**The Professional Touch**

**- how to turn your**

**software into a robust**

**commercial product**

*Alan and Sue Rowley*

Copyright © A. & S. Rowley

**All Rights Reserved**

ISBN 1 85058 033 2

**Published by:**

SIGMA PRESS

5 Alton Road

Wilmslow

Cheshire

U.K.

**Distributors:**

UK, Europe, Africa:

JOHN WILEY & SONS LIMITED

Baffins Lane, Chichester

West Sussex, England

**Australia:**

JOHN WILEY & SONS INC.

GPO Box 859, Brisbane

Queensland 40001

Australia

Printed and bound in Great Britain by

J. W. Arrowsmith Ltd., Bristol

## ACKNOWLEDGEMENTS AND DEDICATION

We should like to thank Acorn Computers Ltd for the loan of a Z80 second processor for the BBC and Microworld Ltd of Edinburgh for lending us an Electron Plus 3 interface.

We should also like to thank Sydney Rowley for preparing many of the figures with great care and effort to a standard to which we could never have aspired.

This book is dedicated to Nicolas and Jessica who will perhaps become programmers themselves in the not too distant future and to their grandma who is much missed by us all.

# PREFACE

This is not another book on programming style, rather it is about program presentation. You will not find any discussion of 'structure' nor dire warnings against GOTO. We are not worried about what may be lurking under the surface of your software but rather about what is seen on the screen and how the computer behaves when the program is at work. We take it for granted that you can write and debug BASIC programs so that they run and do the job that you want them to do, but what it you want to have your programs used by other people? Perhaps you want to try to sell them or to use them in a school or work environment.

Just having a working program is not now good enough. You will have inexperienced, clumsy, or perhaps even downright perverse influences brought to bear on your work. The program must run without 'crashing' and it must be 'user friendly'; that is easy to use even if you are unaware of its innermost workings and limitations. Ideally the friendliness should be extended into 'helpfulness', and you should try to make the software protect the user from making mistakes as far as you can anticipate them. You could let someone enter a date as '30th February' but it would be considerate to prevent it!

Even the simplest program can lead to pitfalls. Let us just examine a simple routine for dividing two numbers and printing the result:

10 INPUT "FIRST NUMBER" N1

20 INPUT "SECOND NUMBER" N2

30 PRINT "RESULT" N1 / N2

We would all probably spot the need to prevent zero being entered as the second number since, apart from being numerical nonsense, it would cause a 'divide by zero' error, but there are all sorts of other possible traps. What if someone enters a number and uses the letter 'O' instead of zero; or if the number is too large for the computer to cope with? This is the sort of question that you have to start asking yourself, and then you must set about making your software able to cope with the consequences of the answer. You need to become a student of human fallibility, and perversity for that matter.

If our little routine is designed to do the division for a particular purpose, say to calculate the average number of oranges that a person eats in a year, then it is reasonable to help the user by now allowing silly input. The first number would be a lifetime's number of oranges, so it cannot be less than zero and will also have some sensible upper limit which we might ponder upon. It must also, presumably, be a whole number. The second number, which now represents the number of years that the subject has lived, must be between 1 and, say, 110. It would be friendly, therefore, to build in some realistic constraints.

Although input is one of the most vulnerable parts of any program it is not the only part that must be robust. What about problems arising from the filing system? Maybe someone puts in an unformatted disc, should they pay for their mistake with a crash that loses all their data? What if they try to use a non-existent printer? Will the machine hang up leaving them with a memory full of wordprocessor text and no way of retrieving it?

Aside from these fairly specific problems you will also have to consider the more general aspects of ease of use of your software, and making the person running it feel in control. It is not very satisfactory, for example, to leave the poor soul looking at a screen with no indication of what, if anything, he must do next. Is the machine waiting for a key press, or is it hard at work with his data so he can sit back and wait for the output? There should always be a clear indication on the screen of what is going on.

If you really want to write friendly robust software you will have to start thinking about this kind of thing otherwise you will certainly not sell much software and you could be responsible for giving the microcomputer a bad name. No computer ever made a mistake, but software is another matter!

Most of the work in producing software for other people is concerned with making it safe and simple to use. Our three line program example quoted above could well have expanded to thirty lines by the time it is properly packaged. It is with this 'packaging' that this book is concerned. By the end we hope you will have a fairly good understanding of how to get from working program to good software.

This is primarily a book about software in BBC BASIC and virtually all the examples will run on any machine using that language. There are some machine-specific features in some of the routines, however, especially in those sections of the book which deal with the disc filing systems. As a general rule all of the listings will work on the BBC Model B, the B Plus or an expanded Model A, with the 1.20, or later, operating system and on the Acorn Electron. The only exceptions are a few routines in chapter 3 which are specific to mode 7, and so will not run on the Electron. The disc filing routines which use the Acorn DFS were developed on a BBC Model B and the ADFS routines on an Acorn Electron with a Plus 3 interface.

All the listings in this book are self-contained in the sense that each is a complete, runnable program in its own right. As far as possible we have constructed the programs as a series of procedures and functions under the control of a short main loop. You can, therefore, enter the complete listings and try them out for yourselves, whereas the functions and procedures can be extracted for use in other contexts with little, if any, modification.

We have referred to 'the user' throughout this book in the male gender. This implies no sexist bias on our part, we are after all one of each kind ourselves, but any alternative mode of expression would have resulted in some very cumbersome writing. Please accept that we use the terms 'he' and 'his' in reference to the whole of personkind. The term BBC is used throughout to refer to the BBC Microcomputer system rather than in its other context.

CP/M is a trade mark of Digital Research and, in this book, refers specifically to the CP/M 2.2 system as implemented on the BBC Micro's Z80 second processor. Z80 is a trademark of Zilog Inc.

*Alan & Sue Rowley, May 1985*

**CONTENTS**

[Chapter 1: Controlling the Program 9](#_Toc380325299)

[Menu Driving 10](#_Toc380325300)

[Command Driving 16](#_Toc380325301)

[Using the Function Keys on the BBC 21](#_Toc380325302)

[Chapter 2: Input of Data from the Keyboard 23](#_Toc380325303)

[Protecting Numerical Input 24](#_Toc380325305)

[String Input 28](#_Toc380325306)

[Screen Handlers 32](#_Toc380325307)

[Editing Data 37](#_Toc380325308)

[Conclusions 40](#_Toc380325309)

[Chapter 3: Output to Screen and Printer 41](#_Toc380325310)

[Choosing the Mode 41](#_Toc380325312)

[Output Screen Handling 42](#_Toc380325313)

[Numerical Formatting 45](#_Toc380325314)

[Text Output 47](#_Toc380325315)

[Printers 49](#_Toc380325316)

[Manipulating the Character Set 52](#_Toc380325317)

[Advanced Mode 7 Colour Control 60](#_Toc380325318)

[Conclusion 64](#_Toc380325319)

[Chapter 4: Error Trapping and Avoidance 65](#_Toc380325320)

[Changing Error Messages 67](#_Toc380325322)

[Fatal Errors 68](#_Toc380325323)

[Chapter 5: Business and Finance 69](#_Toc380325324)

[Computers in Accounting 69](#_Toc380325326)

[Dealing With Dates 75](#_Toc380325327)

[Checking the Validity of Data Entry 81](#_Toc380325328)

[Conclusion 84](#_Toc380325329)

[Chapter 6: Filing Systems 85](#_Toc380325330)

[The User and the Filing System 86](#_Toc380325332)

[Tape Disc Compatibility 86](#_Toc380325333)

[Tape Files 87](#_Toc380325334)

[Disc Filing 88](#_Toc380325335)

[User Disc Drive Selection 88](#_Toc380325336)

[User Entry of Full File Specifications 90](#_Toc380325337)

[User File Name and Directory Entry on the DFS 90](#_Toc380325338)

[User Supplied Filenames and Directories with the ADFS 91](#_Toc380325339)

[Friendliness in User File Name Entry 92](#_Toc380325340)

[Alternatives to User Input of Data Filenames 94](#_Toc380325341)

[User Input of Filing and Operating System Commands – Star Commands 97](#_Toc380325342)

[Random Access Disc Filing 99](#_Toc380325343)

[Manipulating The Contents Of Random Access Files 102](#_Toc380325344)

[Auto Booting 104](#_Toc380325345)

[The CP/M Filing System on the BBC with Z80 Second Processor 105](#_Toc380325346)

[Filing System Commands under CP/M 106](#_Toc380325347)

[Filing System Errors Under CP/M 107](#_Toc380325348)

[Program Discs and Auto Boot under CP/M 107](#_Toc380325349)

[Conclusion 108](#_Toc380325350)

[Chapter 7: On Compatibility 109](#_Toc380325351)

[Operating System Differences 109](#_Toc380325353)

[The BBC and the Electron 109](#_Toc380325354)

[The Model B and the B Plus 110](#_Toc380325355)

[BASIC 1, BASIC 2 and HI-BASIC 110](#_Toc380325356)

[Differences Between BASICs 1 and 2 111](#_Toc380325357)

[The INSTR Bug 111](#_Toc380325358)

[ON…GOTO or GOSUB and ELSE 111](#_Toc380325359)

[Separators in INPUT 112](#_Toc380325360)

[OPENIN and OPENUP 112](#_Toc380325361)

[Number Formatting With @% 112](#_Toc380325362)

[OSCLI 112](#_Toc380325363)

[Assembler Differences 113](#_Toc380325364)

[General Points 113](#_Toc380325365)

[Z80 Basic 113](#_Toc380325366)

[Compatibility with the 6502 Second Processor 114](#_Toc380325367)

[Compatibility with the Z80 Second Processor 116](#_Toc380325368)

[Conclusions 116](#_Toc380325369)

[Chapter 8: The Art of Concealment 117](#_Toc380325370)

[Preventing 'Breaking Into' Programs 117](#_Toc380325372)

[Prevention of Listing 118](#_Toc380325373)

[Hiding a Disc Catalogue 120](#_Toc380325374)

[Chapter 9: Documentation of Programs 122](#_Toc380325375)

[Documentation for Users 122](#_Toc380325377)

[Documentation for Programmers 123](#_Toc380325378)

[Appendix 1: Function Key Strip Maker 125](#_Toc380325379)

[Instructions and User Notes 125](#_Toc380325381)

[Program Documentation for Maintenance 127](#_Toc380325382)

[Appendix 2: Some Useful Addresses 136](#_Toc380325383)

# Chapter 1

# Controlling the Program

There are two types of input that may be required of someone who is using a piece of software: data and instructions. By instructions I mean that if the program is capable of doing several things with the data, the user will have to choose an option to suit his purpose. Data will be required by virtually every piece of software and only very simple programs will not need instructions. This chapter deals with ways of giving the instructions, leaving data input for the next chapter.

You can let your customers give instructions to the program in two main ways, there are known as 'command driving' and 'menu driving'. They represent two quite different philosophical approaches to the writing of interactive software and the difference between them is perhaps best illustrated by an example. Suppose we have written a database, which is organized like an electronic card index, and we want to include the facility to delete a sequence of 'cards'. In the menu drive approach we would have a front page to the database showing the list of facilities provided, one of which would be labelled something like:

3. DELETE CARDS.

Pressing key 3 selects this option and generates prompts to enter the number of the first card to be deleted and then the number of subsequent deletions required. The point is that a series of options and prompts is issued so that the user is led through the sequence of required input a step at a time. To a large extent the program is self-documenting and there is no need for constant reference to a separate manual.

A command drive sequence to achieve the same effect would require a single input such as:

DELETE 20+5

to delete cards 20 to 25 inclusive. This second method will obviously need the back up of a set of separate instructions describing the form and syntax of the various commands, but has the merit that once someone is familiar with the program they can very rapidly give the instructions that they want without having to work through a series of prompts.

In actual use the preference for command or menu driving is largely a personal one. I find that command driving is very convenient for a program which I use often, and so do not need to keep referring to the manual, and, conversely, a menu driven program is much easier for the occasional or novice user.

From a programmer's point of view the choice of driving technique may well be dictated by the nature of the program. Menu driving ideally requires the screen to be taken over by the menu lists at various times, so does not particularly lend itself to software which itself requires the screen, such as drawing packages. This can be overcome by not actually displaying the menu when selections are required but still allowing response to single key presses. The menu still has to be provided in some form, of course, usually in separate, written, instructions and so many of the disadvantages of command driving are introduced. The BBC actually offers the possibility of a half-way house in that if the choices are selected by the red function keys the menu can be supplied as a paper strip which fits under the transparent retainer at the top of the keyboard. Much commercial software uses this approach and I shall be dealing with it later in the chapter.

The main disadvantage with menu driving is that the menus themselves require a rather large amount of storage and so the method is rarely found on really complex programs. The bulk of microcomputer software is still, nonetheless, menu driven, since there can be no question that it provides the most 'friendly' approach.

### Menu Driving

Listing 1.1 shows an example of the main control loop in a typical menu driven program, in this case a database, and Figure 1.1 depicts the screen display from which the user makes his selection.

The actual control structure is quite simple, being contained within the REPEAT…UNTIL loop between lines 20 and 270. The procedure to be called by the menu selection is determined by the contents of the string variable A$ which is input at line 140 with the GET$ function. Lines 210 to 260 send control into the appropriate subroutines. When a return occurs, say from PROCsearch at line 240 with A$ still equal to "4", execution will drop straight through to the UNTIL at line 270 and so go back to line 20 to re-display the menu. Notice that it is vital that A$ is not modified by any of the procedures which are called or the whole system may break down. What would happen, for example, if A$ was set equal to "6" in PROCsearch? The control mechanism is quite simple but what are the pitfalls? What mistakes or difficulties could arise whilst using the routine and how have I allowed for them?

Line 150 ensures that there is no response except to keys which correspond to menu options, but what happens if the person using the software inadvertently presses shift when trying to select from the menu or, rather more likely, accidentally engages the 'shift lock'? Now, for example, the '1' key will not give the expected character but will give an exclamation mark and if this is not recognized as a valid selection the program will appear to seize up. The second part of line 140 covers this eventuality by using the fact that the numerical keys all return either the number, of in shift mode, a character with an ASCII code sixteen higher. You should always ensure when offering menu selections that all states of the shift and caps keys are catered for.

#### Listing 1.1

10 MODE7

20 REPEAT CLS

30 PRINT''TAB(12)"\*MAIN MENU\*"

40 PRINTTAB(5)"1. ENTER NEW DATA."'

50 PRINTTAB(5)"2. VIEW EXISTING DATA."'

60 PRINTTAB(5)"3. DELETE CARDS."'

70 PRINTTAB(5)"4. SEARCH DATA."'

80 PRINTTAB(5)"5. SORT DATA."'

90 PRINTTAB(5)"6. SAVE DATA TO TAPE/DISC."'

100 PRINTTAB(5)"7. END PROGRAM."'

110 PRINTTAB(5,20("Please enter a choice "+CHR$(8);

120 VDU23,1,1;0;0;0;

130 REPEAT \*FX15,1

140 A$=GET$:IF A$<"1" A$=CHR$(ASC(A$)+16)

150 UNTIL INSTR("1234567",A$)

160 PRINT A$;:VDU23,1,0;0;0;0;

170 REPEAT \*FX15,1

180 B$=GET$

190 UNTIL B$=CHR$(13) OR B$=CHR$(127)

200 IF B$=CHR$(127) THEN 110

210 IF A$="1" PROCenter

220 IF A$="2" PROCview

230 IF A$="3" PROCdelete

240 IF A$="4" PROCsearch

250 IF A$="5" PROCsort

260 IF A$="6" PROCsave

270 UNTIL A$="7"

280 MODE 7

290 END

\*MAIN MENU\*

1. ENTER NEW DATA

2. VIEW EXISTING DATA

3. DELETE CARDS

4. SEARCH DATA

5. SORT DATA

6. SAVE DATA TO TAPE/DISC

7. END PROGRAM

Please enter a choice \_\_

##### **Figure 1.1** Output from listing 1.1

The letter keys, remember, can give the upper or lower care equivalents. Always test for both to make sure that you convert input to upper case. A simple way to do this is to use the fact that if the ASCII code of a lower case letter is ANDed with the hexadecimal number &DF it will give the code for the corresponding upper case letter whereas the same operation on the upper case ASCII code leaves it unchanged. The following line, therefore, will always return the upper case letter into A$, irrespective of the state of the shift and caps keys.

10 A%=GET AND &DF:A$=CHR$(A%)

The next problem to be tackled is the question of what happens if the user makes the wrong selection from the menu, it would not really be very fair to leave him stuck with a routine that he never actually wanted. For this reason it is not normally satisfactory to call the subroutine immediately the appropriate key is detected unless there is an easy escape from a wrong selection once you are into it. I will show you how to arrange this later but for the moment I am going to adopt the approach of making it very hard for the user to get into this position in the first place. In Listing 1.1 I have arranged for the number of the option called to be displayed, in line 160, and in order actually to activate the call the return key must be pressed. Alternatively, the delete key will cancel the selection and offer the choice again. The loop between lines 170 and 190 checks for the appropriate key and line 200 returns execution to the initial request for a selection if the delete key was pressed, so anyone using the program will see the selection that they have made and have a chance to correct it if necessary. The printing of the selection also serves another purpose by ensuring that when a valid key is pressed something happens immediately. This is an important general point to note about taking input. Suppose I press a key and nothing obvious happens? Perhaps the computer is, indeed, hard at work but what am I to think? Has it crashed? Have I not, in fact, pressed a valid key? Even if you only clear the screen or generate a bleep with VDU7 do let the user know when an input has been recognized and has started something.

Another thing that I have done in this piece of program is to have a flashing cursor when input is required, line 120 ensures it is switched on. The presence of a cursor is a useful indication to the user that he is to enter something.

The only thing that remains to be explained in Listing 1.1 is the use of \*FX15,1 in lines 130 and 170. This is a command to flush the keyboard buffer and it is most important that it be included. The GET$ function does not actually test for a key press itself; strictly speaking it looks at the input buffer and it has no way of knowing how long a character has been in there. It may not be a recent key press at all. Suppose that you had accidentally pressed the key for '1' whilst in PROCdelete. Unless we flush the buffer the '1' will still be there when line 140 is executed after restoring the menu and it will pop up on the screen as though we had made a selection. This would be irritating and confusing. Always flush the buffer before using any function which is required to detect a recent key press i.e. GET, GET$, INKEY$ or INKEY (with a positive argument). INKEY with a negative argument can be used to test for a specific key, rather than the buffer, but it cannot readily be used to check the whole keyboard and reveal which key was pressed.

Before we leave the example in Listing 1.1 you might like to consider a couple of refinements. The first is particularly easy to apply when you are working with mode 7 on the BBC and that is to have the selected menu item highlighted before the return key is pressed. If you arrange for a teletext control character to be printed in front of the selection, you can make it flash or change colour. To do this make the following modification to Listing 1.1:

Add the lines:

105 FOR I%=2 TO 14 STEP 2:PRINTTAB(4,3+I%)"":NEXT

165 PRINTTAB(4,3+2\*VAL(A$))CHR$(136)

Change line 200 to:

200 IF B%=CHR$(127) THEN 105

Line 165 print the 'flash' teletext code before the selected entry. If you want to have a colour change instead us an appropriate code, e.g. CHR$(131) to turn the entry yellow. Line 105 is necessary to remove any control characters if the delete key is pressed.

The second refinement is not so much a help as a tidying up measure. As the program stands the cursor keys are active and if one is pressed the copy and ordinary cursors will separate, making a rather messy screen. To prevent this you could make the following modifications:-

Change line 120 to:

120 VDU23,1,1;0;0;0;:\*FX4,1

Add the line:

195 \*FX4,0

The first FX call shuts off the cursor keys, or at least makes them return ASCII codes rather than performing their normal functions, and the second call, in line 195 restores them to normal use.

The technique of menu selection illustrated by Listing 1.1 is the simplest, using a different key for each item, but it is not the only possible method. If you want your software to look a little different you could use a 'moving pointer'. The idea is that a marker on the menu is moved by, say the space bar, until it is opposite the required item on the menu and then the selection is made by pressing another key, usually return. Listing 1.2 shows how this can be done.

