Chapter 11

PROGRAMMING TECHNIQUES

11.1 'USER-FRIENDLY' PROGRAMMING

In designing a program it is important to ensure that it will be easy to use effectively − ‘user-friendly’ is the word used to cover all such aspects. In particular it is important to validate any user input and to use attractive display.

Input

There are numerous of ways in which the programmer can make responses easier for the user. For instance, if all responses are by means of a single key, then it is tedious to have to press <RETURN> after every key-press. Similarly, to move objects around the screen, it is nice to use the four arrow keys, rather than a collection of meaningless letter keys. This means redefining the keys. The various methods of input are described in Section 11.2.

Consistency

The most important point here is that the program should be consistent in the Way it expects the user to respond. For instance, if GETs and INPUTs are mixed, the user will not know whether he is expected to press the <RETURN> key, and sometimes having to reply YES and other times Y to a yeslno question is another cause of great frustration, even to a fairly experienced user.

Validation

Whenever the user is asked to make a response to the program, it is important to check both for typing errors, and that the data returned is sensible in the Context of the program. This is discussed fully in Section 11.3.

Display

There is a great deal that can be done in this context, which will be covered in detail in Section 11.4. However, at its simplest it includes such techniques as putting a short message in the centre of the screen rather than in the top or bottom left hand corner. The reaction testing program of Example 11.1 illustrates this.

Information

Within reason, it is desirable that a program should be operable with a minimum of reference to supporting literature, whether this be course notes for a teaching package or an operating manual. This means that the program itself should give as much information as possible. In particular, when asking the user to make a choice it is helpful to explain the significance of those choices. The result may well be a program consisting more of PRINT statements than anything else, but this is not uncommon and should not be a deterrent.

Menu

Whenever a user has to make a choice from a series of options, this should be presented in the form of a menu. The word menu is used in analogy to a restaurant menu (particularly Chinese restaurants) where the list of dishes is presented for the customer’s choice, with a number against each so that a dish can be selected simply by quoting the number.

In the same way a computer program menu allows the user to select his choice simply by pressing the appropriate key. The key-press will often be detected by means of a GET, so if there are more than 9 choices, it is better to use a letter. This can be converted to a number and then used in an ON...GOTO statement. Another possibility is to use initial letters for selection, even if these are not in sequence. For instance, suppose your menu has four choices with initial letters L, S, R and E (ASCII codes 76, 83, 82 and 69) and the code of the key pressed is stored in variable X. By using relational expressions a variable A could be assigned as

```
A=-((X=76)+(X=83)*2+(X=82)*3+(X=69)*4)
```

and A would then be suitable for use in a statement of the form ON A GOTO...

The methods of using menu programs are described in Section 11.7.

11.2 INPUT FACILITIES OF BBC BASIC

One of the great advantages of a microcomputer is that it is ideally suited to interactive use, since one user has sole control of the machine. This interaction may take many forms: perhaps the most exciting is the computer game, but there are many more serious forms, ranging from simply deciding whether to run a program again to positioning a point on a graphical display (which might be to draw a modified character, or to label a feature on a graph, for instance).

The first essential, whenever a user is exvected to interact with a program, is to display a message at the appropriate time so that he knows when and how to interact. The INPUT command is particularly convenient for this purpose, since it allows a message to be printed as part of the INPUT command.

There are, however, a number of other useful ways in which to allow the user to interact, as detailed below.

11.2.1 Joysticks and buttons

For completeness we will mention here that the analogue inputs of the BBC microcomputer make it simple to use joysticks and buttons or similar devices in conjunction with BASIC programs as well as with games. The function ADVAL(N) will read the Nth analogue input channel (N=1 to 4), while ADVAL(0) will indicate whether one or both games buttons are being pressed. For example, X=ADVAL(1) returns into X the value corresponding to the current reading on channel 1 of the analogue input.

Further details of the ADVAL function and use of the analogue inputs are outside the scope of this book.

11.2.2 The GET and GET$ functions

The standard method of receiving INPUT from the user is via the INPUT command. For input of long numbers or strings this is ideal, because it provides editing facilities, in the form of the DELETE key, to correct mistakes during input. For input of a single number or character, however, INPUT is very tedious, as pointed out above.

There is an alternative way of collecting input, via the GET and GET$ functions. These differ from INPUT in that they will accept only a single number or character. They give no new line or prompt and there is no need to press <RETURN>. (Neither do they display the character typed, but you can always PRINT this if required.) Both functions will accept any key, but GET returns a number, the ASCII code of the key, while GET$ returns the single-character string represented by the key. The commands are issued as

`A=GET` or `A$=GET$`

(Note the difference from some other microcomputers, where GET is a Command rather than a function.)

It is possible to use GET$ for multiple character input. Lines 220 to 260 of Example 4.10 use GET$ to input a character string, in the form of an array, Which might be much longer than the 255 character maximum of INPUT.

Exercise 11.1

Modify your program from Exercise 4.14(1) earlier to use GET$ instead of INPUT.

It can on occasion be useful, as pointed out above, to be able to use the cursor editing keys, particularly the arrow keys, for normal input. This can be achieved with the command

```
*FX 4,1
```

After this command has been issued, the arrow and COPY keys can no longer be used for screen editing. Instead, when pressed they return normal ASCII codes, as follows

COPY	135
←	136
→	137
↓	138
↑	139

These keys can be picked up in the usual way with GET$, in a statement such as

```
IF GET$=CHR$(135) THEN...
```

*FX 4,0 restores the keys to their normal function, and note that this must be issued before the end of a program, as the keys will remain disabled after the program has ended. (Note that *FX 4,2 has a second action − it enables the five keys to act as further function keys, with numbers 11 to 15.)

In a similar fashion to *FX 4,1

```
*FX 229,1
```

disables the normal action of the ESCAPE key, and it returns its standard ASCII value of 27. *FX 229,0 restores the normal action.

Exercise 11.2

Write a program to move a character around the screen by means of the four arrow keys, and end when COPY is pressed.

11.2.3 The INKEY function

A feature of both INPUT and GET which is often necessary, but sometimes inconvenient, is that the computer stops and waits for the input. There are times when it is useful, or indeed essential, for the computer to continue the program meantime. (Games are again a very obvious example, where say a gun is occasionally to be fired at a moving target.)

