the screen, and in cases such as this it is possible to access the relevant BASIC interpreter routine from within the machine code program. Particularly easy routines are the *FX commands and the VDU driver routine. *FX commands are invoked by JSR OSBYTE (JSR means Jump to SubRoutine). OSBYTE stands for Operating System BYTE and is at address &FFF4. (You cannot use the word OSBYTE directly in assembly unless you have defined it earlier in the BASIC part of the program using a LET command.)

The particular *FX is selected by the value in the accumulator, X register and Y register. The accumulator specifies which *FX call is to be made and X and Y supply the remaining two values sometimes required. X and Y should be set to zero when no other values are needed so

```
LDA #0
LDX #0
LDY #0
JSR &FFF4
```

correctly assembled and CALLed is equivalent to *FX 0 (print the version number).

Example 10.6

The following program will perform this call.

```
10 DIM P% 10
20 [OPT 1
30 .fx LDA #0
40 TAX
50 TAY
60 JSR &FFF4
70 ]
80 CALL fx
```

(TAX and TAY mean: Transfer (copy) Accumulator to X or Y .) RUNning the above program should print the operating system number.

VDU calls are carried out by JSR OSWRCH, which is at address &FFEE. Where more than one parameter is needed (as is usual with VDU statements) the JSR is simply used repeatedly after loading the accumulator with successive values. (BASIC programs also carry out writing operations by calling OSWRCH.)

Other OS calls that you may wish to use are: OSRDCH (address &FFE0) which reads a character from the currently selected input stream; OSNEWL (address &FFE7) which writes a newline; and OSCLI (address &FFF7) which has been explained in Section 10.7 .1. Other more specialized OS calls are OSFIND (&FFCE), OSGBPB (&FFD1), OSBPUT (&FFD4), OSBGET (&FFD7), OSARGS (&FFDA), OSFILE (&FFDD) and OSASCI (&FFE3). These calls are all documented in the BBC *User Guide*.

One simple use of machine code is in fact to access *FX calls which return parameters. For instance, OSBYTE call &87 (equivalent to *FX 135) returns the character at the text cursor position in the X register, and also the current mode number in the Y register. Thus if you want to ascertain the current mode, for instance to save having to request the mode from the program in a procedure such as PROC inverse from Chapter 11, the following section of program could be used

```
100 A%=135
110 MD=USR(&FFF4)
120 MD=(MD AND &FFFFFF) DIV &10000
```

Note that it is not even necessary to invoke the assembler in order to make an OSBYTE call. The accumulator is set via A% (the X and Y registers could similarly be set by X% and Y% where necessary) and the result (from the Y register in this case) is read from the USR function.

10.8.5 Saving and loading machine code programs

There are two ways of storing machine code programs. If it is satisfactory to assemble the program every time, it can be stored in assembly form as part of a BASIC program (once development is over, the OPT parameter can be changed to 0 or 2 to suppress the assembly listing).

However, if for some reason this is not desired, the raw machine code can be saved direct from memory with the *SAVE command. This is still possible even if it is to be combined with a BASIC program. BASIC and machine code programs can be stored together by *SAVE, and loaded back with the BASIC command LOAD or CHAIN.

*SAVE simply saves a section of memory to the current filing system. It therefore requires three parameters: the filename, and two parameters which specify the section of memory. The section of memory may be specified in One of two forms. In either case the first parameter given must be the start address of the section with the second being the end address or (if preceded by a + sign) the length of the section. In all cases the parameters must be in hexadecimal. For example

```
*SAVE MCPROG1 C80 D00
```

Would save the section of memory from &C80 to &CFF. Note that the last address specified is not saved.

```
*SAVE BASPROG 1900 +400
```

Would save the section of memory from &1900 to &1CFF.

There are two further optional parameters to *SAVE. The first of these is the *execution address*. If the section of memory being saved is a machine code program, and it is subsequently executed with *RUN, this address is Where execution will start. It is only needed when the execution address is other than the start of the section of memory. Thus if a program is saved with

```
*SAVE MCPROG2 C80 D00 CA0
```

then *RUN MCPROG2 would be equivalent to

```
*LOAD MCPROG2
CALL &CA0
```

The second extra parameter, which will only rarely be necessary, is the reload parameter. Normally the section of memory would be loaded back at the same address as it was saved from, unless otherwise specified by the *LOAD command. But the default reload address can be specified to be elsewhere by the fifth *SAVE parameter.

All of these parameters (except the starting address, if a different reload address is given) can subsequently be determined by the *INFO command once the file has been saved. On the DFS, *INFO MCPROG2 would give the response

```
$.MCPROG2      000C80 000CA0 000080 nnn
```

where nnn is the sector on the disc where storage of MCPROG2 starts. The extra two digits for the addresses are for future expansion of the system. (If the file were locked, this would also be shown in the *INFO listing.)

The file can be reloaded by the *LOAD command, which has only two parameters − the filename, and an optional reload address (in hexadecimal) if you do not want to load the file at the default address. (If you want to *LOAD a data or ASCII file you should always specify the reload address, since the default is &0000 which is unacceptable.)

If the file is a machine code program, then an alternative to *LOAD is to load and run it in one operation, from the execution address specified in the *SA VE command as explained above. This is the machine code equivalent of the BASIC command CHAIN. The command *RUN has the ultimate in abbreviations, to a simple star. Thus instead of *RUN MCPROG2, we could simply use

```
*MCPROG2
```

giving the file all the appearances of an extra OS command. In this form, the file can also be accessed in the library directory (as set by the *LIB command) if it does not exist in the current directory.