Down to line 130 everything is much the same as in Listing 1.1, the moving pointer work is all done in the REPEAT…UNTIL loop between lines 150 and 200. At line 160 we print the pointer opposite the first menu entry, I have made the fairly obvious choice of the '>' character. Line 170 tests for the space bar or the return key. When one of these keys is pressed the first job is to delete the pointer character, at line 180, and then, if the key pressed was the space bar the position of the pointer is updated so that when it is re-displayed it will appear opposite the next menu entry. The second IF statement in line 190 returns the pointer to the first menu item if space is pressed when the pointer is at the end of the list. When the return key is pressed execution leaves the loop and moves to line 210, calling the subroutine indicated by the position of the market as determined by the value of A%.

Obviously if you want to use the moving pointer technique you will have to tailor it to your own menu layout but it is a particularly clear means of providing menu selection. In teletext mode you may find it a useful variation to make the 'pointer' a teletext control code which, although itself invisible, will make the selected item flash or change colour. A general word of caution here. If you are using a colour monitor do bear in mind that others may want to run your software in monochrome. Red, for example, is very difficult to see on a green monitor. Always try to check colour effects in monochrome if you can.

The use of menus does not have to be confined to the selection of execution options, there is no reason why they cannot be used to input numerical values or strings from a restricted list. Listing 1.3 shows this in operation with a refinement on the moving pointer technique which allows switching between two menus with the TAB key on the BBC or the equivalent CTRL-I on the Electron. The object here is to allow the user to set serial port receive and send rates. By the way, if you have a serial printer do press CTRL-BREAK after running the program to reset the default rates of your printer may not work properly.

Two moving pointers are used; they are printed at line 160, both being moved by the space bar. The TAB key is used to toggle between the two pointers, only one being active at any one time. The heading of the active menu is made to flash, at line 150. You can thus set up the situation that you want as regards send and receive rates and, when you are quite satisfied with the values, accept them by pressing the return key. This technique of operating multiple menus is rather like filling in a form, in that you can make sure you have selected all the correct options before you 'send' it. We shall be returning to look at some other examples of this very useful type of input in Chapter 2.

#### Listing 1.2

10 MODE7

20 REPEAT CLS

30 VDU23,1,0;0;0;0;

40 PRINT''TAB(12)"\*MAIN MENU\*"''

50 PRINTTAB(5)"1. ENTER NEW DATA."'

60 PRINTTAB(5)"2. VIEW EXISTING DATA."'

70 PRINTTAB(5)"3. DELETE CARDS."'

80 PRINTTAB(5)"4. SEARCH DATA."'

90 PRINTTAB(5)"5. SORT DATA."'

100 PRINTTAB(5)"6. SAVE DATA TO TAPE/DISC."'

110 PRINTTAB(5)"7. END PROGRAM."'

120 PRINTTAB(2,20)"Use the space bar to make your"

130 PRINTTAB(2,21)"selection and then press RETURN"

140 A%=5:\*FX15,1

150 REPEAT

160 PRINTTAB(4,A%)">"

170 REPEAT A$=GET$:UNTIL INSTR(" "+CHR$(13),A$)

180 PRINTTAB(4,A%)" "

190 IF A$=" " A%=A%+2:IF A%>17 A%=5

200 UNTIL A$=CHR$(13)

210 IF A%=5 PROCenter

220 IF A%=7 PROCview

230 IF A%=9 PROCdelete

240 IF A%=11 PROCsearch

250 IF A%=13 PROCsort

260 IF A%=15 PROCsave

270 UNTIL A%=17

280 MODE 7

290 END

#### Listing 1.3

10 MODE 7

20 @%=00000004:REM\*PRINT NUMBERS IN FIELD 4 DIGITS WIDE\*

30 VDU23,1,0;0;0;0;

40 PRINTTAB(4,3)"\*SET RS423 PORT BAUD RATES\*"

50 PRINTTAB(6,5)"SEND";TAB(22,5)"RECEIVE"'

60 RESTORE

70 FOR I%=1 TO 7

80 READ rate%

90 PRINTTAB(6)rate% TAB(23)rate%

100 NEXT

110 PRINTTAB(2,20)"Press Space Bar to change Baud rate"

120 PRINT"TAB to change menu and RETURN to Accept"

130 sendspeed%=7:recspeed%=7:menu$="SEND"

140 REPEAT

150 IF menu$="SEND" PRINTTAB(5,5);CHR$(136);TAB(21,5)CHR$(137) ELSE PRINTTAB(5,5);CHR$(137);TAB(21,5);CHR$(136)

160 PRINTTAB(10,sendspeed%);"<";TAB(27,recspeed%);"<"

170 \*FX15,1

180 REPEAT A$=GET$:UNTIL INSTR(" "+CHR$(9)+CHR$(13),A$)

190 PRINTTAB(10,sendspeed%);" ";TAB(27,recspeed%);" "

200 IF A$=CHR$(9):PROCchangemenu

210 IF A$=" " PROCsetspeed

220 UNTIL A$=CHR$(13)

230 OSCLI("FX8 "+STR$(sendspeed%-6))

240 OSCLI("FX7 "+STR$(recspeed%-6))

250 CLS

260 PRINTTAB(0,10)"Sending at ";FNdecode(sendspeed%);" Baud"'

270 PRINT"Receiving at ";FNdecode(recspeed%);" Baud"

280 END

290

300 DEF FNdecode(speed%):REM\*CONVERT SCREEN POSITION TO BAUD RATE\*

310 LOCAL i%,bauderate%

320 speed%=speed%-6

330 RESTORE

340 FOR i%=1 TO speed%

350 READ baudrate%

360 NEXT

370 =baudrate%

380

390 DEFPROCsetspeed

400 IF menu$="SEND" sendspeed%=FNinc(sendspeed%)

410 IF menu$="REC" recspeed%=FNinc(recspeed%)

420 ENDPROC

430

440 DEFPROCchangemenu

450 IF menu$="SEND" menu$="REC" ELSE menu$="SEND"

460 ENDPROC

470

480 DEF FNinc(speed%)

490 speed%=speed%+1

500 IF speed%>13 speed%=7

510 =speed%

520

530 REM\*POSSIBLE BAUD RATES\*

540 DATA 75,150,300,1200,2400,4800,9600

In Listing 1.3 the actual baud rates are set by lines 230 and 240 which send the appropriate FX calls. These lines will not work in BASIC 1, since OSCLI is not implemented. I shall be explaining how to get round this problem in Chapter 6.

The techniques that I have dealt with in this section cover most of the menu methods that are commonly found. I would just like to conclude with a couple of further points. Firstly, try to keep the number of menus in a program down to a minimum. It is very wearing for a user to be led through a long trial of menu after menu, especially if he is familiar with the program. It is better to put as many options as possible on each menu rather than to proliferate menus. For example, if in a database you want to offer the option of a descending or an ascending sort it is better to have these appear as two explicit options on the main menu rather then have a sort option followed by a prompt for ascending or descending.

The second point concerns the titling of menus. Do try to give each menu a clear and descriptive title so that the users know exactly where they are in the program. The title will also be useful when you are writing any instructions to go with the software since it will clearly identify each menu.

### Command Driving

You are already very familiar with a command driven program, the BASIC interpreter in your machine. If you are to produce a command driven program of your own it will need the same kind of facilities as the BASIC with regard to its ability to decode commands and, most importantly, it must be able to deal with erroneous input and to produce helpful error messages. There are no general rules for constructing a command driven module but routines will always be required with the following purposes:

1. An input routine to accept the commands. This will also prepare the commands for passing to the subroutine which is going to decode them. The kind of thing that may be necessary is the removal of leading and trailing spaces from the input. A user cannot explicitly see these extra spaces on the screen so if their presence leads to an unrecognized command error it can be most confusing. It is also very common, at this stage, to convert any letters in the command to ensure that they are in upper case so that any mixture of small and capital letters is acceptable as input. The BBC BASIC does not actually do this since it wants lower case letters to be recognized as distinct for use in variable names, but many utility ROMs allow lower or upper case input as equivalent.

2. A list of allowed commands, often associated with information as to whether they require parameters and the number and type of any such parameters.

3. A routine which decodes the input and extracts the command for comparison with the list of allowed commands. An error message, such as Bad Command, should be sent if the input does not contain a recognized word. If the command is recognized the routine goes on to extract any parameters that are needed and passes the instruction and the parameters back to the main program for appropriate action. There will usually need to be a second level of error trapping here to intercept problems such as missing or out of range parameters.

4. A section which acts as appropriate on the decoded commands.

Fairly obviously the key to a good command driven problem lies in the efficiency of the error trapping routines and in how helpful the error messages are to users.

Listing 1.4 is a very simple drawing program which lets you move the graphics cursor around the mode 1 screen, with the option of leaving a trail, PENDOWN, or not, PENUP. There is also the possibility of putting text on the screen at the graphics cursor with a command such as PRINT THIS IS WHAT WOULD BE PRINTED so the print command needs a text string as a parameter. The cursor is moved in the obvious ways by the commands RIGHT, LEFT, UP and DOWN, each requiring as a single parameter the number of graphics positions to be moved. You can only draw rectangles and squares of course, but this is intended to be an example of command driving, not a full graphics package.

The screen is divided into two windows: a three line text window at the bottom for the command input and error messages, and the major part of the screen is the graphics window on which pictures are drawn. In this type of application BBC BASIC's window-defining facilities really come into their own. The program has been well provided with REM statements so I do not propose to go through it line by line but just to cover the important feature of the command interpreter.

The input is taken at line 130. I deliberately used INPUT LINE rather than INPUT so that all the keyboard characters will be accepted without confusion. I will be dealing with the problems of INPUT in Chapter 2. One result of using INPUT LINE is that any leading or trailing spaces will be taken in, so these are removed in lines 140 and 150. Line 160 does the conversion of any lower case letters to capitals. At line 170 we now check to see if we have anything left and if so the input is passed to the decoder routine at line 620. The syntax rules for input require that the command appear as the first part of the input line and then any parameters follow separated only by spaces so the loop between lines 640 and 680 reads along the line and extracts the letters into temp$ until it finds a non-alphabetic character. The variable temp$ now contains the command which is checked against the valid commands list in the loop at lines 710 to 750. I have put the allowed commands into the DATA statement at line 1400 so it is only necessary to read along the list until the command is recognized or the end of the list is reached.

The end of the list contains the meaningless mnemonic ZZZ and if this is reached the command must be an unrecognized one and an error message is sent, line 770. The data statement contains two 'attributes' after each command. The first is either 0 or -1, which are recognized by the BASIC as FALSE and TRUE respectively. Their significance here is to indicate whether the command requires a parameter (TRUE) or not (FALSE). The second attribute is a letter which expresses which type of parameter is expected. If no parameter is needed any letter will do, I have used 'X', a numerical parameter is indicated by 'N' and text parameters by 'T'. The point about this second letter is that a command such as PRINT will need its parameter dealt with as a literal string of characters and just printed, whereas a numerical parameter will have to be checked for validity. The great beauty of this DATA statement approach is that new commands are very easy to add; you simply put them into the DATA statement with the correct attributes and the error checking takes them on board. In addition, of course, you have to include the routine that the new command calls.

#### Listing 1.4

10 MODE 1

20 VDU19,1,4,0,0,0:REM\*RED>BLUE\*

30 VDU 28,0,31,39,29:REM\*TEXT WINDOW\*

40 VDU 24,0;100;1279;1023;:REM\*GRAPHICS WINDOW\*

50 GCOL0,129:REM\*BLUE GRAPHICS B/GD.\*

60 VDU 29,0;100;:REM\*SHIFT GRAPHICS ORIGIN\*

70 CLG:MOVE 0,0

80 plotopt%=4:csup%=0:csrt%=0

90 REPEAT CLS

100 REPEAT valid%=TRUE

110 CLS:IF plotopt%=4 PRINTTAB(38,2)"U"; ELSE PRINTTAB(38,2)"D";

120 VDU 30

130 INPUTLINE"Command "comm$

140 comm$=FNstriplead(comm$)

150 comm$=FNstriptrail(comm$)

160 comm$=FNconvert(comm$)

170 IF comm$="" CLS:GOTO 110 ELSE PROCdecode(comm$)

180 UNTIL valid%

190 IF command$="HELP" PROChelp

200 IF command$="HOME" csup%=0:csrt%=0:plotopt%=4

210 IF command$="UP" csup%=FNinc(VAL(param$))

220 IF command$="DOWN" csup%=FNdec(VAL(param$))

230 IF command$="RIGHT" csup%=FNrt(VAL(param$))

240 IF command$="LEFT" csup%=FNlt(VAL(param$))

250 IF command$="CLEAR" THEN IF NOT FNsure command$=""

260 IF command$="CLEAR" CLG:csrt%=0:csup%=0:plotopt%=4

270 IF command$="PENUP" PLOT 71,csrt%,csup%:plotopt%=4

280 IF command$="PENDOWN" plotopt%=5

290 IF command$="PRINT" plotopt%=4:VDU5:PRINT param$;:VDU4

300 PLOTplotopt%,csrt%,csup%

310 IF command$="END" THEN IF NOT FNsure command$=""

320 UNTIL command$="END"

330 MODE 7

340 END

350

360 DEF FNconvert(t$):REM\*TO HIGH CASE\*

370 LOCAL i%,temp$,char$

380 FOR i%=1 TO LEN(t$)

390 char$=MID$(t$,i%,1)

400 IF char%>="a" AND char$<="z" char$=CHR$(ASC(char$) AND &DF)

410 temp$=temp$+char$

420 NEXT

430 =temp$

440

450 DEF FNstriplead(t$)

460 REM\*STRIP LEADING SPACES\*

470 IF LEFT$(t$,1)<>" " =t$

480 REPEAT

490 t$=RIGHT$(t$,LEN(t$)-1)

500 UNTIL LEFT$(t$,1)<>" "

510 =t$

520

530 DEF FNstriptrail(t$)

540 REM\*STRIP TRAILING SPACES\*

550 IF RIGHT$(t$,1)<>" " =t$

560 REPEAT

570 t$=LEFT$(t$,LEN(t$)-1)

580 UNTIL RIGHT$(t$,1)<>" "

590 =t$

600

610 REM\*\*\*\*\*\*\*\*DECODE THE COMMAND LINE\*\*\*\*\*\*\*\*

620 DEFPROCdecode(comm$)

630 LOCAL i%,temp$,char$,param\_expected%,paramtype$:param$=""

640 FOR i%=1 TO LEN(comm$)

650 char$=MID$(comm$,i%,1)

660 REM\*EXIT LOOP ON FIRST NON ALPHABETIC CHARACTER\*

670 IF char$<"A" OR char$>"Z" THEN i%=1000 ELSE temp$=temp$+char$

680 NEXT

690 command$=temp$

700 RESTORE

710 REPEAT

720 REM\*CHECK COMMAND AGAINST VALID LIST\*

730 READ temp$,param\_expected%,paramtype$

740 REM\*EXIT IF COMMAND FOUND OR END OF LIST\*

750 UNTIL temp$=command$ OR temp$="ZZZ"

760 REM\*ERROR IF UNRECOGNISED COMMAND\*

770 IF temp$="ZZZ" PROCerror("Bad Command"):ENDPROC

780 REM\*CHECK FOR UNWANTED PARAMETER\*

790 IF NOT param\_expected% AND comm$<>command$ PROCerror("No Parameter Needed"):ENDPROC

800 REM\*CHECK FOR MISSING PARAMETER\*

810 IF param\_expected% AND comm$=command$ PROCerror("Missing Parameter"):ENDPROC

820 REM\*EXIT IF NO PARAMETER REQUIRED\*

830 IF NOT param\_expected% ENDPROC

840 REM\*EXTRACT PARAMETER\*

850 param$=FNstriplead(RIGHT$(comm$,LEN(comm$)-LEN(command$)))

860 REM\*EXIT IF PARAMETER LITERAL STRING\*

870 IF paramtype$="T" ENDPROC

880 REM\*EXIT IF INVALID CHARACTER IN NUMERIC PARAMETER\*

890 FOR i%=1 TO LEN(param$)

900 char$=MID$(param$,i%,1)

910 IF INSTR("0123456789",char$)=0 PROCerror("Bad Parameter"):i%=1000

920 NEXT

930 ENDPROC:REM\*WITH COMMAND IN command$ AND PARAMETER IN param$\*

940

950 REM\*SEND ERROR MESSAGE AND WAIT\*

960 DEFPROCerror(e$)

970 valid%=FALSE

980 PRINT e$';"Press Space Bar to Continue";:REPEAT UNTIL GET$=" "

990 ENDPROC

1000

1010 DEF FN sure

1020 LOCAL A$

1030 PRINT"\*\*\*ARE YOU SURE?\*\*\*Press Y or N\*\*\*"+CHR$(7);:\*FX15,1

1040 A$=GET$

1050 IF INSTR("Yy",A$) THEN =TRUE ELSE =FALSE

1060

1070 REM\*\*\*\*\*CALCULATE NEW GRAPHICS CURSOR POSITIONS\*\*\*

1080 DEF FNinc(n%)

1090 LOCAL t%

1100 t%=csup%+n%

1110 IF t%>923 t%=923

1120 =t%

1130 DEF FNdec(n%)

1140 LOCALt%

1150 t%=csup%-n%

1160 IF t%<0 t%=0

1170 =t%

1180 DEF FNrt(n%)

1190 LOCALt%

1200 t%=csrt%+n%

1210 IF t%>1279 t%=1279

1220 =t%

1230 DEF FNlt(n%)

1240 LOCAL t%

1250 t%=csrt%=n%

1260 IF t%<0 t%=0

1270 =t%

1280

1290 DEFPROChelp

1300 LOCAL comm$,dummy%,dummy$

1310 RESTORE

1320 CLS

1330 REPEAT READ comm$,p%,dummy$

1340 IF comm$<>"ZZZ" THEN PRINT " ";comm$;

1350 UNTIL comm$="ZZZ"

1360 PRINT'"\*\*SPACE TO CONTINUE\*\*;:REPEAT UNTIL GET$=" "

1370 ENDPROC

1380

1390 REM\*VALID COMMAND LIST\*

1400 DATA HELP,0,X,HOME,0,X,CLEAR,0,X,END,0,X,PENUP,0,X,PENDOWN,0,X,RIGHT,-1,N,LEFT,-1,N,UP,-1,N,DOWN,-1,N,PRINT,-1,T,ZZZ,0,X

Line 790 deals with the error case where a parameter has been included in a command which does not need one. We could just make this routine discard the parameter and obey the command, but this might be dangerous since the user might not actually mean to use that command. There may be a purely fortuitous correspondence between the first letters of the input and an acceptable command. Line 810 deals with the converse case when a parameter is missing when required.

If we have decoded a command that does not have, nor require, a parameter then the error checks are complete and we can leave the routine at line 830, otherwise the parameter has to be extracted from the input. Line 850 does this and also strips any leading spaces from the parameter. Remember we decided to allow spaces between command and parameter as an option. If the parameter is of the type that is to be treated as a literal string, such as that of 'PRINT', we are finished and the routine is exited at line 870, otherwise we have a numerical parameter and this is checked, in the loop in lines 890 to 920 for any non-numerical characters.

Any error in the input sets the variable valid% to FALSE and sends a message. This is achieved in the procedure at line 960. The command which has been rejected remains on the screen with the error message until the space bar is pressed. It is important to have this sort of arrangement if possible, an error message such as 'Bad Parameter' is not very helpful if your input is not available for inspection so that you can see where you went wrong. If valid% has become false execution cannot leave the loop at line 180 and so you have to make another stab at the input but if all is well line 190 is entered to start looking for what the command is meant to do.

I have considered a couple of refinements to the program which are always worth considering. The first is a common feature of command driven software, the 'HELP' facility. This prints out all the allowed commands by calling the procedure at line 1290. Notice again how the use of a DATA statement for the command list is very convenient here; any new command will automatically be included in the help list. If you really want to go to town you could also include an indication of whether the command takes a parameter.