The function INKEY (and the string equivalent, INKEY$) enables the programmer to check for input while a program is running. Whenever a key is pressed it is stored in the keyboard input buffer. The next INKEY function can pick the key value from the buffer when program execution reaches that point.

There are really three separate modes of operation of INKEY. INKEY(N), where N is a positive number, will (if necessary) cause a program to pause for a period of up to N centiseconds to see whether a key is pressed. If a key is pressed during that time, or prior to the INKEY call, INKEY returns the ASCII code of the first item in the keyboard buffer. After N centiseconds the computer gives up and continues operation, the INKEY returning the value −1.

INKEY(N) thus performs a function rather similar to GET, except that GET will cause the computer to wait forever if necessary. The equivalent of GET$ is a closely related function, INKEY$, which returns a single character string variable instead of the ASCII value.

If what you really want is for the program to continue execution while checking for keyboard input, INKEY(()) will do this by checking the keyboard buffer but without any pause. Any program loop could contain, for example, the line

```
150 IF INKEY(0)>=0 THEN GOTO 200
```

which would accomplish this.

However, if a key had been pressed earlier, this would be picked up the first time round the loop. It is possible to ‘flush’ (i.e. empty) the keyboard buffer at any time before you enter the loop by

```
*FX15,1
```

to avoid this possibility.

Example 11.1

The following program uses the properties of INKEY(([]) to give a simple reaction time test. The keyboard buffer is flushed at line 100 just before the loop detecting keyboard entry, to prevent cheating, then a REPEAT...UNTIL loop at lines 140 to 160 keeps track of time while checking for a key-press.

```
 10 CLS
 20 PRINT "When a message appears on the screen"
 30 PRINT '"press any key"
 40 TIME=0
 50 REPEAT: UNTIL TIME>400
 60 CLS
 70 PAUSE=INT(1000*RND(1))
 80 TIME=0
 90 REPEAT: UNTIL TIME>PAUSE
100 *FX 15,1
110 REM CLEARS KEYBOARD BUFFER
120 PRINT TAB(15 2) ;"PRESS NOW!"
130 TIME=0
140 REPEAT
150 T=TIME
160 UNTIL INKEY(0)>=0
170 PRINT '"You took ";T/100;" seconds"
```

```
180 PRINT '"Do you want to try again? (Y/N)"
190 YN$=GET$
200 IF YN$="Y" THEN GOTO 10
```

Since INKEY not only detects a key-press, but also returns its ASCII value, the program could be modified to look for a particular character selected at random. 64+RND(26) would generate the ASCII code for any capital letter, or more ambitiously 32+RND(94) would produce the ASCII code for *any* printable character.

Exercise 11.3

Modify the program above to choose a key at random, measure the reaction time and determine whether the user got it right.

Example 11.2

Another example of the use of INKEY is to temporarily freeze a display, of either text or graphics. In the next program, a series of sine curves are continuously plotted, but the display will be frozen by pressing any key, until a further key is pressed.

```
  10 MODE 4
  20 START=0
  30 REPEAT
  40 MOVE 0,500+400*SIN(RAD(START))
  50 FOR ANGLE=0 TO 360 STEP 5
  60 DRAW 3*ANGLE ,500+400*SIN(RAD (ANGLE+START))
  70 IF INKEY(0)>=0 THEN PROC_wait
  80 NEXT ANGLE
  90 START=START-20
 100 UNTIL START=-360
 110 END
1000 DEF PROC_wait
1010 *FX 15,1
1020 REPEAT
1030 UNTIL INKEY(0)>=0
1040 ENDPROC
```

Line 70 is the key line, where INKEY is used within the plotting loop to check for a key-press, and if detected, call PROC_wait.

The third use of INKEY is with a negative argument. INKEY then checks whether a key is being pressed *at that instant*, returning -1 or zero according to whether the key is pressed or not. This form of the INKEY statement checks for a particular key, the key being determined by the value of the negative argument. The value of the number does not correspond to the key’s ASCII code, but follows an obscure and seemingly random table which is given in Appendix 1.

The most obvious application for this mode of INKEY is for games, where action must be taken according to which key has been pressed, but only for as long as the key is pressed. This mode of INKEY is not applicable to INKEY$ (it would be superfluous anyway).

All keys, including those such as CAPS LOCK, SHIFT, ESCAPE and the function keys, have a code. However, using keys such as ESCAPE or the arrow keys will give rise to a problem, because as soon as the INKEY command has been completed, when the user will almost certainly still have his finger on the key, the usual function will come into effect. It will again be necessary to turn off their normal functions with *FX 4,1 and *FX 229,1 as explained above.

Exercise 11.4

Modify the program of Example 11.2 so that instead of switching the plotting off and on with successive key presses, the display is frozen while the space bar is pressed (INKEY code -99) and plotting starts again as soon as the bar is released.

11.3 VALIDATION

11.3.1 String validation

The first aspect of validation is to ensure that, if the user presses one or more wrong keys the program does not ‘crash’ but instead tells the user that his/her input was wrong and re-poses the original question. Such an approach needs considerably more programming, but it is necessary if the program is to be safely used, particularly by people other than the original author.

Example 11.3

As an example of a helpful friendly style, the following would be an effective way to ask a simple yes/no question

```
 100 REPEAT
 110 PRINT "Random number = ";RND(1)
 120 PRINT "Do you want another number? (Y/N)"
 130 INPUT YN$
 140 IF YN$<>"Y" AND YN$<>"N" THEN PROC_wrongkey
 150 UNTIL YN$="N"
 160 END
1000 DEF PROC_wrongkey
1010 REPEAT
1020 PRINT "Please press the 'Y' key for 'YES'"
1030 PRINT "or the 'N' key for 'NO'"
1040 INPUT YN$
1050 UNTIL YN$="Y" OR YN$="N"
1060 ENDPROC
```

Note that whichever key is pressed (apart from <BREAK>) the result should be intelligible to the user and the program will not crash.