The second refinement relates to potentially disastrous commands, in the case of this program END and CLEAR, which end the program and clear the graphics screen respectively. You really would not want to do either of these by accident, so a second chance is a friendly gesture. When either of these commands is detected FNsure, at line 1010, is called to achieve the safety net.

You might well have decided by now that command driving is a lot of trouble to implement, and this is probably true for a program which only a small number of facilities, but the error checking and input routines on Listing 1.4 would serve just as well for a program with a vastly increased number of commands and, moreover, they would be easy to add. Just put them into the command list and then add the code that they are to implement. You do not have to include extra entries on menus which could well mean completely reorganizing the display screens, and the only increase in memory usage would be that actually required to implement the command. You will not be adding lots of print statements to menus, which consume a lot of memory.

### Using the Function Keys on the BBC

The BBC micro has a conveniently placed set of ten function keys, the 'red' keys, which can be operated in a variety of ways. Normally they are set up so that they can be programmed to return strings of characters, that is when pressed alone. When pressed with 'SHIFT' they give the extended ASCII codes 128 to 137, and with 'CTRL', codes 144 to 153. The un-shifted function keys can also be made to abandon their string storage role and to return ASCII codes by sending the call \*FX 225,n, where n is the code required from the key f0. Key f1 will return n+1 and so on for the rest, up to n+9. You have, therefore, the possibility of using the function keys to access a menu with up to 30 entries, the really convenient point being that they can be labelled directly with the menu choices by means of a paper strip fitted under the transparent retainer at the top of the keyboard.

Apart from this method of displaying the menu choices, which does not consume memory or require the screen, of course, the software can operate in the same manner as a conventional menu driven package. The method is a useful alternative to command driving for programs which need the screen, for example the drawing package in Listing 1.4. In Appendix 1 I have included a program for producing professional looking key-strips on Epson compatible printers and it is easily modified for any printer capable of condensed printing mode printing at eight lines to the inch.

The ability of the red keys to store strings can be exploited in command driven programs to make input of the commonly used commands easier. For example we might as well include in the drawing package in Listing 1.4 a series of key definitions such as:-

\*KEY 0 PRINT

\*KEY 1 PENUP|M

\*KEY 2 PENDOWN|M

\*KEY 3 UP

and so on. Pressing the appropriate function key will now be exactly like typing the input. The |M is equivalent to pressing 'RETURN' and must not appear after commands that take a parameter. The user will need to supply the parameter and then press 'RETURN' in such cases.

An alternative approach, to pre-programming the function keys, is to provide facilities for users to do it themselves, so that the easily accessible commands can be the ones that they find the most useful. If you do this your error trapping on the input must be very effective, since programming of the function keys requires some degree of access to the operating system which copes with the start commands. I will be dealing with methods of letting users have such access without the potential for disaster in chapter 6.

# Chapter 2.

# Input of Data from the Keyboard

Probably the most important part of any program from the point of view of crash protection and user friendliness is the construction of good data input routines. These have to serve two functions. Firstly they must protect the program from any input with which is it not designed to cope and secondly the user should not be allowed to enter anything that is clearly not sensible within the context of what the input represents in the real world. If a date is to be entered, for example, there will be certain obvious rules relating to the number of days in each month and so on. It will be most helpful to the user if the program checks that the input date is not a clearly ridiculous one.

BBC BASIC has two means of taking data from the keyboard, INPUT and INPUTLINE. The main distinction between the two is that INPUTLINE reads the entire line, up to where RETURN was pressed, into a variable whereas INPUT accepts a line of values to be read into any number of variables provided they are each separated by a comma. The following line reads in two numbers to the variables n1 and n2.

10 INPUT"Enter two numbers separated by a comma "n1,n2

It may seem perfectly satisfactory, but I do not recommend this multiple input method for serious software since it can lead to confusion. The program expects to find two numbers on the same line but, if a user only enters one and then presses RETURN it will prompt for a second number on the next line, with the default prompt, a question mark. This is not really very helpful in telling a user what has gone wrong and what is now expected of him. It is far better to prompt for the numbers with separate, explicit, prompts. If you replace INPUT in the above example with INPUTLINE it will not work as you might expect. The whole line, including the comma, will be accepted as n1 and decoded as a number until the comma is reached and then the rest of the line will be discarded. A separate prompt will then be issued for the second number. You cannot take multiple input with INPUTLINE; it regards the comma as a normal character not a delimiter. Exactly the same situation would apply if we were trying to read in two strings rather than two numbers.

Since the comma has a special function when using INPUT you cannot read text input containing a comma into a string variable with this command. What will happen is that the line will be accepted up to the comma and the remainder discarded. Your input will be crudely chopped off. The same will actually apply if a comma should be accidentally entered as part of a number, but this is not anything special as any non-numeric character will have the same effect.

A problem also arrives with INPUT if quotation marks appear in the entered line, especially if they are the first character. Quotation marks, like the comma, have a special significance in BASIC in that they delimit strings.

If you are using INPUT and someone enters one set of quotes as the first character of their input and does not terminate the line with a second set, then BBC BASIC will produce an error with the message Missing ". The reason for this is that it is looking for the input to be totally enclosed within quotes and it cannot find the second occurrence. Single quotes as other than the first character of input are not so disastrous. If they occur in a number their effect is just like that of any non-numeric character, the number is read up to the point where they occur and then chopped. In string input they will be accepted as a normal character.

If you employ the INPUT command at all you leave the possibility that the program can be crashed by someone misusing quotes in this way. The best method of avoiding the problem is always to use INPUTLINE, which recognizes quotes and commas as normal characters, devoid of their special significance in BASIC. Alternatively, you can make sure that your general error trapping will allow the program to survive the Missing " error by careful use of ON ERROR.

The other difference between INPUT and INPUTLINE is that the latter will preserve leading and trailing spaces in the line, whereas the former strips the leading spaces. As regards reading numbers there is no practical distinction but in text input you may well need to decide whether you want the leading spaces. In a word processor, for example, you certainly would.

Generally speaking INPUTLINE provides the securer input, and you can always arrange to strip off the leading spaces from strings if this is important. You will rapidly find, however, that if you want to have really helpful data input routines you must dispense with the pre-packaged BASIC facilities altogether in favour of specially written subroutines.

### Protecting Numerical Input

The obvious way to take numeric input from the keyboard is with a line such as:

10 INPUTLINE "Please enter a number "A

but there are several things that could go wrong here. Probably the most common mistake at the keyboard is to include a non-numeric character in a number; an extra space or the confusion between zero and the letter 'O' are the most likely examples. BBC BASIC is quite kind to this sort of mistake in that it does not produce an error message. What does happen is that the input up to the first non-numeric character is accepted and the rest of the line is ignored. So if you type '23.O45', accidentally keying in 'O' for zero, you will have actually input the number 23 and since no error is triggered you could go on oblivious. This kind of slip can cause real problems in number crunching programs but it can be anticipated and trapped. The way to do this is to take the numbers in as a string so that the characters can be validated before conversion to a number by the VAL function. Having said this, I do not recommend it if you need numbers with the highest possible precision, say for scientific purposes, since the VAL function can sometimes introduce rounding errors. In this case it is best to use direct numeric input and then re-display the numbers for scrutiny and correction if necessary.

If you want to prevent the entry of invalid numbers into a program there are two possible approaches, exemplified by the functions in Listings 2.1 and 2.2. In both of these the number is read as a string. In Listing 2.1 the input is accepted initially and then scrutinized so that an error message can be issued if there are any mistakes and a repeat requested. Listing 2.2 takes a different approach in that it examines each character of the input as it comes in and replaces it with a null string if it is not valid in the context of the characters that have gone before. To the user the key press appears to have been completely ignored.

In both listings I have set the following criteria:

1. Only numbers, decimal points, minus signs and E are valid characters. The latter to permit input in scientific notation.
2. Only one decimal point is permitted and it cannot be in the part of the number after the E, that is in the exponent if scientific notation is in use.
3. No more than two minus signs are allowed, and these can only be either the first character or the character immediately following the E. The exponent may be negative, of course.
4. Only one E is permitted and it cannot be the first character. This is not strictly wrong numerically but is required in practice, since the VAL function does not recognize a string where the first character is E as a number, and so will return zero.

The function in Listing 2.1 takes the string input and examines it, one character at a time to see if the criteria are all met. If so the value TRUE is returned otherwise the result is FALSE. The check is done by extracting each character with the MID$ function, at line 140, on repeated passes through the FOR…NEXT loop between lines 130 and 270. An error in the input results in the value of the loop counter being set to 1000. This has the dual function of causing an exit from the loop when line 270 is next executed and also serves as an error flag, since in the normal course of events i% would never become so large.

Listing 2.2 is an example of the construction of an input routine from scratch, that is not using INPUT or INPUTLINE. The characters are taken from the keyboard, one at a time, by GET$ in line 160 so that they can be checked for validity. Characters which pass the test are concatenated in the variable in$ which is displayed ready for input of the next character, at line 140. If a character is not valid it is replaced by an empty string before being added to in$ so it has no effect. To use the function you must pass, as parameters, the TAB positions on the screen from which you want to take the input, x% and y%, a user prompt, in prompt$, and a list of allowed characters. Carriage return and delete are added to this list automatically at line 110. To anyone using the routine it will appear very like a normal BASIC input sequence but only the numeric keys, E, the minus sign and decimal point will actually respond, and then only if the result makes sense. For example, once one decimal point has been entered that key will cease to work. DELETE works as normal and RETURN serves its usual function of terminating input, with what has to be a valid number.

#### Listing 2.1

10 MODE 7

20 REPEAT

30 REPEAT

40 INPUTLINE"Enter a number "number$

50 valid%=FNvalidate(number$,"0123456789.-E")

60 IF valid% PRINT"You entered ";VAL(number$)' ELSE PRINT"Bad

Number-try again"'

70 UNTIL valid%

80 UNTIL FALSE

90

100 DEF FNvalidate(number$,allowed$)

110 IF number$="" THEN =TRUE

120 LOCAL digit$,points%,expts%,i%

130 FOR i%=1 TO LEN(number%)

140 digit$=MID$(number$,i%,1)

150 REM\*TEST FOR VALID CHARACTERS\*

160 IF INSTR(allowed$,digit$)=0 i%=1000

170 REM\*TEST FOR MORE THAN ONE DECIMAL POINT\*

180 IF digit$="." points%=points%+1:IF points%>1 i%=1000

190 REM\*E AS FIRST CHARACTER OR WITH ZERO FOLLOWING NOT VALID\*

200 IF digit$="E" THEN IF i%=1 OR VAL(MID$(number$,i%+1))=0 i%=1000:GOTO 270:REM\*GOTO NOT NEEDED IN BASIC 2\*

210 REM\*TWO Es NOT ALLOWED\*

220 IF digit$="E" expts%=expts%+1:IF expts%>1 i%=1000

230 REM\*MINUS SIGN MUST BE FIRST CHARACTER OR FIRST AFTER E\*

240 IF digit$="-" AND i%<>1 THEN IF MID$(number$,i%-1,1)<>"E" i%=1000

250 REM\*NO DECIMAL POINTS IN EXPONENT ALLOWED\*

260 IF digit$="E" AND INSTR(MID$(number$,i%),".") i%=1000

270 NEXT

280 IF i%>LEN(number$)+1 VDU7:=FALSE ELSE =TRUE

#### Listing 2.2

10 MODE 7

20 FOR y%=0 TO 21 STEP 3

30 number=FNgetno(0,y%,"Please enter a number","0123456789.E-"

)

40 PRINT'"You entered ";number

50 NEXT

60 PRINT'" \*PRESS THE SPACE BAR TO CONTINUE\*";

70 REPEAT UNTIL GET$=" ":RUN

80

90 DEF FNgetno(x%,y%,prompt$,allowed$)

100 LOCAL a$,in$:in$=" "

110 allowed$=allowed$+CHR$(13)+CHR$(127)

120 REPEAT

130 REPEAT

140 PRINTTAB(x%,y%);SPC(40);TAB(x%,y%);prompt$+" ";in$;

150 \*FX15,1

160 a$=GET$

170 REM\*RESPOND ONLY TO VALID CHARACTERS\*

180 UNTIL INSTR(allowed$,a$)

190 REM\*ONLY ONE DECIMAL POINT NONE IN EXPONENT\*

200 IF a$="." THEN IF INSTR(in$,".")<>0 OR INSTR(in$,"E")<>0 a$=""

210 REM\*E CANNOT BE FIRST CHARACTER OR BE USED TWICE\*

220 IF a$="E" THEN IF INSTR(in$,"E")<>0 OR in$=" " a$=""

230 REM\*MINUS SIGN MUST BE FIRST CHARACTER OR FIRST IN EXPONENT

\*

240 IF a$="-" AND in$<>" " THEN IF RIGHT$(in$,1)<>"E" a$=""

250 REM\*ADD TO STRING IF NOT DELETE OR CR\*

260 IF a$<>CHR$(127) AND a$<>CHR$(13) in$=in$+a$

270 REM\*RESPOND TO DELETE\*

280 IF a$=CHR$(127) in$=LEFT$(in$,LEN(in$)-1):IF in$="" in$=" "

290 UNTIL a$=CHR$(13)

300 =VAL(in$)

Either of the two listings could be modified to introduce other limitations appropriate to specific tasks. You might want to preclude input in scientific notation, for example. This would be simply achieved by leaving E out of the list of allowed characters and you could preclude negative numbers by also omitting the minus sign. If you want to allow only positive numbers but retain the possibility of scientific input with a negative exponent a little more subtlety is required. To achieve this change line 240 in Listing 2.1 to:

240 IF digit$="-" AND MID$(number$,i%)<>"E" i%=1000

and the same numbered line in Listing 2.2 to:

240 IF a$="-" AND RIGHT$(in$,1)<>"E" a$=""

A further refinement to Listing 2.2 that you might like to consider is restricting the length of the input line. This could serve to prevent anyone making a mess of you carefully cultivated screen by wandering onto the next line. Assuming that scientific notation is used for very big and very small numbers, and that no-one is going to go in for lots of pointless leading zeros, the maximum sensible input length for numbers on the BBC is around 15 digits. This allows for the maximum number of meaningful significant figures, two signs (one in the exponent), a decimal point, 'E' and two figures in the exponent. So provided that you want the user that scientific notation will be required for large powers of ten you can stop the entering of more than 15 digits without restricting usage of your software. Indeed you might want to reduce the length of input even further if you are expecting values which do not have to span the full range that can be handled by the BASIC. If you alter line 260 in Listing 2.2 to:

260 IF a$<>CHR$(127) AND a$<>CHR$(13) AND LEN(in$)<16 in$=in$+a

$

then only 15 characters may be entered. The requirement for the length of in$ to be less than 16 rather than 15 might be puzzling you and this brings me to a final point about Listings 2.1 and 2.2. I have had to incorporate a few features in both routines to ensure that they will run on the BBC under BASIC 1 since there is a bug in the INSTR function in the early interpreter which can cause a crash if you attempt to use INSTR to look for a longer string contained within a shorter one. To avoid the danger of this I have made the null version of in$ in Listing 2.2 a space rather than an empty string. If I did not do this line 200, for example, mist find itself looking for a single character in an empty string and would crash the machine. In the routine in Listing 2.1 the GOTO in line 200 ensures that line 260 cannot cause a crash for a similar reason. If you have BASIC 2 or are using an Electron you can omit the dreaded GOTO and please the purists.

So far in the discussion of numerical input I have only dealt with real numbers, but BBC BASIC also allows the use of integers. Indeed it is good practice to use them wherever real numbers are not needed since they save storage space and speed program execution. If you are reading values into an integer variable and someone enters a number with a decimal point, the decimal section will be ignored and the number truncated. There is, therefore, some automatic input protection on values entered into integer variables and there is no need to worry about problems in your program resulting from someone entering 2.456 as the number of a disc drive, it will be truncated to 2. Of course you may want to protect the user from this kind of thing for his own benefit. In this case use a routine such as the ones in Listings 2.1 and 2.2 with only numbers and perhaps the minus sign being permitted as input.

The final thing that we must consider about numbers is the fact that BBC BASIC does have a limit to the range of numerical values with which it can cope. Real numbers can range from -2E38 to 2E38 whereas the integer variables can run from -2,147,483,648 to 2,147,483,647. If a number is entered which is outside the permitted range, either too small or too large, the BASIC will give the error Too Large. Because of the way in which it handles negative numbers BBC BASIC only looks at the absolute size of the input, not its sign, so it regards -2E40 as too large rather than too small, which is the way we would view it. The best general way of protecting a program from out of range numbers is to use ON ERROR and if your program is a number cruncher intended for mathematical or scientific use that is the only practical way, since the users must have access to the full range of the machine's capabilities. In more specialized applications you can take input as a string and restrict its length. For example if the value expected is a date in March it must fall in the range 1 to 31 and so you can restrict the length of the input to a maximum of two characters. The necessary code would be something like:

10 REPEAT

20 REPEAT

30 INPUTLINE"Enter the date"date$

40 UNTIL LEN(date$)<3

50 date%=VAL(date$)

60 UNTIL date%>0 AND date%<32

I have also illustrated here the use of integer variables to contain the date. When line 50 is executed, any decimal portion that may have been entered as part of date$ will be removed, as will any powers of ten entered in scientific notation. If date% were replaced by a real variable this would not happen and the fractional date would remain, probably wreaking havoc later on in the program.

Note that a sequence such as:

10 REPEAT

20 INPUTLINE"Enter the Date"date%

30 UNTIL date%>0 AND date%<32

gives no protection from an entry which is too large for the machine since the 'Too Big' error will occur at line 20 as soon as the value is attempted to be put into date%.

If a number is ultimately to go into a real variable is more difficult to prevent excessively large input unless you preclude scientific notation, when there will be no problem. The point is that when using E you can input an enormous number in very few characters whereas if it is banned by a modified version of Listings 2.1 or 2.2, keeping the input line size down to a length that is adequate for the purposes of the particular application will give the required protection. Really the point to consider, in any particular instance, is whether the users need to be able to enter the full range of numbers with which the BASIC can deal. If this is not necessary then curb their enthusiasm!

### String Input

In the last section, although we were concerned with the input of numerical information, it rapidly became clear that this is most safely taken in the form of a string, preferably with specific limitations to prevent the entry of non-valid characters. In this section, I want to look at some more sophisticated ways of taking string input, but these, of course, can equally well be applied to the reading of numbers in string form.

So far, when asking the user to make an input we have issued a prompt and left him to his own devices as to where the data should be typed. The normal response will be to type the input immediately after the prompt, but we have taken no precautions to prevent a messy soul wandering all over the screen. Consider the program fragment:

30 INPUTLINE TAB(0,10)"Please Type Your Name"name$

40 INPUTLINE TAB(0,11)"Age? ";age$

This will work perfectly well until someone enters a very long name which overflows onto the next screen line, cluttering up the place where he is expected to type his age. The program may still work but the screen will be a shambles and very hard to work on. To prevent this kind of thing we need to restrict the area of the screen which can be used to type the input. The technical term for this is that input must be entered in a 'protected field'. Notice that it is no use, in this case, rejecting an excessively long name after it has been entered. By this time the damage has been done and to re-prompt for the name will only add to the mess on the screen.

#### Listing 2.3

10 MODE 7

20 FOR y%=1 TO 21 STEP 3

30 PRINTTAB(0,y%)"Enter up to 10 Characters";

40 input$=FNinput(27,y%,10)

50 PRINT'"You entered ";input$

60 NEXT

70 PRINT'" \*PRESS THE SPACE BAR TO CONTINUE\*"

80 REPEAT UNTIL GET$=" "

90 RUN

100

110 DEF FNinput(x%,y%,maxlen%)

120 LOCAL a$,in$

130 REM\*CURSOR KEYS OFF\*

140 \*FX4,1

150 REPEAT

160 REM\*DISPLAY CURRENT STATE OF INPUT\*

170 PRINTTAB(x%,y%);SPC(maxlen%);TAB(x%,y%);in$;

180 REPEAT

190 a$=GET$

200 REM\*TEST FOR VALID KEYS\*

210 UNTIL (a$>=" " AND a$<=CHR$(127)) OR a$=CHR$(13)

220 REM\*STRIP LAST CHARACTER IF DELETE PRESSED\*

230 REM\*ADD CHARACTER TO in$ IF NOT ALREADY TOO\*

240 REM\*LONG AND IF INPUT IS NOT DELETE OR CR\*

250 IF a$=CHR$(127) in$=LEFT$(in$,(LEN(in$)-1)) ELSE IF a$<>CHR$(13) AND LEN(in$)<maxlen% in$=in$+a$

260 UNTIL a$=CHR$(13)

270 REM\*CURSOR KEYS BACK ON\*

280 \*FX4,0

290 =in$

Listing 2.3 illustrates a simple example of the protected field technique. The entry is taken in one character at a time and built up into a string until the maximum length is reached. The RETURN key is recognized as terminating the input in the normal way and the DELETE key removes the last character typed. The input string is returned by FNinput, which must receive, as parameters, the TAB positions at which the input is to start, x% and y%, and the maximum length of input, as maxlen%.

Line 170 displays the current state of the input line, which is always held in the variable in$. We must delete the old version each time with SPC(maxlen%); it is not sufficient just to overwrite since we are allowing deletions and the new version may be shorter than the previous one. The characters are accepted by GET$ within the REPEAT…UNTIL loop at lines 180 to 210. I have made all the printable characters, the delete key and RETURN, acceptable but you can easily alter line 210 to restrict the range of allowed input characters, to suit your requirements. Remember that you must always allow delete and carriage return. Once a character has been passed as acceptable, line 250 deals with it appropriately.

The delete key, CHR$(127), and carriage return, CHR$(13), require special action, whereas any other character is simply added to in$, provided, that is, in% has not already reached its maximum allowed length. What the user will find is that the keyboard will cease to respond to anything other than DELETE and RETURN once the maximum length of input has been made. I have also disabled the cursor keys with \*FX4,0 and so the only thing that can be done at the keyboard is to type within the permitted protected field.

Once you are controlling input in this kind of way there are all sorts of refinements that can be applied. For example you might want to have all input letters displayed in lower case, for a spelling program perhaps. It would not really be satisfactory to restrict input only to lower case letters since this would require the caps lock to be off. If you change line 190 to:

a$=GET$:IF a$>="A" AND a$<="Z" a$=CHR$(ASC(a$)+32)

you will have the effect that is required since the lower case letters all have ASCII codes 32 greater than the upper case equivalents.

Since in$, in listing 2.3, is printed out at line 170, if we are working in mode 7 it is a very simple matter to have the input displayed in double height characters. I find this particularly useful in educational programs. Changing line 170 to:

170 FOR i%=0 TO 1:PRINTTAB(x%,y%+i%);SPC(maxlen%+1);TAB(x%,y%+1);

CHR$(141);in$;:NEXT

is all that is needed.

The sort of input facility that we are provided in Listing 2.3 actually already exists in the BBC and Electron operating systems. It is used, for example, when you attempt to enter a program line that is too long and further input is refused with a rude bleep. It is possible to access this routine and so avoid having to write something like Listing 2.3. Listing 2.4 shows an example of how to do this. It requires a little insight into the operating system but the essentials are simple enough. A parameter block has to be set up with the relevant information as to the maximum line length to be allowed and the location of a buffer in which the keyboard input is to be temporarily stored. Listing 2.4 some free locations on zero page, starting at &70, and the block is set up at line 50. The first two bytes point to the buffer and the third byte contains the maximum line length, here set to 10, but anything up to 255 could be allowed. The last two bytes of the parameter block contain the lowest and highest ASCII codes to be accepted by the routine. I have set them to cover all the keyboard characters from space to tilde. There is no real point in doing anything else, in fact, since irrespective of these settings, the keyboard responds and the character is displayed on the screen. Out of range characters are not entered into the buffer and do not ultimately figure in the input, so if you do restrict the range there can be great confusion since a user finds that some characters have been ignored even though they appear on the screen.

In Listing 2.4 I have reserved the necessary space for the buffer with a DIM statement at line 40, if you want to take a longer line you will have to enlarge the buffer as necessary. All that is now needed, to take input of limited length with this routine, is to place the cursor at the position where the input is to start and then call OSWORD with zero in the accumulator and with the X and Y registers pointing to the parameter block. This is done at line 90. Whatever is entered is stored in the buffer and can be recovered with the string indirection operator, as at line 100 in listing 2.4. An attempt to exceed the allowed length will be prevented and a bleep will sound just as happens with BASIC line entry.

Once you are taking input in a protected field all kinds of possibilities for convenient and helpful input routines arise. In particular, you can use the whole screen in a far more constructive manner than is possible if people who are using your software are free to wander all over the place. An attempt to take two inputs on the same screen line with a sequence such as:

50 INPUTLINETAB(0,10)"Surname"s$:INPUTLINETAB(20,10)"Pre-

Names"pn$

is a recipe for disaster since a long surname will obtrude into the area needed for the pre-name input, but with protected fields it is no problem.

There is no reason why the input field cannot become the medium for a proper input editor. In the examples so far we have allowed the deletion of the last character of an input, but this would not be very convenient if the input lines were long. If the user missed out a letter at the beginning of an 80 character line and did not notice until he was ready to press RETURN, he would have to delete the whole lot and start again; not really very satisfactory. It would be better to allow him to move the cursor back to the mistake and to insert the character. Deletions should also be possible in a similar manner. The input function in Listing 2.5, which is a refinement of Listing 2.3, provides these facilities.

#### Listing 2.4

10 MODE 6

20 osword%=&FFF1:maxlen%=10:low\_char%=32:high\_char%=126

30 REM\*SPACE FOR BUFFER\*

40 DIM buffer% 20

50 ?&70=buffer% MOD 256:?&71=buffer% DIV 256:?&72=10:?&73=

low\_char%:?&74=high\_char%

60 REPEAT

70 PRINT"Enter up to 10 Characters ";

80 REM\*CALL OSWORD 0\*

90 A%=0:X%=&70:Y%=0:CALL osword%

100 in$=$buffer%

110 PRINT in$

120 UNTIL FALSE

The essential mechanism of the routine uses the variable cx% to keep track of the position of the cursor along the input line so that when a character is entered it can be spliced into the line at the last part of line 250. The first part of this line checks that the line is not already too long and that the key pressed is a character rather than an instruction to move the cursor, to delete, or to accept the line with carriage return. In these cases the character must not be added to the line, merely acted upon.

Line 220 handles deletions, which are carried out by removing the character before the cursor in the conventional manner and then shifting any characters to its right one place to the left. Both deletions and insertions are possible at any point in the line, the cursor being moved in the normal manner by the left and right cursor keys. There are always, of course, the constraints that nothing can be typed outside the protected field area and the line length is limited.

### Screen Handlers

In all input routines discussed up until now the user has been issued with prompts one at a time. There is no chance for the whole picture to be seen before starting to enter the data, each new prompt appears out of the blue. Also, what happens if the user suddenly realizes that a mistake was made ten lines back? There is no way of returning to correct it. The usual method is to wait until all the data is in and then to re-display it with an option to correct, but the inexperienced computer user may not appreciated that this is coming and panic can set in. After all if you are entering responses on a form and you realize that your misspelt your name in the first line you can go back and correct it before you send off the form. What we require is an equivalent system on the computer screen where a screenful of data can be entered, scrutinized and edited if necessary. A key is then pressed to send the entire screen of data, just like posting your form. This kind of thing can easily be done by an extension of the protected field idea.

#### Listing 2.5

10 MODE 7

20 FOR y%=1 TO 21 STEP 3

30 PRINTTAB(0,y%)"ENTER UP TO 10 CHARACTERS ";

40 input$=FNgetinput(27,y%,10)

50 PRINT'"YOU ENTERED ";input$

60 NEXT

70 PRINT'" \*PRESS THE SPACE BAR TO CONTINUE\*";:REPEAT UNTIL GET$=" "

80 RUN

90

100 DEF FNgetinput(x%,y%,maxlen%)

110 \*FX4,1

120 LOCAL in$,maxx%,cx%,a$:cx%=x%:maxx%=cx%+maxlen%

130 REPEAT

140 PRINTTAB(x%,y%);SPC(maxlen%);TAB(x%,y%);in$;TAB(cx%,y%);

150 REPEAT

160 a$=GET$

170 UNTIL (a$>=" " AND a$<=CHR$(127)) OR INSTR(CHR$(13)+CHR$(13

7)+CHR$(136),a$)

180 REM\*CURSOR RIGHT BUT NOT BEYOND END OF\*

181 REM\*CURRENT INPUT LINE\*

190 IF a$=CHR$(136) cx%=cx%-1:IF cx%<x% cx%=x%

200 REM\*CURSOR LEFT BUT NOT BEYOND END OF FIELD\*

210 IF a$=CHR$(137) tx%=cx%:cx%=cx%+1:IF cx%>x%+LEN(in$):cx%=tx

%

220 IF a$=CHR$(127) THEN IF cx%>x% in$=LEFT$(in$,cx%-x%-1)+MID$

(in$,cx%-x%+1):cx%=cx%-1

230 REM\*ADD CHARACTER TO in$ AT CURSOR IF\*

240 REM\*ROOM AND THE CHARACTER IS NOT A CONTROL\*

250 IF LEN(in$)<maxlen% AND INSTR(CHR$(127)+CHR$(136)+CHR$(137)

+CHR$(13),a$)=0 in$=LEFT$(in$,cx%-x%)+a$+MID$(in$,cx%-x%+1):cx%=c

x%+1

260 UNTIL a$=CHR$(13)

270 \*FX4,0

280 =in$

What we have to do, is to set up a series of variables, normally array elements, for the various pieces of data, and to associate each element with a particular field on the screen. We must then have a mechanism for moving a cursor around the screen, and by keeping track of the position of that cursor we can tell which field we are in, and hence to which of our variables any input is to be added. There is also no reason why the lengths of, and the characters allowed in, each of the fields cannot be different. We could, for example, allow only numbers where appropriate and force the filling in of textual information in the traditional block capitals. All that remains is to arrange for a particular key to be recognized as terminating the input and so accepting the whole screen. Input software which works in this fashion is often described as a 'screen handler'.

Listing 2.6 applies this form-filling idea to the entry of basic personal particulars. The RETURN key moves the cursor from one protected field to the next and the TAB key is pressed to accept the screen. On the Electron CTRL-I will have to be used. Within each field the cursor can be moved left and right to insert characters and the DELETE key used as normal. The procedure, PROCsetup, at line 380, sets up the screen and Figure 2.1 shows what the user will actually see, the square brackets marking the boundaries of the various input fields. Each input has four attributes, or limitations, which must be taken into account. The first three are its maximum length, and the two tab positions of the start of the input field. These numerical attributes are all held in the two dimensional array entformat%.

For the field n, entformat%(n,0) holds the length, and the remaining two sub-elements of the nth element hold the X and Y tabs respectively. The fourth attribute is a string which defines the characters acceptable for that field. These are held in the corresponding elements of the string array allowed$, the entries themselves being accumulated into the array ent$. The attributes of each field are specified in the DATA statement at line 150, simple abbreviations being permitted for the commonest sets of allowed characters, that is numbers or block capitals. The main control loop for the taking of input is the REPEAT…UNTIL structure between lines 180 and 250, which is only terminated if the TAB key is pressed, and if all sections of the 'form' are completed. The function at line 750 checks this latter condition.

PROCtakeinput at line 500 actually reads in the input. It works in exactly the same way as the function in Listing 2.5 and is exited on either the RETURN or TAB keys, the specific key being passed back in the global variable a$. The main loop can then take the appropriate action, at lines 220 or 250, either accepting the screen or moving the cursor to the next field. If the screen is accepted the data entered is displayed, starting at line 270.

#### Listing 2.6

10 MODE 6

20 no$="1234567890"

30 text$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"

40 DIM entformat%(5,2),ent$(5),allowed$(5)

50 RESTORE

60 FOR I%=0 TO 5:FOR J%=0 TO 2

70 READ entformat%(I%,J%)

80 NEXT

90 READ allowed$(I%)

100 IF allowed$(I%)="NO" allowed$(I%)=no$

110 IF allowed$(I%)="TEXT" allowed$(I%)=text$

120 NEXT

130 REM\*ATTRIBUTES OF INPUT REQUIRED\*

140 REM\*MAXIMUM LENGTH, X-TAB, Y-TAB AND ALLOWED CHARACTERS\*

150 DATA 15,16,10,TEXT,20,16,12,TEXT,2,16,14,NO,1,16,16,YN,2,36,16,NO,1

,16,18,MF

160 PROCsetup

170 I%=0

180 REPEAT

190 REM\*READ INPUT INTO ent$(I%)\*

200 PROCtakeinput(I%,entformat%(I%,1),entformat%(I%,2),entforma

t%(I%,0),allowed$(I%))

210 REM\*CR MOVES TO NEXT FIELD\*

220 IF a$=CHR$(13) I%=I%+1:IF I%>5 I%=0

230 REM\*ACCEPT SCREEN ON TAB KEY PROVIDED ALL SECTIONS COMPLETE

D\*

240 REM\*a$ DETERMINED ON EXIT FROM PROCtakeinput\*

250 UNTIL a$=CHR$(9) AND FNcompleted

260 REM\*DISPLAY INPUT DATA\*

270 CLS

280 PRINT ent$(1)+" "+ent$(0)+" is"

290 PRINT ent$(2)+" years old,";

300 IF ent$(3)="Y" PRINT " married"; ELSE PRINT "single,";

310 IF ent$(5)="M" PRINT " and male." ELSE PRINT " and female."

320 IF VAL(ent$(4))<>0 THEN PRINT;ent$(4);" Children."

330 PRINT'"\*PRESS SPACE BAR TO RE-RUN THE PROGRAM\*"

340 REPEAT UNTIL GET$=" "

350 RUN

360

370 REM\*EXAMPLE FORM\*

380 DEFPROCsetup

390 CLS

400 PRINT"Please enter your particulars. DELETE and the left and right cursor keys can be used to edit. RETURN will take you to the next entry and back to the first entry from the last. When you are happy"

410 PRINT"that everything is correct press the TABkey to accept the whole screen."

420 PRINTTAB(0,10)"Surname";TAB(15,10);"[";SPC(entformat%(0,0))

;"]"

430 PRINTTAB(0,12)"Given Names";TAB(15,12);"[";SPC(entformat%(1

,0));"]"

440 PRINTTAB(0,14)"Your age"TAB(15,14);"[";SPC(entformat%(2,0))

;"]"

450 PRINTTAB(0,16)"Married(Y/N)";TAB(15,16);"[";SPC(entformat%(

3,0));"]"

460 PRINTTAB(20,16)"No of Children";TAB(35,16);"[";SPC(entforma

t%(4,0));"]"

470 PRINTTAB(0,18)"Your Sex(M/F)";TAB(15,18);"[";SPC(entformat%

(5,0));"]"

480 ENDPROC

490

500 DEFPROCtakeinput(i%,x%,y%,maxlen%,allowed$)

510 \*FX4,1

520 LOCALmaxx%,cx%:cx%=x%:maxx%=cx%+maxlen%

530 allowed$=allowed$+CHR$(127)+CHR$(13)+CHR$(136)+CHR$(137)+CH

R$(9)

540 REPEAT

550 REM\*DISPLAY CURRENT INPUT IN CURRENTLY SELECTED FIELD\*

560 PRINTTAB(x%,y%);SPC(maxlen%);TAB(x%,y%);ent$(i$);TAB(cx%,y%

);

570 REPEAT

580 a$=GET$

590 REM\*CHECK FOR VALID KEYS\*

600 UNTIL INSTR(allowed$,a$)

610 REM\*CURSOR LEFT\*

620 IF a$=CHR$(136) cx%=cx%-1:IF cx%<x% cx%=x%

630 REM\*CURSOR RIGHT\*

640 IF a$=CHR$(137) tx%=cx%:cx%=cx%+1:IF cx%>x%+LEN(ent$(i%)):c

x%=tx%

650 REM\*DELETE\*

660 IF a$=CHR$(127) THEN IF cx%>x% ent$(i%)=LEFT$(ent$(i%),cx%-x%-1)+MID$(ent$(i%),cx%-x%+1):cx%=cx%-1

670 REM\*ADD CHARACTER IF ROOM AND CHARACTER NOT CONTROL\*

680 IF LEN(ent$(i%))<maxlen% AND INSTR(CHR$(127)+CHR$(136)+CHR$(137)+CHR$(13)+CHR$(9),a$)=0 ent$(i

%)=LEFT$(ent$(i%),cx%-x%)+a$+MID$(ent$(i%),cx%-x%+1):cx%=cx%+1

690 REM\*EXIT ON CR OR TAB KEY\*

700 UNTIL a$=CHR$(13) OR a$=CHR$(9)

710 \*FX4,0

720 ENDPROC

730

740 REM\*CHECK THAT ENTRIES MADE IN ALL FIELDS\*

750 DEF FNcompleted

760 LOCAL i%

770 FOR i%=0 TO 4

780 IF ent$(i%)="" THEN i%=1000

790 NEXT

800 IF i%>5 THEN =FALSE ELSE =TRUE

Please enter your particulars. DELETE

and the left and right cursor keys can

be used to edit. RETURN will take you to

The next entry and back to the first

entry from the last. When you are happy

that everything is correct press the TAB

key to accept the whole screen.

Surname [ ]

Given Names [ ]

Your age [ ]

Married(Y/N) [ ] No of Children [ ]

Your Sex(M/F) [ ]

##### **Figure 2.1.** Screen presented by Listing 2.6

A useful refinement to taking input in this manner, is to make the allowed entries into some of the fields dependent upon what has been input elsewhere. Perhaps an example will help to clarify this. Suppose we were to take a high moral tone and to decide that only married people had children, then anyone entering 'N' into the box for Married? should be precluded from entering anything other than zero in the box for the number of children!

To achieve this we require to make the array element allowed$(4), which relates the number of children, a variant dependent upon the contents of ent$(3), which contains the marital status information. This is not quite as straightforward as it may seem at first, however, since there is no constraint on the order in which the entries are made on the 'form'.

A number of children, other than zero, may have been entered *before* the marital status is set to 'N', for example. To allow for this we need to monitor the marital status continuously and, if it ever becomes a 'N', arrange to set the number of children to a null, both on screen and as stored in ent$(4), as well as changing the value in allowed$(4) to '0'. Also, since the martial status may equally be changed in the opposite direction, from 'N' to 'Y', or even simply deleted, we must be prepared to restore allowed$(4) to its normal state of allowing any numbers. The addition of the following lines to Listing 2.6 will have the desired effect.

685 IF ent$(3)="N" AND i%=3 allowed$(4)="0":ent$(4)="":PRINTTAB

(entformat%(4,1),entformat%(4,2));" ":entformat%(4,0)=1

686 IF(a$="Y" AND i%=3) OR ent$(3)="" THEN allowed$(4)=no$:entf

ormat%(4,0)=2

This kind of variant entry can be extremely helpful to inexperienced users, in particular, but be careful not to try anything too fancy until you have a good grasp of the method, since it can lead you into some very complex programming.

### Editing Data

So far we have dealt only with the entry of completely new data, but many applications require data already present to be edited. This may originate from earlier input that was incorrect or it may have been loaded from tape or disc and require to be updated. The editor software will need to meet the following requirements.

1. Allow the user to identify and select the piece of data to be altered.
2. Display the old data for reference whilst the new version is being entered.

A really friendly editor will also try to give the user the maximum assistance by allowing him either to copy from the old version directly to the new, making any amendments as required, or to change the old entry directly by means of cursor editing. The BBC BASIC editor adopts the first of these methods, whereas many editors on other machines have cursor editing facilities.