The question requiring a yeslno response is actually a very common situation, but conventions vary as to whether the user responds with Y (N) or YES (NO). The program can be made to accept either response, (but also unfortunately YABOO or NUTS) by using INPUT with the string YN$, then testing the first character of YN$ for a Y or N. Try the following modifications to the above program

```
 140 IF LEFT$(YN$,1)<>"Y" AND LEFT$(YN$,1)<>"N" THEN
     PROC_wrongkey
 150 UNTIL LEFT$(YN$,1)="N"
1050 UNTIL LEFT$(YN$,1)="Y" OR LEFT$(YN$,1)="N"
```

11.3.2 Numerical validation

Because the number of ‘valid’ input strings is usually limited it is relatively easy to check them as above. Checking that a number is valid is more difficult. It is, however, worth mentioning some simple general methods.

In many cases a valid input number must lie within a known range. This can be easily tested with the <, > and = operations and a suitable warning printed if required. Similarly it is sometimes necessary for a number to be an integer. We can test if the number NUM is an integer as follows

```
    ...
    ...
    ...
100 REPEAT
110 INPUT "Type a number: ",NUM
120 IF NUM<>INT(NUM) THEN PROC_invalid
130 UNTIL NUM=INT(NUM)
    ...
    ...
```

where PROC invalid prints out a suitable warning message.

On the BBC microcomputer, a value typed into a numerical variable in response to an INPUT command acts as if it were input as a string and converted to a numeric value by the VAL command (see Section 4.4). Although at first sight this appears to be a user-friendly technique, it suffers from the danger that there is no warning if a non-number key is accidentally pressed. The particularly prevalent mistake is to type the letter O instead of zero, for example 1O for 10, which would be converted to 1 with no warning, or perhaps more disastrously O.1 for 0.1 which would be converted to zero.

To avoid this possibility it is advisable to adopt a somewhat indirect approach to numerical input. You can pre-empt the operation of INPUT if the ‘number’ is initially input as a string variable and the function VAL is then used to convert it to an arithmetic variable (if the conversion is possible, otherwise VAL will give the result zero). Therefore, we can test whether the

STR$ of the arithmetic variable is equal to the original string and, if not, issue a warning and reprompt for input.

Example 11.4

A suitable piece of user-friendly program for the input of a number is

```
     ...
     ...
 100 REPEAT
 110 PRINT "Input a number"
 120 PROC_validate
 130 UNTIL TEST=1
     ...
     ...
 990 END
1000 DEF PROC_validate
1010 REM PROCEDURE TO VALIDATE NUMERICAL INPUT
1020 LOCAL ST$
1030 INPUT ST$
1040 NUM=VAL(ST$)
1060 IF ST$=STR$(NUM) THEN TEST=1: ENDPROC
1070 PRINT "PLEASE DO NOT PRESS ANY NON-NUMBER KEYS"
1080 PRINT "(apart from 'E' for a number"
1090 PRINT "in exponent form)"
1100 TEST=0
1110 ENDPROC
```

These sections of program will assign the required number to the arithmetic variable NUM. The procedure can be used whenever a number is to be input (though it would be necessary to assign NUM to the required variable).

Exercise 11.5

Use the section of program above to input a number directly and print out its Value and also its square. Investigate the effect of typing in a number and also a string including a letter (as if you had made a typing error). Delete lines 100 and 130 and replace line 120 by

```
120 INPUT NUM
```

and repeat the tests. In the latter case there will be no warning of the incorrect input.

One limitation of this kind of validation is that it is necessary for the number to be typed in in the same format as is adopted by the STR$ function for that particular number (i.e. exponent form for numbers outside the range 0.1 to 10^9). A much more complex checking process is required in order to detect any invalid input without imposing this restriction on the user, or alternatively using INPUT directly with a number and risking incorrect input. The next subroutine will accept almost any valid input, but trap any error.

Example 11.5

PROC_validate in the program below will validate the input of a number and return with the number stored in the variable ZQ. This can then be assigned to any desired variable. The program itself serves merely to give you an opportunity to test out the procedure.

```
   10 REPEAT
   20 PRINT "INPUT A NUMBER"
   30 PROC_validate
   40 PRINT "YOUR NUMBER IS: ";ZQ
   50 UNTIL FALSE
   60 END
10000 DEF PROC_validate
10010 REM PROCEDURE TO VALIDATE THE INPUT OF A NUMBER
10020 REM THE NUMBER IS RETURNED IN VARIABLE ZQ
10030 REM DPTS COUNTS DECIMAL POINTS
10040 REM EXN    "    EXPONENT E'S
10050 REM ST MARKS THE START OF NUMBERS
10060 REM NMT COUNTS THE POWER OF 10 OF THE MANTISSA
10070 REM INP$ IS THE NUMBER
10080 REM EX$ IS THE EXPONENT
10090 REM CH$ IS A SINGLE CHARACTER OF THE NUMBER
10100 LOCAL J,DPTS,EXN,ST,NMT,INP$,NUM$,CH$,CANC$,LST$
10110 INP$="": NUM$="": DPTS=0: EXN=0: ST=0: NMT=0
10120 PROC_inchar
10130 PRINT
10140 ZQ=VAL(INP$)
10150 ENDPROC
11000 DEF PROC_inchar
11010 CH$=GET$
11020 IF CH$=CHR$(13) THEN ENDPROC
11030 IF CH$=CHR$(127) THEN PROC_cancel: ENDPROC
11040 PRINT CH$;
11050 IF ((CH$<"0" OR CH$>"9") AND CH$<>"." AND CH$<>"-" AND
      CH$<>"+" AND CH$<> "E" AND CH$<>CHR$(13)) THEN
      PROC_invalid: ENDPROC
11060 IF (CH$="-" OR CH$="+") AND ST=1 THEN PROC_invalid:
      ENDPROC
11070 IF EXN>0 OR DPTS>0 OR CH$<"0" OR CH$>"9" THEN ELSE IF
      NMT<=37 THEN NMT=NMT+1 ELSE PROC_invalid: ENDPROC
11080 IF EXN=0 OR CH$<"0" OR CH$>"9" THEN ELSE IF
      (VAL(NUM$+CH$)+NMT)<=38 THEN NUM$=NUM$+CH$ ELSE
      PROC_invalid: ENDPROC
```

```
11090 IF CH$<>"." THEN ELSE IF DPTS=0 AND EXN=0 THEN DPTS=1
      ELSE PROC_invalid: ENDPROC
11100 ST=1
11110 IF CH$<>"E" THEN ELSE IF EXN=0 THEN EXN=1: ST=0: ELSE
      PROC_invalid: ENDPROC
11120 INP$=INP$+CH$
11130 PROC_inchar
11140 ENDPROC
12000 DEF PROC_invalid
12010 PRINT CHR$(7);CHR$(127);
12020 PROC_inchar
12030 ENDPROC
13000 DEF PROC_cancel
13010 IF LEN(INP$)=0 THEN VDU 7: PROC_inchar: ENDPROC
13020 PRINT CH$;: CANC$=RIGHT$(INP$,1)
13030 INP$=LEFT$(INP$,LEN(INP$)-1)
13040 IF LEN(INP$)=0 THEN LST$="" ELSE LST$=RIGHT$(INP$,1)
13050 IF LST$="E" OR LST$="" THEN ST=0 ELSE ST=1
13060 IF CANC$="." THEN DPTS=0
13070 IF CANC$="E" THEN EXN=0
13080 IF CANC$="+" OR CANC$="-" THEN ST=0
13090 IF EXN=0 AND DPTS=0 AND CANC$>="0" AND CANC$<="9" THEN
      NMT=NMT-1
13100 IF EXN>0 AND CANC$>="0" AND CANC$<="9" THEN
      NUM$=LEFT$(NUM$, LEN(NUM$)-1)
13110 PROC_inchar
13120 ENDPROC
```