Listing 2.7 shows an example of a simple copy editor. The current data is displayed to the user at lines 90 to 130. Since there are more entries than will fit on one screen, 'page' mode is switched on at line 90. Each entry in the data table is numbered to enable the user to select the entry to be edited at line 220. Line 250 to 290 display the existing contents of the line and take the new input. Copying can be done in the normal way using the 'COPY' and cursor control keys and to facilities this the copy and ordinary cursors are separated, the former being placed at the start of the old version of the data and the latter where the new input is to be taken. This is done in PROCup at line 350. Line 380 is the important one for performing this trick; it deceives the machine into thinking that the cursor up key has been pressed five times, so placing the copy cursor correctly. The call \*FX138,n,p enters the character with ASCII code p into the machine buffer n. The keyboard buffer has n=0 and CHR$(255) is the control code recognized as cursor up. The cursor can also be moved down, right or left, respectively by entering CHR$(254), CHR$(253) or CHR$(252) into the keyboard buffer in the same way.

The example in listing 2.7 is a simple one, the input being taken in a fairly straightforward manner, but any of the more sophisticated input techniques that we have already discussed could be used to provide appropriate protection.

Selecting the data to be edited from a numbered list, as in Listing 2.7, is essentially the same as operating a menu, and other menu selection techniques, such as the moving pointer, can equally be applied, although the latter is only really adapted to editing blocks of data which can fit into one screen.

If your data does not fit onto a single screen a more sophisticated editing technique, the screen editor, can easily be applied. This has many features in common with the screen handler we employed in the 'form filling' input routine of the last section of this chapter. In a similar manner the input required to be edited is identified by moving a cursor around the screen and, by keeping track of the position of this cursor, we can determine which variable is to be altered, and at what point, when inputs are made.

Listing 2.8 shows an example of a screen editor. Twenty five lines of text are displayed and these can be edited by moving a cursor around the screen freely to make insertions and deletions. The 25 lines are held in the array text$ and are printed on the screen at vertical tab positions that correspond numerically to the elements of the array in which each line is contained. For example text$(10) is displayed at screen line 10 and so the vertical cursor position is easily related to the element of text$ being edited. The function which actually takes the input begins at line 220 and works in a very similar manner to FNgetinput in Listing 2.5, the main difference being that the current contents of the string being edited have to be passed to the subroutine in Listing 2.8, since there is no reason to suppose that it starts as a null in this case.

#### Listing 2.7

10 MODE 7

20 @%=00000002:REM\*TWO NUMBERS IN FIELD\*

30 DIM line$(50)

40 FOR i%=0 TO 50

50 line$(i$)="THIS IS LINE "+STR$(i%+1)

60 NEXT

70

80 REPEAT CLS

90 VDU 14:REM\*PAGE MODE ON\*

100 FOR i%=0 TO 50

110 PRINT,i%+1;" ";line$(i%)

120 NEXT

130 PRINT'"'C'-Edit:'R'-Redisplay:'E'-End"

140 REPEAT A$=GET$:UNTIL INSTR("CcEeRr",A$)

150 IF INSTR("cC",A$) PROCedit

160 UNTIL INSTR("Ee",A$)

170 END

180

190 DEFPROCedit

200 LOCAL ln%,temp$

210 REPEAT CLS

220 INPUTLINETAB(0,10);"Which Line is to be edited ? "ln%

230 UNTIL ln%>0 AND ln%<52

240 REPEAT CLS

250 PRINTTAB(0,5)"Current Line ";ln%;" is:-"'

260 PRINT;line$(ln%-1)''

270 PRINT"Enter the New Version-25 characters max"''TAB(25);">

";CHR$(11)

280 PROCup(5)

290 INPUTLINE'temp$

300 IF LEN(temp$)>25 VDU7:PRINT'"TOO LONG"'e$=INKEY$(500)

310 UNTIL LEN(temp$)<26

320 line$(ln%-1)=temp$

330 ENDPROC

340

350 DEFPROCup(n%)

360 LOCAL i%

370 FOR i%=1 TO n%

380 \*FX138,0,255

390 NEXT

400 ENDPROC

#### Listing 2.8

10 MODE 4

20 DIM text$(24)

30 FOR i%=0 TO 24:text$(i%)="THIS IS LINE "+STR$(i%):NEXT

40 control$=CHR$(127)+CHR$(13)+CHR$(137)+CHR$(138)+CHR$(139)+C

HR$(136)

50 FOR i%=0 TO 24:PRINTTAB(5)text$(i%):NEXT

60 PRINT'"Press RETURN when data is all acceptable"

70 i%=0

80 REPEAT

90 test$(i%)=FNtakeinput(text$(i%),5,i%,30)

100 REM\*CURSOR DOWN\*

110 IF a$=CHR$(138) i%=i%+1:IF i%>24 i%=0

120 REM\*CURSOR UP\*

130 IF a$=CHR$(139) i%=i%-1:IF i%<0 i%=24

140 REM\*EXIT ON RETURN KEY\*

150 UNTIL a$=CHR$(13)

160 CLS:PRINTTAB(10)"THE ENTRIES NOW READ"''

170 FOR i%=0 TO 24:PRINTTAB(5)text$(i%):NEXT

180 PRINT'" PRESS THE SPACE BAR TO RE-RUN PROGRAM"

190 REPEAT UNTIL GET$=" "

200 RUN

210

220 DEF FNtakeinput(ent$,x%,y%,maxlen%)

230 \*FX4,1

240 LOCALmaxx%,cx%:cx%=x%:maxx%=cx%+maxlen%

250 REPEAT

260 REM\*DISPLAY CURRENT INPUT IN CURRENTLY SELECTED FIELD\*

270 PRINTTAB(x%,y%);SPC(maxlen%);TAB(x%,y%);ent$;TAB(cx%,y%);

280 REPEAT

290 a$=GET$

300 REM\*CHECK FOR VALID KEYS\*

310 UNTIL (a$>=" " AND a$<="~" OR INSTR(control$,a$)

320 REM\*CURSOR LEFT\*

330 IF a$=CHR$(136) cx%=cx%-1:IF cx%<x% cx%=x%

340 REM\*CURSOR RIGHT\*

350 IF a$=CHR$(137) tx%=cx%: cx%=cx%+1:IF cx%>x%+LEN(ent$):cx%=

tx%

360 REM\*DELETE\*

370 IF a$=CHR$(127) THEN IF cx%>x% ent$=LEFT$(ent$,cx%-x%-1)+MI

D$(ent$,cx%-x%+1):cx%=cx%-1

380 REM\*ADD CHARACTER IF ROOM AND CHARACTER NOT CONTROL\*

390 IF LEN(ent$)<maxlen% AND INSTR(control$,a$)=0 ent$=LEFT$(en

t$,cx%-x%)+a$+MID$(ent$,cx%-x%+1):cx%=cx%+1

400 REM\*EXIT ON CURSOR UP OR DOWN OR CR\*

410 UNTIL INSTR(CHR$(13)+CHR$(138)+CHR$(139),a$)

420 \*FX4,0

430 =ent$

The cursor up and down keys are used to move through the list, the necessary code being at lines 90 to 130, and the return key exists from the editor and, in this test program, merely displays the current entries.

### Conclusions

By the time that you have reached this point in the chapter you will probably be beginning to wonder why Acorn bothered to provide any line input routines at all in BBC BASIC. You do need them of course, but except in the most simple circumstances, they rarely provide the best way of taking input. The other thing that has perhaps crossed your mind is that, if you use the sort of techniques that we have been concerned with, a lot of memory is going to be swallowed up before you start doing anything with the data that is entered.

There is often much truth in this and it is frequently worth considering whether to write the input and editing routines as a completely separate program which, ultimately, writes the validated input to a disc or cassette file from which the main program can retrieve it. This is much more suited to disc than tape, of course, but even with tape it is worth considering, especially if the people who are going to be using the input data are not the same as those making the input. In a school application, for example, the teacher may be entering and editing a set of works into a Hangman program for use by pupils. Here it is very appropriate to have two separate modules, an editor for the teacher and the actual game program for the pupils, both being able to access the same tape or disc file.

# Chapter 3.

# Output to Screen and Printer

You will usually want the output from your software to go to the screen, perhaps with an option to use the printer for the increasing number of users who own one. Even so, except in special applications, the screen will generally be the most important medium for output. The clarity and neat organization of your displays is one of the things by which your software will stand or fall, and indeed many programmers' work shows a distinctive style in this area. It is certainly very worthwhile giving careful consideration to screen layout and to exploiting the very considerable facilities in the hardware to best advantage. As we will see, you can actually get the computer to do most of the hard work in screen formatting for you.

### Choosing the Mode

The BBC and Electron have a great variety of display modes and some careful consideration is required when choosing the right one for any particular job. Assuming that you are running on an Electron or BBC without a second processor or other memory expansion you will not only have to consider the ideal mode for the display but you may also have to come to a compromise with the memory requirements of the program. The following points are all worth considering.

1. Eighty column text modes are rarely satisfactory if you are intending the software to be useable with a domestic television. You may just get away with an ordinary resolution colour monitor but ideally a monochrome or high resolution colour monitor is required, especially for prolonged use without eye strain. If your display relies on eighty columns *and* colour distinction it will only really be seen to effect by users with high resolution colour monitors.
2. Unless you need high resolution graphics use the text only modes 3 and 6, or mode 7. Their 25 lines per screen spacing makes text much more readable then the rather cramped 32 lines per screen used in the graphics modes. On the BBC, mode 7 is usually a better choice than mode 6 for 40 column display, because of its great memory advantage, very clear characters and colour and graphics capability. Programs written for the BBC in mode 7 will run on the Electron, defaulting to mode 6, but will only be fully compatible if no teletext control characters are used.

Any such characters will have unpredictable, and possibly weird, effects on the Electron display.

1. The 20 column modes, 2 and 5, are unsuitable for large amounts of text output. The rather oddly shaped characters are really intended for labelling and short messages on graphics presentations and in games.
2. Do consider the colour combinations that you use. Remember that someone may want to run your software in monochrome and that a very impressive colour display may be quite unintelligible on a black and white television or green monitor. I do not mean by this that you should forget about the colour capabilities of your machine. Far from it, but if you are using colour for emphasis, try to make sure that it serves its purpose in all instances. No one will expect to play an arcade game in monochrome but be ready for complaints if your statistics program outputs the numbers in red on black. It may look great on your colour monitor but a serious user may well not be using one and he will not be too happy about having to turn up the contrast to tube-blowing point!

Points to bear in mind are that black, magenta, blue and red are not easily distinguished in monochrome making the last three unsuitable if you work with the default background. Green and yellow are also not well contrasted, nor are cyan and white. The best advice is to have a look at your software in both monochrome and colour whenever possible.

### Output Screen Handling

Always approach the design of a screen in the same way as you would attack the design of a diagram or the layout of a page in a book. The user must be able to find all the relevant data and it must be clearly labelled. You will also need to tell him what has to be done next. Is there more data accessed by a key press, for example? I have said this already and I shall, no doubt, say it again; never leave the user looking at a screen with no indication of what to do next, even if it is just a message that something is going on and that he should wait.

The output must always be under control and if there is more than one screenful you will have to provide some way of presenting it a screen at a time. The simplest way of doing this is by setting the machine in 'page mode', with VDU14. This will halt the display every 18 or 24 lines, depending on the mode, and wait for the SHIFT key.

Listing 3.1 shows how page mode can be used to provide a continuously scrolling display of, in this case, 100 lines of output organized so that when line 100 has been printed it is followed by line 1 again and so son. The loop between lines 170 and 220 keeps printing the output but since page mode is on, after filling the screen, the display stops until SHIFT is pressed. One of the problems of using page mode in this way is that when the machine is waiting for the SHIFT key program execution is suspended and you cannot test for another key to exit the routine; only SHIFT is recognized. In fact the escape key will also be active and so one way of arranging an exit is to use an ON ERROR statement to detect the escape. This does restrict where you can jump to; you cannot go into a procedure, for example, so in Listing 3.1 I have used an alternative technique, testing for the DELETE key at line 200.

#### Listing 3.1

10 MODE 6

20 VDU19,0,4,0,0,0:REM\*BLUE BACKGROUND\*

30 PRINTTAB(0,23)"Press SHIFT key to continue when the disp

lay stops. SHIFT/DELETE to exit.";

40 VDU28,0,22,39,0:REM\*TEXT WINDOW\*

50 DIM line$(99)

60 FOR I%=0 TO 99

70 IF I% MOD 2 e$="\*" ELSE e$="#"

80 line$(I%)=e$+"THIS IS LINE NUMBER "+STR$(I%+1)+e$

90 NEXT

100 PROCdisplay

110 MODE 6

120 END

130

140 DEFPROCdisplay

150 LOCAL i%,exitflag%:CLS

160 VDU 14:REM\*PAGE MODE ON\*

170 REPEAT

180 FOR i%=0 TO 99

190 PRINT line$(I%)

200 IF INKEY(-90) i%=1000:exitflag%=TRUE

210 NEXT

220 UNTIL exitflag%

230 ENDPROC

If DELETE is detected, this causes an exit from the FOR…NEXT loop and sets exitflag% to FALSE so that line 220 will exit the procedure. Whilst screen scrolling is frozen by a page end the test at line 200 will never be applied but this is overcome by telling the user to press DELETE *and* SHIFT to exit the routine. This ensures that the screen is scrolling and the test at line 200 is continually executing.

#### Listing 3.2

10 MODE 6

20 VDU19,0,4,0,0,0:REM\*BLUE BACKGROUND\*

30 PRINTTAB(0,23)"Cursor up and cursor down to move aroundthe

list. Press RETURN to exit";

40 VDU28,0,22,39,0:REM\*TEXT WINDOW\*

50 DIM line$(99)

60 FOR I%=0 TO 99

70 IF I% MOD 2 e$="\*" ELSE e$="#"

80 line$(I%)=e$+"THIS IS LINE NUMBER "+STR$(I%+1)+e$

90 NEXT

100 PROCprintpage

110 PROCupdown

120 CLS

130 END

140

150 DEFPROCupdown

160 LOCAL a$,allowed$

170 allowed$=CHR$(135)+CHR$(138)+CHR$(139)+CHR$(13)

180 \*FX4,1

190 REPEAT

200 REPEAT \*FX15,1

210 a$=GET$

220 UNTIL INSTR(allowed$,a$)

230 REM\*COPY KEY HOMES TO TOP\*

240 IF a$=CHR$(135) PROCprintpage

250 REM\*MOVE DOWN\*

260 IF a$=CHR$(139) FOR i%=0 TO ABS(INKEY(-1)\*9):PROCmovedown:

NEXT

270 REM\*MOVE UP\*

280 IF a$=CHR$(138) FOR i%=0 TO ABS(INKEY(-1)\*9):PROCmoveup:NE

XT

290 REM\*EXIT ON CR\*

300 UNTIL a$=CHR$(13)

310 \*FX4,0

320 ENDPROC

330

340 DEFPROCprintpage

350 LOCAL i%

360 CLS:firstline%=0:lastline%=22

370 FOR i%=0 TO 22

380 PRINT line$(i%);

390 IF i%<>22 PRINT

400 NEXT

410 ENDPROC

420

430 DEFPROCmoveup

440 firstline%=FNmod100(firstline%+1)

450 lastline%=FNmod100(lastline%+1)

460 PRINTTAB(0,22)'line$(lastline%);

470 ENDPROC

480

490 DEFPROCmovedown

500 firstline%=FNmod100(firstline%-1)

510 lastline%=FNmod100(lastline%-1)

520 VDU30,11:PRINTTAB(0,0) line$(firstline%);

530 ENDPROC

540

550 DEFFNmod100(n%)

560 IF n%>99 THEN =n%-100

570 IF n%<0 THEN =n%+100

580 =n%

In Listing 3.1 the data being printed out is contained in an array but this is only for compactness in the example. You could have any sequence of print statements in place of lines 180 to 210 and the routine would work, the only thing to remember is to test the DELETE key at least once every 18 lines, and preferably more frequently, so as to allow an exit on any page by setting the exit flag and then jumping to the equivalent of line 220, perhaps with a GOTO. In Listing 3.1, since the printing involves a FOR…NEXT loop, it is convenient to apply the test once at each pass through the loop and no jump is necessary, we merely ensure that an exit from the FOR…NEXT structure will be made when line 210 is next executed by changing the value of the counter, i% to a high value.

Another point to note, with regard to Listing 3.1, is that the instructions for operating the display are continuously shown on the two bottom screen lines. They are printed at line 30 and then the active screen is restricted to a text window, at line 40, which does not include the last two lines of the display.

Page mode can only be used to achieve controlled scrolling of the screen in the normal direction, if you want to be able to scroll the display either up or down then you will have to arrange it yourself. Listing 3.2 shows how this can be done for the same output as in listing 3.1. The cursor up and down keys are used to scroll in the appropriate directions and RETURN exists from the routine. An upward scroll is achieved by moving the cursor to the bottom of the screen and printing a return character whereas scrolling downwards involves homing the cursor to the top left of the screen and then sending a cursor-up code, character 11. Once the screen has been scrolled there will be a blank line, at either top or bottom, and the correct output data must now be printed there. In order to relate the array elements of the output to the screen I have used two pointers, lastline% and firstline%, which are moved so as to remain 22 lines apart, so the line to be put in the vacant screen line can easily be identified. There is a slight complication in that we want to operate a continuous loop, so if, for example, the last line becomes 101, we need to subtract 100, so that the line displayed is actually line 1. Similarly a line number of less than one must be increased by 100, so that line zero becomes line 100, for example. This is done by the function at line 550, remember the array elements run from 0 to 99 rather than 1 to 100.

Output scrolling in both directions, as in Listing 3.2, can only really be achieved if the data is in an array since it is essential to have the lines numbered, as it were, in order to sort out which is to be printed when the screen scrolls. Moreover, you need advance warning of the size of the away if you want to operate in a continuous loop. The single direction scroll, as exemplified by Listing 3.1, is much more generally applicable, being able to deal with arbitrarily long runs of output from any combination of data structures.

Another method that you might like to consider for dealing with output requiring more than one screen is to display it as a series of pages that, rather than scrolling continuously, can be changed like turning the pages in a book. Listing 3.3 shows how this can be implemented in a very flexible manner and also demonstrates that there are still uses for GOSUB in BBC BASIC. As usual I have used the cursor keys to control the sequencing and RETURN for the exit. The main work is done in the procedure at line 180 which calls a separate subroutine for each page by means of the computer GOSUB at line 200. There is no way of calling a procedure in BBC BASIC on the basis of a numerical result, except by the rather cumbersome method of using a succession of IF statements. In this sort of situation the computed GOSUB is neater and more flexible. All that remains to be done is to incorporate whatever output sequence you want into the various subroutines and then the user can page through the results at leisure. Adding additional 'pages' is simplicity itself, just increase the value of maxpage% and add page n as subroutine n\*1000. The only drawback with the computed GOSUB is that you will need to be careful about renumbering the program.

### Numerical Formatting

BBC BASIC has very powerful facilities for controlling the format in which numbers are output. The key to the system is the use of the variable @% which is best treated as a four byte hexadecimal number with the most significant byte, on the left, being regarded as byte 4.

The use of @% is fully explained in the User Guides so I do not intend to rehearse it here except to point out the particular usefulness of byte 4 of @% for creating neat screens. This byte can be either 01 or 00 and controls whether the STR$ function, which converts a number into a string, takes account of @% when it operates. The default is zero, which means that STR$ will ignore @%. Turning on the option for STR$ to take account of @% can be extremely useful, since it lets us case a number into a suitable form and then examine it as a string variable and so look at things like the length of screen line that it will occupy.

Listing 3.4 illustrates the use of the technique for a particularly common type of output involving a list of items each with a price or cost attached. At line 20 I have set @% to &0102020a. This means, reading the bytes from left to right:

Byte 4 = &01 – STR$ to take account of @%

Byte 3 = &02 – Output always to have fixed number of decimal places

Byte 2 = &02 – Two decimal places. The second place will be rounded up if appropriate.

Byte 1 = &0A – Print field width to be 10.

#### Listing 3.3

10 MODE 6

20 PRINTTAB(0,23)"Cursor down to move forwards, cursor up to move back, RETURN key to exit.";

30 VDU19,0,4,0,0,0:REM\*BLUE BACKGROUND\*

40 VDU28,0,22,39,0:REM\*TEXT WINDOW\*

50 VDU23,1,0;0;0;0;:REM\*CURSOR OFF\*

60 \*FX4,1

70 maxpage%=3:page%=1

80 REPEAT

90 \*FX15,1

100 PROCselect(page%)

110 REPEAT a$=GET$:UNTIL INSTR(CHR$(13)+CHR$(138)+CHR$(139),a$)

120 IF a$=CHR$(138) page%=page%+1:IF page%>maxpage% page%=1

130 IF a$=CHR$(139) page%=page%-1:IF page%<1 page%=maxpage%

140 UNTIL a$=CHR$(13)

150 MODE 6

160 END

170

180 DEFPROCselect(page%)

190 CLS:pn$="Page "+STR$(page%):PRINTTAB(39-LEN(pn$))pn$;

200 GOSUB (page%\*1000)

210 ENDPROC

220

1000 PRINTTAB(10,20)"THIS IS PAGE 1"

1010 RETURN

2000 PRINTTAB(10,20)"THIS IS PAGE 2"

2010 RETURN

3000 PRINTTAB(10,20)"THIS IS PAGE 3"

3010 RETURN

This will ensure that any monetary value is represented correctly in pounds followed by pence as two decimal places and that this format is observed when STR$ is applied at line 160. The main job of line 160 is to build up a string of exactly 40 characters, which starts with the description of the item on the price list and ends with the second figure in the pence column. This will ensure that the prices are all correctly aligned right-justified on a forty column screen. Notice the semicolon at the end of line 80 to suppress the line feed since we are printing up to the right right margin and so there is an automatic new line generated. If you want to print with double spacing just omit the semicolons. The routine can be easily modified for eighty columns by changing the 40 in line 160 to 80.

#### Listing 3.4

10 MODE 6

20 @&=&0102020A

30 REPEAT CLS

40 PRINTTAB(12)"TODAY'S SPECIALS"'

50 RESTORE

60 FOR I%=1 TO 5

70 READ descript$

80 PRINT FNcashprint(descript$,RND(10)+RND(1));

90 NEXT

100 REM\*DESCRIPTIONS\*

110 DATA Guinea Pig Tails,Cheese Rolls,Bacon and Egg Pie,Water Cress Sandwiches,Cheshire Cheese Rolls

120 REPEAT UNTIL GET$=" "

130 UNTIL FALSE

140

150 DEF FNcashprint(text$,lsd)

160 =text$+STRING$(40-LEN(text$)-LEN(STR$(lds))," ")+STR$(lsd)

Whenever you want full control of numerical output there is much to be said in favour of converting the numbers into strings as I have done in Listing 3.4 and the ability to construct strings of specified format by invoking the fourth byte of @% makes the technique doubly useful.

### Text Output

Text output is generally controlled in BBC BASIC with the TAB command and this is excellent provided that you know exactly what you have to print on the screen, but if you have to handle varied and perhaps unpredictable string variable output, say from a wordprocessor, you will need to arrange for more flexible format control.

Printing of lines justified left on the screen is simple enough, just make sure that you start at TAB(0) but what if you want to fully indent lines to come up against the right margin? Or perhaps to centre lines on the screen?

#### Listing 3.5

10 MODE 6

20 screen\_width%=40

30 REPEAT

40 REPEAT CLS

50 INPUTLINE A$

60 UNTIL LEN(A$)<41

70 PROCcentre(A$,5)

80 PROCrightadj(A$,10)

90 REPEAT UNTIL GET=32

100 UNTIL FALSE

110

120 DEFPROCcentre(text$,y%)

130 PRINTTAB((screen\_width%-LEN(text$)) DIV 2,y%)text$;

140 ENDPROC

150

160 DEFPROCrightadj(text$,y%)

170 PRINTTAB(screen\_width%-LEN(text$),y%)text$;

180 ENDPROC

Listing 3.5 shows how both of these can be achieved for a forty column screen; for eighty columns change the variable screen\_width% appropriately. There is nothing very profound about Listing 3.5. I just wanted to make the point that the computer itself can do the formatting for you.

When the size of text output increases to more than one screen line we have to face the problem of avoiding words being split between lines. If you know in advance what is to be printed then this is not difficult, but for varied and unpredictable output, such as might be produced by a database or wordprocessor, formatting software will have to be provided. Essentially what is required is a scrutiny of each word in the output, before it is printed, so that a check can be carried out to see if the word will overflow the current line. The pseudo-variable COUNT in BBC BASIC is a great help here since it holds the number of characters output since the last linefeed and so can be used to check where on a line the current print position is. Note that COUNT is only reset to zero by an actual software linefeed; the move onto the next line that arises as a result of overflowing a screen line is a function of the hardware and does not reset COUNT. Clearing the screen also resets count except that in BASIC 1 the clear screen that goes with a mode change leaves COUNT unchanged. This anomaly has been corrected in BASIC 2.

#### Listing 3.6

10 MODE 6

20 REPEAT

30 CLS

40 INPUTLINE a$

50 FOR I%=1 TO 5

60 PROCneatprint(a$,39)

70 NEXT

80 REPEAT UNTIL GET=32

90 UNTIL FALSE

100

110 DEFPROCneatprint(text$,len%)

120 LOCAL i%,next\_char$

130 REM\*ADD SPACE IN CASE OF FOLLOWING TEXT\*

140 text$=text$+" "

150 FOR i%=1 TO LEN(text$)

160 REM\*EXTRACT NEXT CHARACTER\*

170 next\_char$=MID$(text$,i%,1)

180 REM\*IF SPACE DETERMINE WORD LENGTH AND NEWLINE IF NOT ENOUG

H ROOM\*

190 IF next\_char$=" " THEN IF COUNT+FNnext\_wordlen(i$,text$)+1>

len% PRINT:next\_char$=""

200 REM\*DETERMINE LENGTH OF FIRST WORD AND NEWLINE IF NOT ENOUG

H ROOM\*

210 IF i%=1 THEN IF COUNT+FNnext\_wordlen(0,text$)+1>len% PRINT

220 REM\*PRINT NEXT CHARACTER\*

230 PRINT next\_char$

240 NEXT

250 ENDPROC

260

270 DEF FNnext\_wordlen(p%,text$)

280 LOCAL counter%:counter%=p%

290 REM\*DEALS WITH TRAILING SPACES\*

300 IF p%=LEN(text$) THEN =0

310 REPEAT

320 counter%=counter%+1

330 REM\*FIND NEXT SPACE OR END\*

340 UNTIL MID$(text$,counter%+1,1)=" " OR counter%=LEN(text$)

350 =counter%=p%

Listing 3.6 is a general routine for formatting text into any width in such a way that words are not split between lines. The width required is passed to the procedure at line 110 as the parameter len%. The other parameter is the actual text, which will be limited to 255 characters, the maximum string length, but the procedure can be called repeatedly with following text and will simply add it on to the end of what is already there.

Line 140 ensures that each batch of text is separated by a space so each section should be terminated at the end of a word not in the middle of one. The test routine sets the width to 39 characters on a forty column screen. If you attempt to use all forty columns the coding becomes much more elaborate because you have the problem of a hardware linefeed if a word just fits onto the end of a line and so, as already explained, COUNT is not so easily used. Similarly, on an eighty column screen the width should be set to 79. This is a minor limitation for a great simplification to the coding. The routine need not be used to work right across the screen, the line length, passed in len%, can be set to any value, but some odd effects may occur if any words longer than the line length are output.

The hard work is done in the FOR…NEXT loop at lines 150 to 240. The characters from the proposed output are examined one at a time and the start of a new word is recognized by the detection of a space, line 190, or in the special case of the first character to be output, line 210. The length of the next word is calculated by FNnext\_wordlen, at line 270, and then a check carried out to see if there is enough room on the current line for the word, and its preceding space if appropriate. If there is insufficient room a linefeed is output and the space suppressed, since we do not need a space between words on different lines.

The function at line 270 which determines the length of the next word does so by counting along to find either the end of the text to be formatted or the next space, line 240. Line 300 is the only complication and is needed to deal with the situation where the data ends with a trailing space. This would not be found by the remainder of the code in the function since the REPEAT…UNTIL loop must execute at least once and if it is entered with p% equal to LEN(text$) it will go on for ever since the result of line 340 can never be true.

As well as being suitable for controlling text output from a program the routine in Listing 3.6 is also useful for formatting things like the instruction page for a program, without having to spend time putting linefeeds into the text at the right points to avoid running words between lines, another example of making your machine do the donkey work for you.

### Printers

Although it is still one of the more expensive items of computing equipment the dot matrix printer is becoming increasingly common, even on home based computer systems. Indeed, many people buy machines with word processing very much to the forefront of their minds and so a printer may even be given precedence over disk drives. When appropriate, therefore, it is really a very good idea to include hard copy output facilities in your software but there are several important considerations to be aware of, both as regards the habits of different printers and the way in which they can interact with the computer.

The BBC has two possible means of connecting to a printer, either to the parallel port, which is the one under the computer, or via the serial interface, the DIN plug on the back. The Electron currently has only the parallel port on the Plus 1 interface but a plug in module is promised to provide the serial port in the near future. Most people will use the parallel port for their printers and the default condition for the computer is to send all printer output to it. To change to the serial port the command \*FX5,2 has to be issued, parallel output is restored by \*FX5,1. Once selected the setting will remain until the next hard reset, CTRL-BREAK, or power off, of course. Always let your user know, preferably in the documentation with your program, which printer connection is assumed so that he can reset his machine if necessary.

Another thing that you will need to provide information about is whether the computer is going to send linefeeds to the printer. The default situation is that the computer suppresses linefeeds and so only carriage returns will be sent. The reason for this is that many printers automatically carry out a linefeed for each carriage return and so if the computer sent one as well the paper would advance by two steps instead of the required one. Some printers, however, do not insert linefeeds and so will need to receive them from the computer with each carriage return. To arrange this issue the command \*FX6,0 which sets the character to be suppressed by the computer to a null, ASCII code zero. To restore the suppression of linefeeds use \*FX6,10.

Most printer users will have their machines set up to operate correctly with their own printers and will know what FX calls to use if they have an unusual configuration. Most commercial software assumes a parallel printer connection and that linefeeds are not required, but it is a friendly gesture to inform users what your default settings are.

In recent years dot matrix printers have become increasingly intelligent providing many facilities, such as different sizes and densities of typeface and control over line spacing, which are accessed by sending specific control characters to the printer. Unfortunately there is no industry standard for these control codes so it is impossible to write a piece of software which will give full access to all the facilities available on each and every printer. Some specific tailoring of the printer-driving part of any program is necessary if you want to do anything other than straightforward printing in the default mode.

Having said this the control codes used by Epson are rapidly acquiring a universality which approaches standardization and so software configured for Epson printers is often very widely usable, but do bear in mind that not all Epsons, even, have identical capabilities. My advice would be to avoid the use of printer dependent facilities, unless these are absolutely essential to the program, and then your software will work with any printer.

Provided that you observe certain rules there is no need to write separate routines for sending output to the screen and to the printer. The command VDU2 will ensure that all screen output is simply copied to the printer but there may be problems if you are using control codes, especially teletext control characters, which may have unpredictable effects on the printer operation. If you want a routine to work on either the printer or the screen make sure that it does not use any control characters. There is also no point in sending user defined characters to the printer unless it has a special driver to deal with them. Remember, also, that in TAB statements with two parameters, the second parameter, the line number, means nothing to the printer and it will generally be ignored. The printer cannot go back to a previous line to add data as you can on the screen, so output intended for screen or printer should be constructed, exclusively, from top to bottom with no cursor backtracking. The rule is to stick to single parameter TAB statements and linefeeds to get your formatting correct.

The width of the printer paper will normally be at least 80 characters so if you are going to echo a 40 column screen to the printer there may not be complete correspondence. This will arise if some of the newlines on the screen are generated by printing into column 40. Such linefeeds originate in the computer hardware and will not be echoed to the printer. To overcome this the WIDTH pseudo-variable should be set to 40 and the printer formatting will then look like a 40 column screen. Remember to set WIDTH to the default, 0, when you have finished with the printer or your screen output may be in a mess.

#### Listing 3.7

10 MODE 6

20 REPEAT

30 IF FNprinteron PRINT"Printer Ready" ELSE PRINT"No Printer,

printer off line or no power"

40 REPEAT UNTIL GET=32

50 UNTIL FALSE

60

70 DEF FNprinteron

80 \*FX3,10

90 VDU127,127,127

100 \*FX3,0

110 IF ADVAL(-4)<63 THEN =FALSE ELSE =TRUE

If you want to send output to the printer only and not to the screen there are two ways of doing it. The sequence VDU2,21 turns on the printer and then disables the vdu drivers so that the screen is not used. To restore normal, screen only, output use VDU 3,6. Alternatively the command \*FX3,10 will send all output to the printer; things being restored to normal by \*FX3,0. Note that there is a mistake in the *BBC User Guide* in that it is suggested that \*FX3,6 will send all output to the printer. This is not the case. Whenever you are sending output to the printer only, it is most important that you have the error trapping under the control of an ON ERROR statement that will, amongst its other functions, re-enable the screen, so that if an error occurs whilst the printing is in progress, screen output will still be restored.

One of the biggest problems can arise with software which has printer output occurs if a user tries to use a non-existent printer, or perhaps forgets that the printer is not on or is offline. The machine will 'hang' with the printer buffer full and nowhere to go. The only way out is to restore the printer or to hit ESCAPE or BREAK.

If you have suitable trapping on the ESCAPE key this need not be disastrous and we will look at this in Chapter 4, but it is possible to test for the presence of the printer and so issue an error message if it is not there. Listing 3.7 shows a suitable routine. The function, FNprinteron, sends, or attempts to send, three delete characters to the printer. If a printer is not available the characters will only get as far as being trapped in the printer buffer. ADVAL(-4) at line 110, returns the number of free spaces in this buffer enabling our trapped characters to be deleted. If the printer is available the delete characters have no effect upon it since there is not actually anything to be deleted and, of course, the buffer tests as empty. In case you are wondering why we send three deletes the reason is that the check requires at least two characters since the first does not actually stay in the buffer, even if there is no printer, it moves out into the port itself. In the present case, since we are using delete, we actually need three characters since one of them is 'deleted' by its fellow.

#### Listing 3.8

10 MODE 6

20 REM \*SET UP OSWORD CALL 10\*

30 osword%=&FFF1:A%=10

40 REM\*X% AND Y% POINT TO PARAMETER BLOCK\*

50 X%=&70:Y%=0

60 REM\*FIRST BYTE OF PARAMETER BLOCK CONTAINS CODE OR CHARACTER TO BE READ\*

70 ?&70=ASC("A")

80 CALL osword%

90 REM\*RE-DEFINE CHARACTER 255\*

100 VDU23,225,?&71,?&72,?&73,?&74,?&75,?&76,?&77,?&78

110 PRINT CHR$(225)

120 END

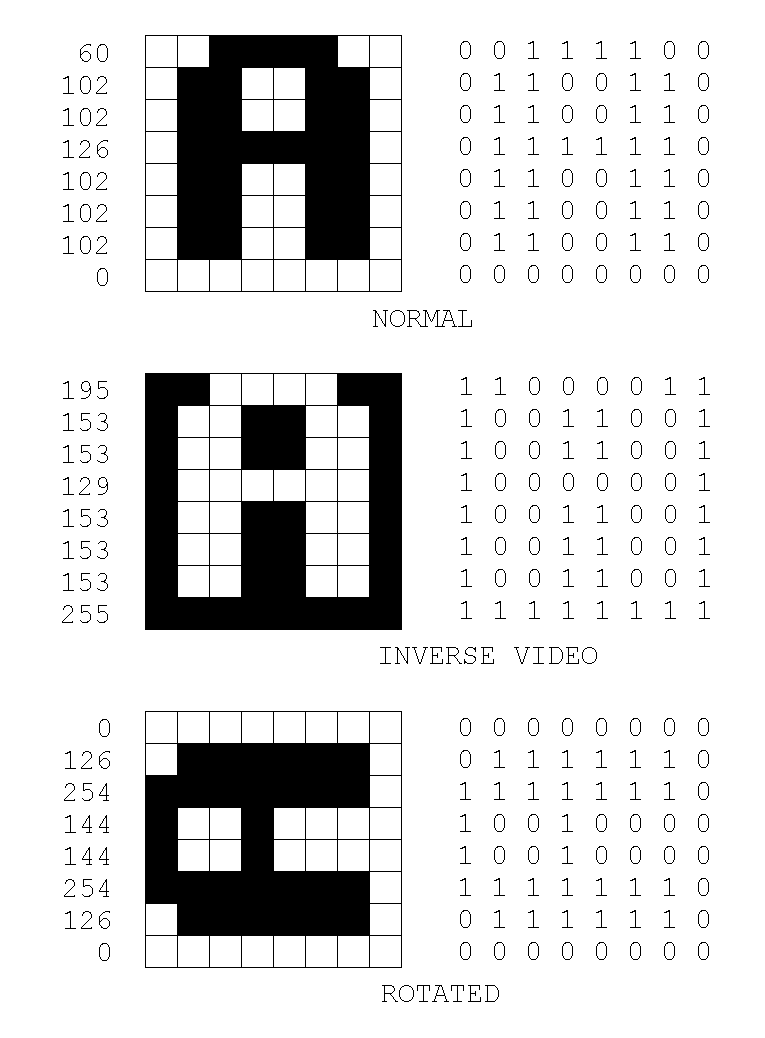
### Manipulating the Character Set

The BBC and the Electron both allow the entire character set to be re-defined by the user with the VDU23 command. Much has already been written on user defined characters and on the topic of utilities to make the task of constructing them easier. I do not proposed to cover this ground again here but, rather, to show you some useful ways of modifying the characters that are already available. This provides a very simple means of producing effects such as inverse video and rotated characters.

This section requires some familiarity with assembly language for a full understanding but I shall endeavour to explain the programming as clearly as possible and would hope to tempt those of you who have not yet grasped the nettle to try your hand at the assembler.

The key to the techniques described here is the use of the operating system call to OSWORD with 10 in the accumulator which is used to read the dot pattern of a character; in other words, to retrieve the values that you would have to put into the VDU23 statement to define the character in the first place. The best way to understand the call and to appreciate its value, is to look at an example of its use from BASIC before turning to the assembler version. Listing 3.8 is such an example.

The OSWORD call requires a parameter block to be set up from which it can draw information and into which it returns its results. When the call is made the X and Y registers must contain the address of this block, with the most significant byte in the Y register. In the example in Listing 3.8 I have placed the parameter block on zero page starting at &70 and the pointers are set up in line 50. The character whose dot pattern is to be retrieved must have its ASCII code placed in the first byte of the parameter block, in this case &70. Line 70 does this. The call is then made at line 80 and returns the dot pattern, as the eight VDU23 numbers, into bytes 2 to 9 of the parameter block. Line 100 retrieves the values and uses them to define character 255. The effect, in this trivial example, is to copy the definition of 'A' into character 225.



##### **Figure 3.1.** Relationships between Normal, Inverse and Rotated Characters

Figure 3.1 shows the relationship between the dot matrix definitions of the normal form of the character 'A' and its inverse video and rotated equivalents. Let us look at the inverse video case first. This is fairly simple, all that is required is to take each byte in the VDU23 definition and to invert it, in the sense that all the ones in the binary number become zeros and vice versa. The operation is achieved by taking each byte and exclusive ORing it with the binary number %11111111, which is hexadecimal &FF. Any character can thus be converted to inverse video by retrieving its dot definition and transforming each byte in this way. All that is then needed is to define a new character using these transformed values in the VDU23 statement, and to print this newly defined character instead of the original.