Note the use of recursion, and also the use of null statements between THEN and ELSE to avoid the ambiguity of IF...THEN...IF...THEN...ELSE.

11.4 DISPLAY LAYOUT

A well thought out and attractive presentation of text is necessary for programs which are to be pleasant and interesting to use. BBC BASIC includes several useful commands to control the presentation of text.

11.4.1 Layout of text

The layout of numbers is more complicated than that of text, so we will deal with text first.

All screen character output takes place at the cursor position, and the cursor can be positioned prior to printing in a variety of ways. The simplest is by means of the alternative PRINT list separators, the semicolon (;), comma (,) and apostrophe (ʼ).

The effect of the apostrophe is simplest. It generates a new line and returns the cursor to the left of the screen. It is the one item of punctuation in

BBC BASIC which is not standard in other versions of BASIC. It has been used fairly extensively in programs listed in this book, particularly to avoid repeated PRINT statements. Thus

```
PRINT ''"LEAVE A SPACE"'"NEW LINE"
```

is equivalent to

```
PRINT: PRINT: PRINT "LEAVE A SPACE": PRINT "NEW LINE"
```

The effect of the semicolon is to separate items which are to be printed consecutively, with no intervening spaces. For instance

```
PRINT "DIAL";999;"F0R POLICE"
```

This appears as

```
DIAL999FOR POLICE
```

with no spaces. In order to space it out properly, it is essential to put spaces inside the quotation marks

```
PRINT "DIAL ";999;" FOR POLICE"
```

`PRINT "DIAL";999; "FOR POLICE"` will not do

Note that very often it is unnecessary to include the semicolons. Whenever the computer can make sense of the print list without them, it will do so. Generally speaking, almost anything but numbers or real variables can be PRINTed without intervening semicolons. The problem with numbers and real variables is that they simply merge into one another. Even string and integer variables, on the other hand, can be distinguished by the $ or % at the end.

Thus we could put

```
PRINT "DIAL "999" FOR POLICE"
```

Similarly

```
A$= "STRINGING "
B$= "THINGS "
C$= "TOGETHER "

PRINT A$B$C$
```

will produce the desired message, because the computer recognizes the dollar signs as denoting the end of each string variable.

However, it is not always possible to leave the semicolons out. Try the effect of

```
PRINT "STRINGING ""THINGS ""TOGETHER"
```

Two consecutive quotation marks are used to generate an actual quotation mark within a string. (They are also needed at the start of a DATA item or a string response to INPUT, because these may optionally be enclosed in quotation marks − for example, if a comma is to be included in the string. See Section 4.2.3.)

Generally speaking, it is best to develop the good habit of always including the semicolon.

One extra feature of the semicolon is that, if used at the end of a PRINT statement, it suppresses the newline, and leaves the cursor immediately after the printed text, which will enable you to carry on printing at a later stage if you wish. Try

```
PRINT "PUTTING THINGS";: PRINT "TOGETHER"
```

Also try just

```
PRINT "SO FAR SO GOOD";
```

The comma serves quite a different function. It behaves exactly like a preset tabulator key on a typewriter, so that each comma moves the cursor to the next available tab position across the screen. Normally the tab positions are ten characters apart. Try

```
PRINT "ME","AND YOU","AND US","."

PRINT "TOO LONG FOR ONE","TAB POSITION"
```

Ending a PRINT statement with a comma will not cause the next PRINT to occur on the same line.

The default ‘tab setting’ of 10 characters can be changed. This value forms the first byte of the resident integer variable @?%, so changing this, for example to

```
@%=@%+5
```

will increase the spacing to 15 characters.

Exercise 11.6

Experiment with layouts of strings while altering the value of @%.

11.4.2 TAB, SPC and POS

More elaborate layouts are possible with various combinations of the PRINT commands TAB and SPC, and the function POS.

The TAB command has two forms. The simpler is TAB(X), which when used within a PRINT statement moves the cursor along to column X in the line. It thus acts like an adjustable tabulator, giving much better control over layout. A few points should be noted about TAB. The range of X is 0 to 255. Larger numbers are treated modulo 256 (as are negative numbers). Non-integer numbers are truncated (not rounded). Numbers larger than the screen width will cause wrap round for as many lines as necessary. Also, if column X has been passed, wrap round will occur to column X on the next line. This last point is a potential source of danger.

Note that the left hand column of the screen is column 0, not 1, and thus on a 40 column display the right hand column is 39. This is actually fairly logical, since PRINT TAB(5);"STRING" will generate 5 spaces before STRING.

Another way in which output can be controlled is with the SPC(N) command, which must be used in the same way as TAB(X) within a PRINT statement. SPC(N) will move the cursor on N spaces, wrapping round if necessary. The value of N follows the same rules as for X in TAB, so the effective range is 0 to 255.

A useful function in conjunction with TAB and SPC is POS, which returns the current column in which the cursor lies. This could be used with TAB to check that the cursor had not passed the intended TAB position, or with SPC to produce an effect similar to TAB. It will be appreciated that there is a large element of duplication in these functions, giving the programmer the choice of his preferred technique, as is emphasized by the following exercise.

Exercise 11.7

Try the following statements which all give an equivalent layout

```
PRINT "OTHER,A.N.(type 5 spaces)01-234-56 78(type 4 spaces)LONDON"
PRINT "OTHER,A.N.";SPC(5);"01-234-5678";SPC(4);"LONDON"
PRINT "OTHER,A.N.";TAB(15);"01-234-5678";TAB(30);"LONDON"
PRINT "OTHER,A.N.";SPC(15-POS);"01-234-5678";SPC(30-POS);
      "LONDON"
```

Exercise 11.8

Write a procedure, PROC_table, to print out as a table the string array A$ having DIMension A$(10,5), in the format of 10 rows of 5 columns, each 7 characters wide. The table should be headed with the string array HEAD$ (DIMension HEAD$(5)), which should be separated from the body of the table by two blank lines. Use it in conjunction with the following program.

```
10 DIM A$(10,5) ,HEAD$(5)
20 CLS
30 FOR J=1 TO 5 : HEAD$(J)="COL"+STR$(J): NEXT J
40 FOR J=1 TO 5
50 FOR K=1 TO 10
60 A$(K,J)=STR$(K)+STR$(J)
```

```
 70 NEXT K
 80 NEXT J
 90 PROC_table
100 END
```

11.4.3 TAB(X,Y)

The second, and much more powerful, use of TAB is in the form TAB(X,Y), with two parameters which give the horizontal and vertical positions on the screen to which the cursor is to be moved. As with TAB(X), the left hand column is column zero, and the top row (not the bottom) is row zero.

TAB(X,Y) will move the cursor immediately to any point on the screen, without printing intervening spaces as TAB(X) does. Thus TAB(X,Y) does not overwrite the earlier part of the line, or indeed anywhere else on the screen. The whole philosophy of output to the screen should be altered when using TAB(X,Y), so that instead of printing lines successively down the screen, output is directed and redirected to any point on the screen.

If either parameter of TAB (X,Y) is out of range (outside the values 0 to 39 for X and 0 to 24 for Y in Mode 7, for example), the command is ignored completely, in contrast to TAB(X).

Example 11.6

As an example of the use of TAB(X,Y), the program below prints out a string forwards and then backwards.

```
 10 REM PROGRAM TO ILLUSTRATE TAB(X,Y)
 20 CLS
 30 A$="A MAN A PLAN A CANAL PANAMA"
 40 L=LEN(A$)
 50 LFT=INT(19-L/2)
 60 FOR J=1 TO L
 70 PRINT TAB (LFT+J-1,8);MID$(A$,J,1)
 80 PRINT TAB(LFT+L-J,16) ;MID$(A$,L+1-J,1)
 90 TIME=0
100 REPEAT: UNTIL TIME=20
110 NEXT J
120 PRINT TAB(0,24);
```

If you have the supplementary disc which is available to accompany this book, program TABDEMO shows a more sophisticated use of TAB(X,Y), for an attractive method of input of data to set up a simple database.

Exercise 11.9

If you have the supplementary disc, program CLOCK on the disc allows you to set a time and then provides the time as the variables HOURI, MIN1 and SEC1. It also provides a stopwatch function so that pressing 1 starts the watch, 2 stops it and 3 resets it. The stopwatch time is provided in HOUR2, MIN2 and SEC2. Write a procedure, PROC clock, needed for program CLOCK, to display the time and the stopwatch readings on the centres of lines 8 and 16 respectively.

Corresponding to the function POS, which is useful in conjunction with TAB (X) and SPC, there is a second function VPOS which returns the vertical position of the cursor. Thus to move to the start of the current line, you might use

```
PRINT TAB(0,VP0S);
```

After using TAB(X, Y) in a PRINT statement, note that TAB(X) and commas will be reset to measure from the position of the cursor after the TAB(X,Y) call, instead of the left-hand edge of the screen.

11.4.4 Output to a printer

On the BBC microcomputer, the layout of printer output is quite simple, because the functions TAB(X), SPC and POS all work for printer as well as screen output. TAB(X, Y) of course cannot possibly work with printers, and is ignored. Moving printer paper backwards requires special codes particular to each printer, so all printed output should be planned to move continuously down the page.

There are two extra commands, WIDTH and COUNT, which affect both screen and printer output, but are probably most relevant to the latter.

WIDTH N

Sets the *page width* to N characters. It can thus be used to limit the screen display (especially perhaps for Mode 0) , but it also generates a newline code and so could be particularly useful to limit the width of printer output. For example

```
WIDTH 72
```

would restrict an 80 column printer to 72 columns, which would be a more suitable width for A4 paper.

COUNT

Is a function returning the number of characters output since the last newline. Within one screenwidth it is equivalent to POS, but POS relates only to the cursor position on the screen. Thus COUNT should be used rather than POS for printer output, where wrap-round may have occurred on the screen.

11.5 FORMATTING NUMERICAL OUTPUT

11.5.1 Layout of numbers

All of the previous discussion of layout of text also applies to numbers, that is, the printing of real or integer variables. In addition, however, there are, considerations of field width, significant figures, format and mode of display.

The first point to notice is that numbers are normally printed right justified. Thus if we issue the command

```
PRINT 12345,67890
```

the result is

```
.....12345.....67890
```

(Note that the dots do not appear on the screen, but indicate the number of spaces between the characters printed.) The effect of the semi-colon is not only to act as a separator in the PRINT list, but also to cancel this right justification. Try

```
PRINT ;12345,67890
```

The reason for the right justification is so that integer numbers of different lengths, printed in a column, all have their units column aligned. Try

```
PRINT 1: PRINT 23: PRINT 456
```

However, real numbers with decimal fractions cannot be aligned this way unless they all have the same number of decimal places. Try

```
PRINT 1.2: PRINT 3.4: PRINT 5.67
```

If you want to retain this right justification while tabulating with TAB(X) or TAB(X, Y) rather than commas, you must omit the semicolons which good practice would normally dictate that you should include. Try

```
PRINT 1 TAB(15) 23: PRINT 45 TAB(15) 678
```

then

```
PRINT 1;TAB(15);23: PRINT 45;TAB(15);678
```

On the other hand, if you want numbers printed with spaces in between, then you must include the semicolons. Compare

```
PRINT 12" "34" "56  and PRINT 12;" ";34;" ";56
```

11.5.2 Controlling the format of printed numbers

It is often desirable to have a close control over the format of printed numbers. Two very common requirements are to fix the number of decimal places displayed, and to print decimal numbers in columns such that the decimal points are properly aligned.