#### Listing 3.9

10 MODE 6

20 DIM code% 50

30 PROCassemble

40 REPEAT

50 CLS

60 INPUTLINE test$

70 PROCprint\_inverted(text$,5,10)

80 REPEAT UNTIL GET$=" "

90 UNTIL FALSE

100

110 DEFPROCprint\_inverted(text$,x%,y%)

120 LOCAL i%

130 PRINTTAB(x%,y%);

140 FOR i%=1 TO LEN(text$)

150 A%=ASC(MID$(text$,i%,1)):CALL invert

160 PRINT CHR$(225);

170 NEXT

180 PRINT

190 ENDPROC

200

210 DEFPROCassemble

220 osword%=&FFF1:oswrch%=&FFEE

230 P%=code%

240 [

250 .invert

260 STA &70

270 LDX #&70

280 LDY #0

290 LDA #10

300 JSR osword% \Write character definition to zero page

310 LDX #&71 \X points to start of definition

320 .loop

330 LDA #&FF

340 EOR &0,X \Invert bits

350 STA 0,X

360 INX \Increase pointer

370 CPX #&79 \Are all bytes done ?

380 BNE loop \If no loop back

390 LDA #23:JSR oswrch% \Otherwise define CHR$(225) as invert v

ersion

400 LDA #225:JST oswrch%

410 LDX #&71

420 .loop2

430 LDA 0,X:JSR oswrch%

440 INX

450 CPX #&79

460 BNE loop2

470 RTS

480 ]

490 ENDPROC

Listing 3.9 shows this in action. The string of characters to be printed in inverse video is passed to the procedure at line 110 as text$ and the characters are extracted from it one at a time, at line 150. The call to the invert routine results in the retrieval of the definition of each character by OSWORD 10 and the dot matrix is then inverted and copied into character 255, which is then printed, at line 160.

I have carried out the transformations and definition of CHR$(225) in machine code, since the user of BASIC drastically slows down printing. The code is assembled starting at line 210. The ASCII code of the character to be inverted is passed to the routine in the accumulator so that first job is to store this at the start of the OSWORD parameter block, line 260 does this. Again, zero page is used for the parameter block. Lines 270 to 300 set up the pointers and call OSWORD 10, exactly as in the BASIC example and then the X register is set to point to the first byte of the character definition which is now on zero page. The loop between lines 320 and 380 then words its way along the definition EORing each byte with &FF. Lines 370 and 380 check for the end of the loop, and once the exit is made, lines 390 to 460 define character 225 using the new parameters. The operating system OSWRCH is the machine code equivalent of VDU. Loading the accumulator with the number n and then calling OSWRCH is exactly equivalent to VDUn.

Rotation of characters can be achieved in a similar manner, although the assembler code is a little more complex. Refer again to Figure 3.1 and look at the rotated version of 'A' in comparison with the normal character. The operation required now is to rotate the dot matrix definition so that line 1 in the normal character becomes column 1 in the rotated version and line 2 becomes column 2 etc. In order to do this we have to manipulate individual bits of the character definition numbers since, for example, line 1 of the rotated definition is actually constructed from the least significant bits of each of the VDU23 numbers in the normal character. There is luckily, a machine code instruction that enables us to do this, called 'rotate left, accumulator'; its assembler mnemonic is ROL. When this instruction is applied the most significant bit of the accumulator is moved into the carry flag and all the remaining bits move one place to the left, the now vacant least significant bit being loaded form the original contents of the carry flag.

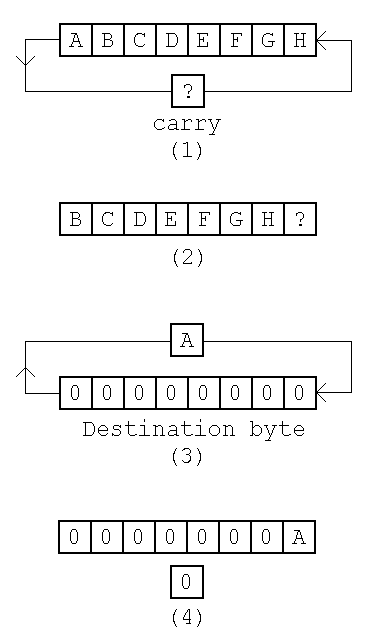
This is illustrated diagrammatically in Figure 3.2. Position (1) represents the original condition of the accumulator and the carry, and position (2) the situation after the execution of ROL. The most significant bit of the source byte has effectively been moved to the carry flag. To copy this bit into some destination byte we simply load the byte into the accumulator and apply ROL again, as represented in part(3) of Figure 3.2, the resulting situation being as shown at (4). The net effect is to move the most significant bit of the source byte, the bit marked A, into the least significant bit of the destination byte. We can now go on and build up the destination byte by repeated operations, rotating each bit into it from the carry flag, and, at the same time, moving the destination byte one bit to the left. After eight such operations the byte will be fully constructed with the bit marked 'A' in state (4) of Figure 3.2 now being the most significant bit.

Figure 3.3 shows the sequence required to transform the first column of a character matrix into a row, by a succession of ROL instructions. The operation represented in Figure 3.3 will need to be executed eight times to transform the entire character matrix, once for each byte.

Listing 3.10 is a complete utility routine for rotating characters. The simple single rotation can be followed by further rotations of CHR$(225) to provide inverted printing and printing rotated through 270 degrees. The inverted characters are something of a curiosity but the partially rotated ones are ideal for labels on graphs and diagrams.

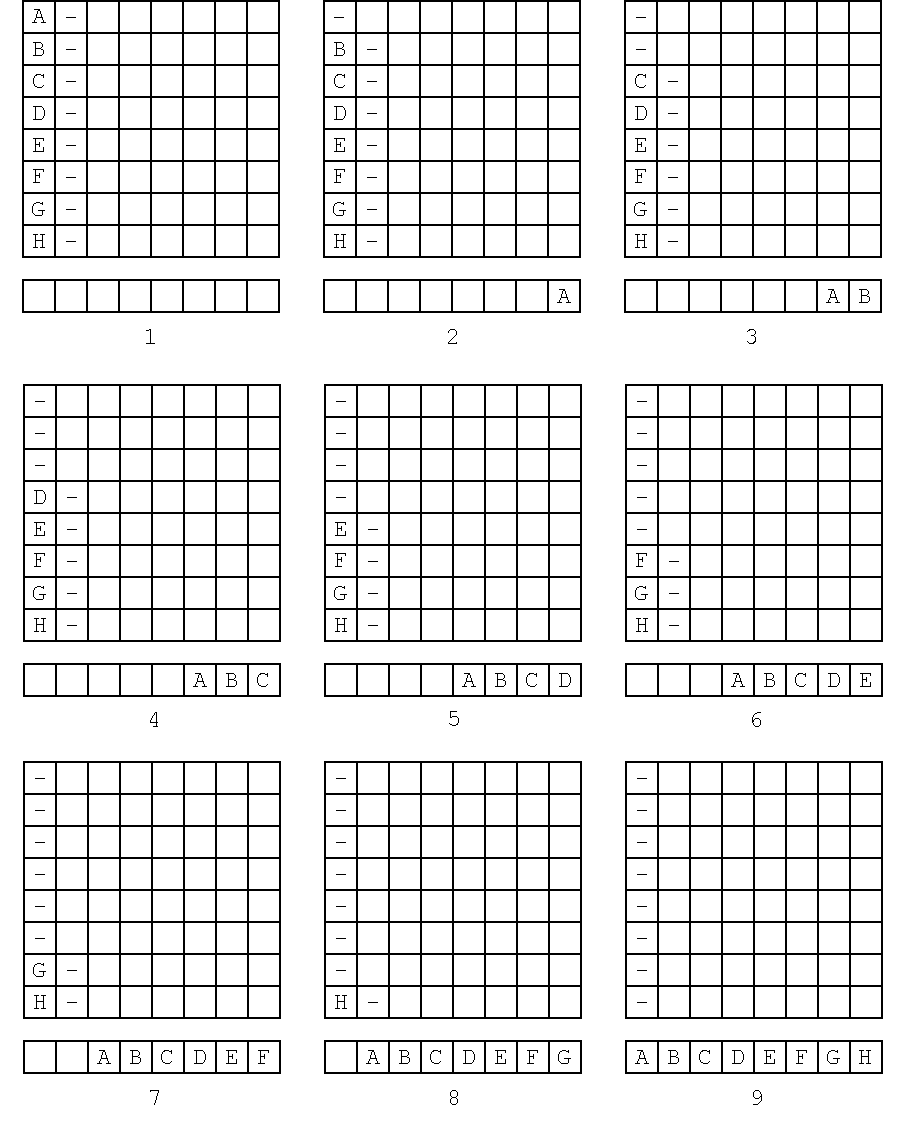
Another useful modification to the character set is illustrated in listing 3.11. Here each character is underlined before being output. This is simply achieved by making the last parameter of the VDU23 definition 255, &FF. It is usually zero in the case of the on-board characters. The necessary code is at lines 310 and 320.

Listing 3.12 takes the idea of manipulating the character set a little further, allowing the printing of double height characters. These are particularly useful in the 20 column modes, 2 and 5, since the normal character set used here is rather squat and chunky for good legibility. A little vertical stretching works wonders!



##### **Figure 3.2.** Operations of ROL

The trick used in listing 3.12 is to take the top half of the character definition and double up the lines of dots which can then be used as the definition of CHR$(225). A similar operation defines CHR$(226) as the double height bottom half of the character. Figure 3.4 illustrates the point diagrammatically. To print the expanded character we merely print CHR$(225) and CHR$(226) on subsequent screen lines, but at the same horizontal TAB position. Line 160 does this.



##### **Figure 3.3.** Transformation of Column to Row by Succession of ROL Operations

None of the character set manipulations that I have been discussing will work in mode 7, of course, since user-defined characters are not then available. The double height routine will also be unsatisfactory in the text only modes, 3 and 6, since there is a space between the screen lines and the two halves of the character will not be joined up. The only other point to be made about these routines is that they are not restricted to the on-board character set, they will work equally well with characters that you have defined for yourself.

#### Listing 3.10

10 MODE 4

20 DIM code% &60

30 PROCassemble

40 REPEAT CLS

50 INPUTLINE text$

60 PROCprint\_at\_angle(90,text$,0,31)

70 PROCprint\_at\_angle(180,text$,38,31)

80 PROCprint\_at\_angle(270,text$,38,0)

90 REPEAT UNTIL GET$=" "

100 UNTIL FALSE

110

120 DEFPROCprint\_at\_angle(ang%,text$,x%,y%)

130 LOCAL i%

140 ang%=ang% DIV 90

150 FOR i%=1 TO LEN(text$)

160 A%=ASC(MID$(text%,i%,1))

170 REM\*ROTATE CHARACTER 90 DEG ANTI-CLOCKWISE\*

180 CALL rotate

190 REM\*PRINT IF REQUIRED ROTATION 90\*

200 IF ang%=1 PRINT TAB(x%,y%-i%+1)CHR$(225);

210 REM\*ROTATE FURTHER 90 DEG AND PRINT\*

220 IF ang%=2 A%=225:CALL rotate:PRINTTAB(x%-i%+1,y%)CHR$(225);

230 REM\*ROTATE ANOTHER 90 DEG AND PRINT\*

240 IF ang%=3 A%=225:CALL rotate:PRINTTAB(x%,y%+i%-1)CHR$(225);

250 NEXT

260 ENDPROC

270

280 DEFPROCassemble

290 oswrch%=&FFEE:osword%=&FFF1

300 P%=code%

310 [

320 .rotate

330 STA &70 \Character code for osword

340 LDX #&70

350 LDY #0 \Point to osword parameter block

360 LDA #10 \Osword call number 10

370 JSR osword% \read character definition

380 LDA #&71:STA &79 \Contents of &79 point to source byte

390 LDA #&88:STA &89 \Contents of &89 point to destination byte

400 .loop1

410 LDX &79

420 ROL 0,X \Bit 7 of source into carry

430 LDX &89

440 ROL 0,X \Carry into bit 0 of destination

450 LDX &79

460 INX \Increment X

470 STX &79

480 CPX #&79 \Have all eight source bytes been done ?

490 BNE loop1 \If not loop back

500 LDA #&71:STA &79 \If yes reset source pointer to first byte

510 LDY &89 \Recover destination byte pointer

520 DEY \Decrement it

530 STY &89 \Store it

540 CPY #&80 \Have every bit been done ?

550 BNE loop1 \If not loop

560 LDA #23:JSR oswrch% \If yes define CHR$(225) using rotated

bytes

570 LDA #225:JSR oswrch%

580 LDX #&81

590 .loop2

600 LDA 0,X:JSR oswrch%

610 INX

620 CPX #&89

630 BNE loop2

640 RTS \Back to BASIC

650 ]

660 ENDPROC

#### Listing 3.11

10 MODE 6

20 DIM code% 50

30 PROCassemble

40 REPEAT

50 CLS

60 INPUTLINE text$

70 PROCprint\_underlined(text$,5,10)

80 REPEAT UNTIL GET$=" "

90 UNTIL FALSE

100

110 DEFPROCprint\_underlined(text$,x%,y%)

120 LOCAL i%

130 PRINTTAB(x%,y%);

140 FOR i%=1 TO LEN(text$)

150 A%=ASC(MID$(text$,i%,1)):CALL under

160 PRINT CHR$(225);

170 NEXT

180 PRINT

190 ENDPROC

200

210 DEFPROCassemble

220 osword%=&FFF1:oswrch%=&FFEE

230 P%=code%

240 [

250 .under

260 STA &70

270 LDX #&70

280 LDY #0

290 LDA #10

300 JSR osword% \Write character definition to zero page

310 LDA #&FF

320 STA &78 \Insert underline

330 LDA #23:JSR oswrch% \Define CHR$(225) as underlined version

340 LDA #225:JSR oswrch%

350 LDX #&71

360 .loop

370 LDA 0,X:JSR oswrch%

380 INX

390 CPX #&79

400 BNE loop

410 RTS

420 ]

430 ENDPROC

### Advanced Mode 7 Colour Control

The teletext mode on the BBC is extremely useful since it allows extensive multi-coloured effects whilst consuming only 1 Kilobyte of memory. If, however, you want to do something like have an entire screen with a white background and blue text every line has to start with the appropriate teletext control characters and these must be re-printed each time the screen is cleared. This is in marked contrast to the other modes where the text and background colours can be set by a couple of simple COLOUR commands. There is a way of achieving a similar effect in mode 7, however, by using an event driven routine to put the required control characters into the screen memory at every screen scan. Since there are 50 scans per second this means that the characters are, effectively, always there. Even if they get overwritten they will be re-established within 1/50th of a second.

An event is a software interrupt which signals that something has taken place in the machine. Such events are normally ignored, but they can be enabled with the \*FX14 call. The screen scan start event is number 4 so \*FX14,4 will enable it to be recognised. When an event is enabled, every time it occurs the processor indirects through a vector at &220. Normally this just points to a return instruction and so nothing happens, but if this vector is changed to point at a machine code routine which we have provided, that routine will run every time that the event occurs.

In Listing 3.13 I have used the event which is triggered by the start of a screen scan to call a routine which pokes a set of control characters into the mode 7 screen memory. These are direct pokes, so the routine is not second processor compatible, but this is irrelevant since events are only available in the I/O processor anyway. The routine will only work on the Model B Plus if the screen memory is in main RAM rather than the shadow RAM, since the pokes can only be made into the main memory.

In Listing 3.13, I have set the routine up to give a white background with blue text. After the program has been run the screen will retain this configuration, even after a CLS, until you issue the command \*FX13,4, either directly or within a program. This disables the event and stops the stream of control characters. Break will also switch the effect off since it resets the vector and disables all events.

#### Listing 3.12

10 MODE 5

20 DIM code% 100

30 PROCassemble

40 REPEAT

50 CLS

60 INPUTLINE text$

70 PROCprint\_enlarged(text$,0,10)

80 REPEAT UNTIL GET$=" "

90 UNTIL FALSE

100

110 DEFPROCprint\_enlarged(text$,x%,y%)

120 LOCAL i%

130 PRINTTAB(x%,y%);

140 FOR i%=1 TO LEN(text$)

150 A%=ASC(MID$(text$,i%,1)):CALL expand

160 PRINT CHR$(225);CHR$(10);CHR$(8);CHR$(226);CHR$(11);

170 NEXT

180 PRINT

190 ENDPROC

200

210 DEFPROCassemble

220 osword%=&FFF1:oswrch%=&FFEE

230 P%=code%

240 [

250 .expand

260 STA &70

270 LDX #&70

280 LDY #0

290 LDA #10

300 JSR osword% \Write character definition to zero page

310 LDX #&71 \X points to start of definition

320 \DEFINE CHR$(225) AS TOP HALF

330 LDA #23:JSR oswrch%

340 LDA #225:JSR oswrch%

350 .loop1

360 LDA 0,X:JSR oswrch%

370 LDA 0,X:JSR oswrch%

380 INX

390 CPX #&75

400 BNE loop1

410 \DEFINE CHR$(226) AS BOTTOM HALF

420 LDA #23:JSR oswrch%

430 LDA #226:JSR oswrch%

440 LDX #&75

450 .loop2

460 LDA 0,X:JSR oswrch%

470 LDA 0,X:JSR oswrch%

480 INX

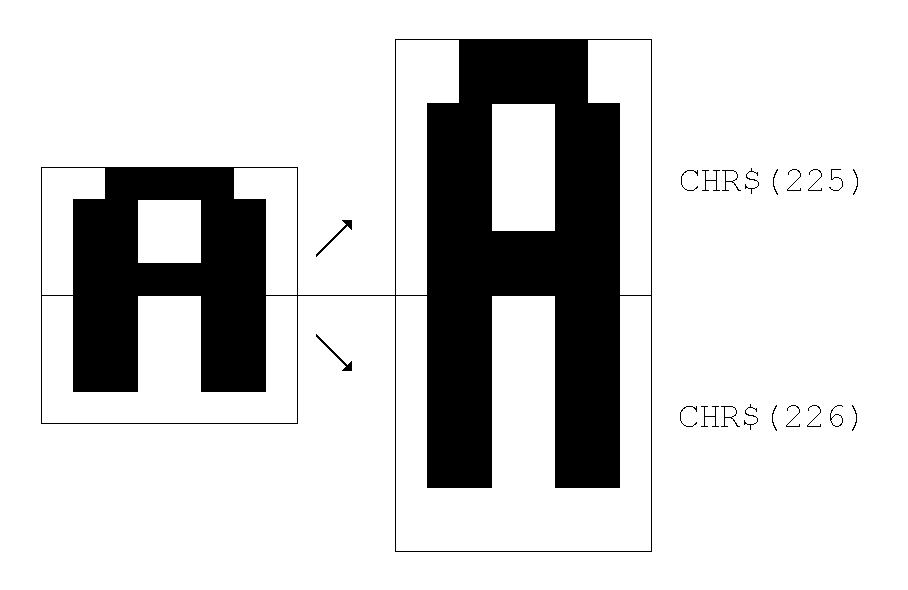
490 CPX #&79

500 BNE loop2

510 RTS

520 ]

530 ENDPROC



##### **Figure 3.4.** Double Height Character constructed from two User Defined Characters

#### Listing 3.13

10 MODE 7

20 DIM code% 255

30 bckgd%=135:REM\*WHITE BACKGROUND\*

40 text%=132:REM\*BLUE TEXT\*

50 PROCassemble(bckgd%,text%)

60 CLS

70 REM\*ENABLE SYNCH START EVENT\*

80 \*FX14,4

90 PRINTTAB(3,10)"FULL COLOUR IN MODE 7"

100 END

110 DEFPROCassemble(bckgd%,textcol%)

120 LOCAL opt%

130 IF bckgd%=0 newback%=0- ELSE newback%=157

140 ?&72=?&220:?&73=?&221:REM\*PRESERVE OLD EVENT VECTOR\*

150 REM\*POINT EVENT VECTOR AT NEW ROUTINE\*

160 ?&220=code% MOD 256:?&221=code% DIV 256

170 FOR opt%=0 TO 2 STEP 2

180 P%=code%

190 [

200 OPT opt%

210 STA &70

220 PHA

230 TXA

240 PHA

250 TYA

260 PHA

270 PHP \preserve Registers

280 LDA &70

290 CMP #4

300 BNE exit

310 LDA #&0

320 STA &70

330 LDA #&7C

340 STA &71 \Screen Start Address

350 LDX #25 \No of Screen Lines

360 .LOOP

370 LDA #bckgd%

380 LDY #0

390 STA(&70),Y \Background Colour at TAB(0)

400 LDA #newback%

410 INY

420 STA(&70),Y \Newbackground code at TAB(1)

430 LDA #textcol%

440 INY

450 STA(&70),Y \Text colour at TAB(2)

460 CLC

470 LDA &70

480 ADC #40

490 STA &70

500 LDA &71

510 ADC #0

520 STA &71 \Add 40 to starting address to go to next line on s

creen

530 DEX \Decrease loop counter

540 TXA

550 BNE LOOP \go back again if loop counter not zero

560 .exit

570 PLP

580 PLA

590 TAY

600 PLA

610 TAX

620 PLA \Restore registers

630 JMP(&72) \return to normal event handling

640 ]

650 NEXT

660 ENDPROC

There are a couple of limitations with this facility. The first three tab positions on the screen will not be usable, anything appearing there will be obliterated at the next screen scan by the control characters, so any output must appear at TAB(3) onwards. The other limitation is that the screen must not be scrolled. If this happens you will get some rather odd effects. The reason is that, because of the way the BBC handles scrolls, the screen start address has changed.