To a limited degree these features can simply be achieved with the aid of the resident integer variable @%. To print all numbers with N decimal places, in columns of width W, set

```
@%=&20000+&100*N+W
```

Example 11.7

The program below generates numbers whose magnitudes and number of figures, as well as values, are random, and prints them with 2 decimal places in 3 columns of width 10 characters. This would for instance be very suitable for displaying columns of money in pounds.

```
  10 CLS
  20 @%=&2020A
  30 FOR ROW=1 TO 20
  40 FOR C0L=1 TO 3
  50 REPEAT
  60 NUM=FNrannum
  70 UNTIL ABS(NUM)<1E5 AND ABS(NUM)>0.1
  80 PRINT ,NUM;
  90 NEXT COL
 100 PRINT
 110 NEXT ROW
 120 END
1000 DEF_FNrannum
1010 LOCAL DECPLACES, NUM
1020 DECPLACES=10^RND(6)-1)
1030 NUM=INT((RND(1)-0.5)*10^RND(9))
1040 =NUM/DECPLACES
```

It would be as well to press <BREAK> after using this program, to restore the default value for @%.

The variable @% can do a great deal more than just produce a fixed number of decimal places, but its use is quite complicated, and is therefore described in Appendix F.

The biggest limitation of the example above is that all columns must be of equal width and, moreover, the decimal places are displayed even for integers. We might, however, want a column of 3 figure integers, two columns of numbers of up to 7 significant figures, having two decimal places, then two columns of numbers less than one, having just 3 decimal places. What is needed is a way in which successive numbers can be printed in different formats.

Example 11.8

The following procedure will make it easy to print numbers with close control over their format. It prints the number NUM, starting from the cursor position on the current line, with up to N decimal places and a field width of W (or more, if the number cannot be accommodated in that number of characters). The cursor is left at the end of the number. Numbers are justified within the field width so that the decimal point is N places from the right of the field, and only significant figures are printed. If N is set to 0, numbers are printed as integers. (The procedure will not cope with numbers larger than 10^9 or smaller than 0.1 that need to be in exponent form.)

```
10000 DEF PROC_formprint(NUM,N,W)
10010 REM PROCEDURE TO PRINT NUM
10020 REM WITH A FIELD WIDTH OF W
10030 REM AND N DECIMAL PLACES.
10040 LOCAL ST1$,ST2$,ST3$,ST4$,ST5$,M,J,DPPOS
10050 @%=@% OR & 1000000
10060 ST1$=" "+STR$(NUM)
10070 ST2$=" "
10080 ST3$="": ST4$=""
10090 ST5$="."
10100 IF NUM=INT(NUM) THEN ST1$=ST1$+".":ST5$=" "
10110 M=W-N+(N>0): REM M IS MINIMUM NUMBER OF FIGURES BEFORE
      DECIMAL POINT
10120 DPPOS=0
10130 REPEAT
10140 DPPOS=DPPOS+1
10150 UNTIL MID$(ST1$,DPP0S,1)="." OR DPP0S=LEN(ST1$)
10160 IF DPPOS>M THEN ST3$=LEFT$(ST1$,DPPOS-1): GOTO 10210
10170 ST3$=LEFT$( ST1$,DPPOS-1)
10180 FOR J=DPPOS TO M
10190 ST3$=ST2$+ST3$
10200 NEXT J
10210 ST4$=RIGHT$(ST1$,LEN(ST1$)-DPP0S)
10220 FOR J=1 TO N
10230 ST4$=ST4$+ST2$
10240 NEXT J
10250 PRINT ;ST3$;
10260 IF N>0 THEN PRINT ;ST5$+LEFT$(ST4$,N);
10270 ENDPROC
```

Adapt the program from Example 11.7 to generate numbers with the same random format as before, and print them using the above procedure, by deleting line 20 and replacing line 80 by

```
80 PR0C_formprint(NUM,4,13)
```

Add the procedure above and test out the program. You will probably find this procedure easier to use than @% as described in Appendix F.

Exercise 11.10

Write a program to use PROC_formprint to print numbers (generated at random) in the format suggested earlier, that is one 3 figure integer, two 7 figure numbers and so on.

In general use of this procedure, it is important to make W large enough for all possible numbers that might occur (and perhaps check their size before printing).

11.6 GENERAL PRESENTATION

There are several features that will generally add to the presentation of the output of a program. The most obvious is that you should always clear the screen at the start, either with CLS or, better still, a MODE command which will also ensure that you are in the desired mode.

Another small point is that whenever you want to attract the user's attention, for instance when prompting for input or warning of a mistake in entering a value, use the 'bell' character, either with PRINT CHR$(7) or VDU 7.

As well as using sound to attract a user's attention, it is possible to emphasize text in various ways, to draw attention to it or simply to make a presentation more attractive.

When we consider the ways of producing attractive presentation we must distinguish Mode 7, which is different from all the other modes.

11.6.1 Mode 7

The method of producing special displays in Mode 7 is quite unlike that in other modes. These are not produced by BASIC commands such as COLOUR, but by ASCII codes printed (invisibly) on the screen. Only predefined characters and shapes can be used, at a fixed set of character positions, and colours are fixed for a whole character position at a time.

The text display consists of 25 lines of 40 characters, making 1000 character cells, and all the screen control is through the contents of these cells. If a cell contains one of the special ASCII codes controlling the display, it modifies the way in which the characters to the right of the code on the same line are interpreted.

Try the following commands in Mode 7.

```
PRINT "ABCD";CHR$(129);"efgh";CHR$(130);"IJKL"
```

ASCII codes 129 and 130 alter the colour of the rest of the line to red and green respectively. Notice that the codes each occupy a character cell. It is not possible to produce a continuous line of text, such as

```
ABCDefgh
```

with, say, the lower case letters of different colour to that of the capitals. The same range of steady colours is available in Mode 7 as in Mode 2 (with the exception of black), and the codes that produce them are shown in Table 11.1.

Table 11.1 Special display codes for Mode 7.

		Special effects	
Colour	*Code*	*Code*	*Purpose*
red	129	136	flash on
green	130	137	flash off
yellow	131	141	double height characters
blue	132	140	normal characters
magenta	133	157	new background colour
cyan	134	156	black background
white	135	152	conceal display

It is possible to make the colours flash, but unlike other modes the flash is between the single colour and the background, and the onloff periods are of unequal duration.

As we have seen before, there are two ways to produce ASCII codes. One way is to use CHR$() from within a PRINT statement. The second way, which is more convenient when a string of ASCII codes i to be issued, as will sometimes be necessary, is via the VDU command. All the values following VDU are issued as ASCII codes. In other display modes, VDU is used as the most convenient way to issue low value ASCII control codes which require parameters, but in Mode 7 it will equally well issue the high values needed.

There is actually a third way to produce the display control codes. Pressing <SHIFT> and a function key simultaneously will produce the character with ASCII code of 128 plus the function key number. This is primarily designed for use in direct mode, but the characters can be embedded in PRINT strings to produce, for example, coloured text from within a program. Try

```
PRINT "ABCD<SHIFT-f1>efgh<SHIFT-f2>IJKL"
```

This method has the additionial advantage that the colours show up when the program is listed in Mode 7, but watch out for the fact that the codes will not be apparent in a printer listing of the program.

Just as in the other modes, it is possible to change the background from black to a colour, or back again to black. Code 157 must be issued after the code of the background colour you want. You will then need to issue a new colour code for the foreground text, or it will be invisible. For example, to print in green on a red background you would need

```
VDU 129,157,130: PRINT "THIS MAKES ME SEE RED"
```

This is where the VDU command becomes particularly useful.

You will see that the above line creates a complete background line in red. You could prevent this by finishing with code 156, which causes the background to revert to black. However, the leading ‘spaces’ of the two codes cannot be prevented from appearing in red. Try

```
VDU 129,157,130: PRINT "ON AND OFF";: VDU 156: PRINT
```

Another special effect in Mode 7 is the production of double height (but normal width) characters, which is initiated by code 141 (and switched off, if necessary later in the line, by 140). An unexpected feature, but which can be understood from the strict way that Mode 7 works with character cells only, is that double height letters must be printed twice on successive lines, both lines being identical (including code 141 at the start). Try the effect out with lines such as

```
PRINT CHR$(141);"A BIG MESSAGE": PRINT CHR$(141); "A BIG
MESSAGE"
```

Try also the effect of disobeying the rules, with for instance

```
PRINT CHR$(141);"NOT A": PRINT CHR$(141) ;"MATCH"
```

There are four obvious ways when using Mode 7 to emphasize a section of text.

1. Use coloured text.
2. Use double height text.
3. Use flashing text. This is particularly eye catching, and can be used to tell the user what to do − for example

   ```
   PRINT CHR$(136);"Press space bar to continue"
   ```

   In other contexts it should be used in moderation or it can become overpowering.
4. To print text on a white or coloured background (unlike other modes, it is not possible to have black on a coloured backgound).

The following program illustrates all these techniques.

Example 11.9

```
10 MODE 7
20 CLS
30 DIM DY$(7)
40 FOR I=1 TO 7: READ DY$(I): NEXT I
50 PRINT TAB(10,6);CHR$(130);CHR$(141);"SPELLING TEST"
   'TAB(10);CHR$(130);CHR$(141);"SPELLING TEST"
```

```
60 PRINT TAB(10,9);CHR$(131);"Days of the week"
70 FOR I=1 TO 10
80 DAY=RND(7)
90 PRINT TAB(0,12);"What is the name of day ";DAY; " of the
   Week? "
100 PRINT SPC(14);: INPUT DAY$
110 PRINT TAB(0,12) ;SPC(80)
120 IF DAY$=DY$(DAY) THEN PRINT TAB(13,12);CHR$(129);
    CHR$(136);"CORRECT":SC=SC+1: ELSE PRINT TAB(0,12);
    CHR$(132);"Day ";DAY;" of the week is spelt:-";
    CHR$(136);DY$(DAY)
130 PRINT TAB(5,23) ;CHR$(136); CHR$(134);"Press any key to
    continue": A$=GET$
140 PRINT TAB(0,23);SPC(40);
150 NEXT I
160 CLS
170 IF SC>=8 THEN PRINT TAB(10,10);CHR$(141);
    "CONGRATULATIONS"'TAB(10);CHR$(141);"CONGRATULATIONS"
180 PRINT TAB(7,15);CHR$(133);"You scored ";SC;" out of 10"
190 DATA SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY
    SATURDAY
```

11.6.2 Modes 0 to 6

With Modes 0 to 6 almost unlimited effects are possible, but this would generally involve the use of graphics. With Modes 1, 2 or 5 dramatic effects with colour are also possible. The simple commands available for graphics, plus related commands also applying to text, are dealt with in Chapter 6. We shall therefore deal with just one technique, which can be used in *all* modes − printing in inverse, that is black on a white background. If you wished, it should not be difficult to modify the procedure to print in one colour on a differently coloured background.

Example 11.10

The following procedure will print in inverse the string or string variable TEXT$, in any mode (even Mode 7, but here the text has to be in blue, because you cannot print in black in this mode).

```
10000 DEF PROC_inverse(TEXT$)
10010 LOCAL WHITE
10020 A%=135
10030 MD=USR(&FFF4)
10040 MD=(MD AND &FFFFFF) DIV &10000
10050 IF MD=7 THEN GOTO 10130
10060 WHITE=1 10070 IF MD=1 OR MD=5 THEN WHITE=3
10080 IF MD=2 THEN WHITE=7
10090 COLOUR 128+WHITE: COLOUR 0
10100 PRINT TEXT$;
```

```
10110 COLOUR 128: COLOUR WHITE
10120 ENDPROC
10130 VDU 157,132: PRINT TEXT$;
10140 VDU 156,135
10150 ENDPROC
```

Exercise 11.11

Use PROC inverse above to enhance one of the earlier programs, or a program of your own, with inverse text.

11.7 MENU DRIVEN PROGRAMS

A useful feature of a disc-based BBC microcomputer is its ability to function as a ‘turnkey’ system − which automatically loads and runs the required program when <SHIFT-BREAK> is pressed. The method of doing this was explained in Section 8.5. If a turnkey system is simply required to run the main program on the disc, then the !BOOT file will probably be a *EXEC ASCII file containing the single command CHAIN "<main program name>".

The idea of a turnkey system is that it should make it as easy as possible for a complete non-programmer to get the program on the disc up and running. Very often, a disc will contain more than one program; the standard way of allowing the user to select the particular program that he wants is by means of a *menu*. A list of the available programs is presented to the user, like a restaurant menu, and the user selects one by a single key − usually either a number or letter. Instead of the main program on the disc, the !BOOT file must CHAIN a preliminary menu program.

As an example, suppose that a disc contains 3 main programs

1. PAY − a program that deals with the company payroll.
2. TAX − a program that deals with tax liabilities.
3. BONUS − a program which deals with the company’s bonus and incentives scheme.

Example 11.11

The program (MENU) below shows a suitable preliminary menu program for a menu driven turnkey system.

```
10 REM MENU PROGRAM
20 CLS
30 DIM P$(3)
40 P$(1)="PAY"
50 P$(2)="TAX
60 P$(3)="BONUS"
70 PRINT "The programs avaiLabLe on this disc"
80 PRINT '"deal with:"
90 PRINT '"1)  The company payroll"
```

```
100 PRINT '"2)  Tax liabilities"
110 PRINT '"3)  The company's bonus and incentive"
120 PRINT "scheme"
130 PRINT '""Press 1, 2 or 3 for your choice"
140 A$=GET$
150 IF VAL(A$)<1 OR VAL(A$)>3 THEN PRINT "YOU MUST PRESS KEYS
    1, 2 OR 3": GOTO 140 160 CHAIN P$(VAL(A$))
```

It is frequently convenient in a turnkey system for the last executed statement of the programs on the disc (PAY, TAX and BONUS in the above example) to bring the user directly back to the menu by rerunning the menu program. The command

```
CHAIN "MENU"
```

will achieve this.

Example 11.12

A more elaborate menu, which is another example of the use of attractive presentation, is given in the program below.

```
  5 *TV0,1
 10 MODE 1
 20 DIM PROG$(6)
 30 PROG$(1)="PAY"
 40 PROG$(2)="TAX"
 50 PROG$(3)="BONUS"
 60 PROG$(4)="PROFIT"
 70 PROG$(5)="BALANCE"
 80 PROG$(6)=" INVESTMENT"
 90 PRINT TAB(0,3);"This disc contains programs for the"
100 PRINT '"company's financial affairs"
110 PRINT
120 STARS$=STRING$(40,"*")
130 PRINT STARS$;
140 GAP$="*"+STRING$(38," ")+"*"
150 PRINT GAP$;
160 PRINT "*   1) The company payroll             *";
170 PRINT GAP$;
180 PRINT "*   2) Tax liabilities                 *";
190 PRINT GAP$;
200 PRINT "*   3) Bonus and incentive scheme      *";
210 PRINT GAP$;
220 PRINT "*   4) Profit and loss account         *";
230 PRINT GAP$;
240 PRINT "*   5) Balance sheet                   *";
250 PRINT GAP$;
```

```
  260 PRINT "*   6) The company's investments       *";
  270 PRINT GAP$ ;
  280 PRINT "*   7) END                             *";
  290 PRINT GAP$ ;
  300 PRINT STARS$
  310 PRINT "Press: ";
  320 PROC_inverse("<SPACE BAR>")
  330 PRINT " to move up and down"
  340 PRINT "                   the list"
  350 PRINT 'SPC(7);
  360 PROC_inverse("<RETURN>")
  370 PRINT " to run required program"
  380 VDU 23;8202;0;0;0;: REM TURN OFF CURSOR
  390 VDU 19,1,8;0;: VDU 19,130,143;0;
  400 PROGNUM=2: DIRN=-1: CURS=11
  405 PROC_arrow
  410 REPEAT
  420 CH$=GET$
  430 IF CH$=" " THEN PROC_arrow
  440 UNTIL CH$=CHR$(13)
  450 MODE 1: REM ALSO RESTORES CURSOR
  460 IF PROGNUM<7 THEN PRINT TAB(7,15);:PROC_inverse
      ("Loading requested program"): B%=PROGNUM: C%=1: CHAIN
      PROG$(PROGNUM)
  470 END
10000 DEF PROC_inverse(TEXT$)
10010 COLOUR 129: COLOUR 0
10020 PRINT TEXT$;
10030 COLOUR 128: COLOUR 1
10040 ENDPROC
20000 DEF PROC_arrow
20010 PRINT TAB(2,CURS);"  ";
20020 IF PROGNUM=1 OR PROGNUM=7 THEN DIRN=-DIRN
20030 PROGNUM=PROGNUM+DIRN
20040 CURS=2*PROGNUM+7
20050 COLOUR 1: COLOUR 130
20060 PRINT TAB(2,CURS);"->";
20070 COLOUR 4: COLOUR 128
20080 ENDPROC
```

Exercise 11.12

Write a !BOOT file and menu program so that, when <SHIFT-BREAK> is pressed, the programs on the disc are automatically catalogued. Then the user is asked which program is to be run, the answer is loaded into a string P$, say, and that program is CHAINed.

Exercise 11.13

Write a menu driven program to carry out one of a set of possible mathematical operations on a provided argument. The program should first use GET to choose from the following list of operations on a number N

reciprocal of N
ten to power N
factorial of N
PI to N decimal points
N random numbers

Use a second GET to request a single number argument for the operation, with an explanatory request.

(Factorial N can be obtained from PROD=1: FOR J=1 TO N: PROD= PROD*J: NEXT J).