The routine in Listing 3.13 can be modified to give any colour combination. All you have to do is to change the value of bckgd% and text% to the required teletext colour character codes. To retain the default, black, background set bckgd% equal to zero. You can even make text% a graphics colour control code, this will make any lower case letter or number sent to the screen appear as its graphics character equivalent.

### Conclusion

This chapter has covered a wide variety of techniques for handling output on the BBC and the Electron. It is up to you to select what is needed for the task in hand. The main thing to remember is, as always, to keep your users fully in the picture as to what is being displayed and where they are to go next.

# Chapter 4.

# Error Trapping and Avoidance

Whatever steps are taken to avoid a program running into errors, the possibility that this may occasionally happen always exists. At best Murphy's Law is ever present, as is the first law of computer programming which states that "there is always one more bug".

However hard you try to foresee possible problems you will almost certainly overlook something so it is essential to ensure that errors which do arise will not break the program. The most probable source of errors is the filing system since there are any number of ways that things can go wrong here, most of them completely outside the control of the original programmer. The user may load the wrong disc or tape, making a required data file impossible to find or load, a disc may be corrupted or unformatted. Well-constructed error trapping will not solve such problems but it should ensure that the user is told about the error in an informative manner and that the error does not abort the program and lose any data. BBC BASIC has an excellent mechanism for doing this, the ON ERROR command.

Normally, when an error occurs whilst a BASIC program is running, whether the error be from the program or the filing system, execution will stop and an error message will be generated. If an ON ERROR has been invoked, however, the code following the statement will be executed. ON ERROR RUN would be the simplest example and would simply ignore all errors and restart the program; not really a particularly helpful response. Far more sophisticated responses are possible since the error number and the line in which it occurred are available to us, in the pseudo-variables ERR and ERL respectively, so we can arrange to deal with errors intelligently and selectively.

The most generally useful type of ON ERROR command is something like:

10 ON ERROR CLS:REPORT:E$=INKEY$(1000):GOTO300

Report gives the normal error message, but without the line number information since this is of little significance to a user, and then the INKEY$ function causes a wait for ten seconds, or until a key is pressed, for the message to be digested. The GOTO should send execution to a safe point in the program, in menu driven software this is most conveniently the main menu. The user should be able to pick things up again from there.

There are, however, some limitations on where the GOTO can point. You cannot jump into a procedure, function or subroutine, even if the error occurred inside one of these structures. The error clears the stack so there is no longer a return address. Nor can you jump into the middle of a REPEAT...UNTIL or FOR...NEXT loop from ON ERROR...GOTO any more than you could with an ordinary GOTO. The GOTO must always send you into the main part of the program and for this reason alone it is a good idea to operate things like your main menu within the primary control structure and not in a subroutine of any kind. Listing 4.1 shows the sort of thing that is required in a simple menu-drive program such as that illustrated in Chapter 1.

#### Listing 4.1

10 MODE 7

15 ON ERROR CLS:REPORT:E$=INKEY$(1000):GOTO20

20 REPEAT CLS

30 PRINT''TAB(12)"\*MAIN MENU\*"''

40 PRINTTAB(5)"1. ENTER NEW DATA"'

50 PRINTTAB(5)"2. VIEW EXISTING DATA"'

60 PRINTTAB(5)"3. DELETE CARDS"'

70 PRINTTAB(5)"4. SEARCH DATA"'

80 PRINTTAB(5)"5. SORT DATA"'

90 PRINTTAB(5)"6. SAVE DATA TO TAPE/DISC"'

100 PRINTTAB(5)"7. END PROGRAM"'

110 PRINTTAB(5,20)"Please enter a choice "+CHR$(8);

120 VDU23,1,1;0;0;0;

130 REPEAT \*15,1

140 A$=GET$:IF A$<"1" A$=CHR$(ASC(A$)+16)

150 UNTIL INSTR("1234567",A$)

160 PRINT A$;:VDU23,1,0;0;0;0;

170 REPEAT \*FX15,1

180 B$=GET$

190 UNTIL B$=CHR$(127) THEN 110

210 IF A$="1" PROCenter

220 IF A$="2" PROCview

230 IF A$="3" PROCdelete

240 IF A$="4" PROCsearch

250 IF A$="5" PROCsort

250 IF A$="6" PROCsave

270 UNTIL A$="7"

280 MODE 7

290 END

Line 15 always sends execution back to the top of the main menu, at line 20. If you enter the program fragment and then select any menu option but number 7 you will get a 'No such FN/PROC' error, but the program will not end, you will be told about the error and then the menu will come back.

If you press the ESCAPE key this will also be treated as an error and reported in the same way. Escape is regarded as error number 17 in BBC BASIC and this opens up the possibility of using the escape key in a very powerful manner, to return to the main control part of a program from anywhere. The escape key is continuously scanned by the operating system and so can generate error 17 at any time. Line 15 would provide this function, as it stands, but we do not really want to be told that we pressed ESCAPE, we just want the computer to get on with the job and put the menu back. Changing line 15 to:

15 ON ERROR IF ERR=17 THEN 20 ELSE REPORT:E$=INKEY$(1000):GOTO20

will achieve this.

Once this facility is set up, all that you have to do is to tell the user that ESCAPE will restore the menu at any point in the program. This is most useful since it anyone gets lost, or in a tangle, say by selecting the wrong menu option, ESCAPE is now a panic button!

If you are operating this technique in a program that needs a lot of numerical input the closeness of the escape key to '1' can be a nuisance so it is a good idea to move the escape function to some other key. This is done with the operating system command \*FX220,n where n is the ASCII code of the key which is to generate the escape. \*FX220,64 makes @ the escape key, for example. The normal situation is restored by \*FX220,27.

This ability to move the escape key can be very useful in educational programs where you might want the teacher to have access to the escape facilities but to hide them from the pupils, who may know a bit about the keyboard. Try something along the lines of \*FX220,0 which will make the escape key CTRL/@.

The use of the escape key in this way is by no means restricted to menu-driven programs. So long as there is a point in the program where it can be re-entered safely then you can use the method. You will need to provide such a point, anyway, under most circumstances, to allow the real error trapping part of ON ERROR to operate properly. The command-driven program in Listing 4.1 could safely be entered at line 90 and so the ON ERROR trap could point there.

Whenever you are using an ON ERROR statement it is essential that it is truly capable of dealing with any set of circumstances that might prevail when the error occurs. Any open files should be closed otherwise there is a danger of data loss, the screen must be restored if you have any routines that are outputting to the printer with the screen off and any keyboard functions which have been modified, such as the effect of the cursor keys, should be restored. If you do not do this properly you may find the menu coming out on the printer rather than the screen or that there is no cursor editing when you next need it, or some other such alarming happening. If you need to do a lot of tidying up you will probably not be able to have a simple, self-contained ON ERROR line such as I have illustrated, but you will need something like:

15 ON ERROR IF ERR=17 THEN 30 ELSE GOTO 1000

The detailed error trapping and tidying up would then be written as a sequence of lines starting at 1000 and terminating with GOTO 30.

You should not insert the final error trapping into your software until it is running satisfactorily, after all you want to be able to find, and correct, errors at the development stage. If, however, you are using the escape key as I have suggested you can get the correct response by having, in your developing program, a line such as:

15 ON ERROR IF ERR=17 THEN 30 ELSE REPORT:PRINT" at line ";ERL

:END

This will intercept only escape and give the normal error reports for other problems.

The use of escape in this way is only one example of the constructive use of a deliberately generated error, with careful coding other errors can be used in a similarly useful manner as we shall see in later chapters.

### Changing Error Messages

One very useful job that the ON ERROR statement allows you to do is to change the actual message which is printed when an error occurs so as to make it much more informative for the inexperienced user. For example, if you have a program which requires the user to enter the name of a disc data file that is to be loaded and a filename which is not represented on the disc is entered then the error message Channel will be displayed by the disc filing system. This is not really very informative but the number of this error is &DE so if you include in your error trapping a line such as:

15 ON ERROR IF ERR=&DE PRINT "No such file on this disc":E$=IN

KEY$(1000):GOTO300 ELSE ... rest of error trap

then the exact problem is specified and the user is much more fully informed. Channel is a particularly obscure error message but it is worth considering expanding any error message that has a high probability of being triggered into a more informative version. Too Large and Division by Zero may well occur in number-crunching programs and could usefully be treated in this way.

### Fatal Errors

Not every error can be trapped by ON ERROR, those with error number zero are designated 'Fatal errors' and will always abort the program. The only such error which can arise at present in BBC BASIC whilst the program is running is 'No Room', signifying that there is no more memory left. You would hope, of course, that a well written and tested program would never have this problem but there is one set of circumstances where it can creep up unexpectedly. This arises from a rather bad habit of BBC BASIC in that it is rather untidy in the way it deals with string variables.

When a string is first declared the BASIC sets aside storage space for it, roughly equal to the number of characters, a few extra bytes are, in fact, required as pointers and some scope for expansion may be left. If the contents of the string are changed by the program at any time, provided that the new contents are not substantially longer than the old, all will be well. If the new contents *are* longer than the old by any great amount, however, the string variable storage is moved to a new location in memory so as to allow the expansion, but, and here is the problem, the old location is abandoned completely and can never be re-used by the program. Any routine, therefore, that involves a large amount of string editing could consume enormous amounts of memory and in a quite unpredictable manner. A user might keep continually increasing the length of a string in small steps, for example, eating away great chunks of memory each time, especially when the string becomes long.

To be fair I have deliberately exaggerated the problem here; it is not so bad in practice and has been somewhat alleviated by modified coding in BASIC 2 as compared to BASIC 1, but it is always to be borne in mind when a lost of user editing of string variables is envisaged. Luckily there is a simple solution. Always ensure when a string variable is first declared, that it is set to the greatest length that it is ever likely to have, even if it has to be padded with spaces. It is certainly good practice when declaring a large string array to fill it up with spaces straight away. This will ensure that the string storage location never has to be moved and you will not be faced with the wrath of a user who spends a few hours carefully entering and editing a word list for a spelling program only to suddenly hit a No Room error and lose all his data!

As numerical variables always occupy the same amount of memory, irrespective of the value which they contain, they do not suffer from similar problems.

# Chapter 5.

# Business and Finance

The microcomputer made its first impact in the world of science and engineering but the next area into which it is moving is the office. Arguably its effect here will be far more revolutionary since at least the scientist and then engineer already made extensive use of mainframe computers; the micro introduced only a change in emphasis, and often an increase in convenience, the use of computers in business circles, however, was generally to be found rather remote from the "shop floor". The BBC is quite a popular business micro, especially when equipped with the Z80 second processor, which runs under CP/M and so gives access to the massive amount of business software available on this popular software bus as well as providing the powerful programming facilities of BBC BASIC. The ABC series of computers take over where the BBC leaves off and so extend the possible range of facilities even further. The Electron, with its more limited capabilities, is not so likely to be found in the office but it can, nevertheless, serve very well for many business applications, and home accounting and data storage.

In this chapter I want to discuss some techniques which are of special relevance in developing programs for use in the financial and business environment. In these areas we have to be especially concerned with good input validation and the whole question of the security of data held in computers rears its head.

### Computers in Accounting

The microcomputer would seem to be an ideal tool to take over much of the drudgery in all branches of accounting and this is indeed the case, but great care must be taken with the software or numerical results can be wildly imprecise. This may seem a strange statement to make since we normally regard the computer as a model number cruncher. Indeed it is for the scientist and engineer but the accountant requires a degree of arithmetical precision that would be irrelevant in the other two disciplines and the floating point arithmetic of most BASICs as implemented on micros is not good enough. The reason lies in the number of significant figures used to store real numbers (in BBC BASIC nine are used). The lack of precision may well increase as calculations are performed with the data since each step involves manipulation and storage of intermediate results, still to nine significant figures, and so each stage may introduce a rounding error. This is not a problem for many scientific purposes since the experimental errors involved in the determination of the data are usually a bigger source of inaccuracy than the computer, although if very large or very small numbers are being dealt with the situation may not be so happy.

#### Listing 5.1

10 MODE 6

20 FOR y%=0 TO 21 STEP 3

30 price%=FNgetprice(0,y%,"Enter the price ")

40 PRINT'"You entered ";FNmoneyout(price%)

50 NEXT

60 PRINT'" \*PRESS THE SPACE BAR TO CONTINUE\*";

70 REPEAT UNTIL GET$=" ":RUN

80

90 DEF FNgetprice(x%,y%,prompt$)

100 LOCAL a$,in$:in$=" "

110 REPEAT

120 REPEAT

130 PRINTTAB(x%,y%);SPC(40);TAB(x%,y%);prompt$+" ";in$;

140 \*FX15,1

150 a$=GET$

160 REM\*RESPOND ONLY TO VALID CHARACTERS\*

170 UNTIL INSTR(CHR$(13)+CHR$(127)+".0123456789",a$)

180 REM\*ONLY ONE DECIMAL POINT\*

190 IF a$="." THEN IF INSTR(in$,".") a$=""

200 REM\*ONLY TWO CHARACTERS AFTER DECIMAL POINT\*

210 IF INSTR(in$,".") THEN IF a$<>CHR$(13) AND a$<>CHR$(127) AN

D (LEN(in$)-INSTR(in$,"."))=2 a$=""

220 REM\*ADD TO STRING IF NOT DELETE OR CR AND IF in$ NOT TOO LO

NG\*

230 IF a$<>CHR$(127) AND a$<>CHR$(13) AND LEN(in$)<12 in$=in$+a

$

240 REM\*RESPOND TO DELETE\*

250 IF a$=CHR$(127) in$=LEFT$(in$,LEN(in$)-1):IF in$="" in$=" "

260 REM\*EXIT ON RETURN,IF PENCE COMPLETED, AND NOT TOO BIG\*

270 UNTIL a$=CHR$(13) AND INSTR(in$,".")=LEN(in$)-2 AND VAL(in$

)<21374836.48

280 =VAL(LEFT$(in$,LEN(in$)-3)+RIGHT$(in$,2))

290

300 DEFFNmoneyout

310 LOCAL cash$

320 cash$=STR$(amount%)

330 REM\*ADD LEADING ZEROS IF PENCE ONLY\*

340 IF LEN(cash$)<3 THEN REPEAT cash$="0"+cash$:UNTIL LEN(cash$

)=3

350 REM\*INSERT DECIMAL POINT\*

360 =LEFT$(cash$,LEN(cash$)-2)+"."+RIGHT$(cash$,2)

Accountants, however, must work to the nearest penny and there are, of course, no errors at all in their original data, so nine significant figures my not be good enough and the cumulative inaccuracies that can build up in calculations with real numbers can easily exceed tolerable limits. The requirements are for arithmetic to be done precisely and for the results and the intermediate values to be stored precisely. The best way of doing this is to stick to integer variables and integer arithmetic. Since BBC BASIC supports four byte signed integers this allows financial calculations to be done to an accuracy of one penny in twenty million pounds, or thereabouts. If we are going to handle sums of money as integers then we must convert all our input into pence, and, after the calculations, re-insert the decimal point for output.

Listing 5.1 shows how both of these can be achieved. The function, FNgetprice, at line 90 takes input in the form of pounds followed by two decimal places, representing the pence. Only numbers and the decimal point are accepted as input, all other keys are ignored and two figures must be typed after the decimal point, to complete the pence column, otherwise the input cannot be made. Line 270 makes the necessary checks and also ensures that the value entered is not so large that, when converted to pence, it will produce a 'Too Big' error. Line 280 eliminates the decimal point and returns the sum entered in pence. Notice that line 280 uses no arithmetic, the decimal point is removed purely by string operations, and so there is no danger of introducing a rounding error.

For output we use the function at line 300 which re-inserts the decimal point, again by purely string manipulations, and adds leading zeroes to sums of less than one pound so the output will look neat.

Additions, subtractions and multiplications of integer sums of money can be carried out in a straightforward manner and there are no worries about rounding errors, but division requires a little more thought. Division will be required for such operations as calculations of taxation and profit margins. To illustrate the possible pitfalls I am going to look at the calculation of VAT.

This may seem deceptively simple. All this is required, after all, is to calculate 15% of the pre-VAT price, but we want, as far as possible, to use only integer arithmetic so as to escape the rounding errors. There are bound to be results which include fractions of a penny and we shall have to be particularly careful how we handle these since, ultimately, we shall have to round to the nearest penny. The best way is for our own software to take complete charge of the rounding, rather than leaving it to the computer's arithmetic or to the @% output formatting variable.

#### Listing 5.2

10 MODE 6

20 REPEAT

30 INPUTLINE"Enter the price before VAT "P%

40 PROCvat(15,P%)

50 PRINT "VAT-whole pence=";vat%'"Fraction of pence=";correct

60 UNTIL FALSE

70

80 DEFPROCvat(rate%,price%)

90 vat%=(price% DIV 100)\*rate%

100 correct=((price% MOD 100)/100)\*rate%

110 vat%=vat%+correct

120 correct=FNdecimal\_part(correct)

130 ENDPROC

140

150 DEF FNdecimal\_part(n)

160 IF n<1 THEN =n

170 REPEAT

180 n=n-1

190 UNTIL n<1

200 =n

In Listing 5.2 the procedure PROCvat illustrates some of the important points to bear in mind with this type of calculation. Line 90 does the initial determination of the VAT using purely integer arithmetic. The brackets are vital to ensure that the DIV is done before the multiplication as a floating point number, with a possible error. The result of the calculation in line 90 is not quite what we want, however, since the integer division only returns the quotient and ignores any fractional part of the result. Line 100 calculates the correction necessary to allow for this. Here we have no choice but to resort to real numbers, since fractions of pence have to be considered, but there is no danger of a significant error since the value of the correction must fall in the range 0 to just below 15 and even at the top of this range 9 significant figures is more than sufficient. Four significant figures for a value of 10 would still be reliable is more than sufficient. Four significant figures for a value of 10 would still be reliable to the nearest on hundredth of a penny and we are actually doing much better than this.

Line 110 of Listing 5.2 adds the correction, but since vat% is an integer variable the effect is to add only the integer part of the variable 'correct'. This gives the VAT to the nearest penny, with the rounding being downwards, and line 120 determines the fractional part of 'correct' which can either be used to round the value of vat% or retained for future examination. In the former care we would add a line such as:

125 IF correct>=0.5 THEN vat%=vat%+1

The rounding technique is completely under our control, whatever we choose to do.

The routine in Listing 5.2 requires a little modification if we have to calculate VAT, or any tax or discount, at a rate which has a fractional percentage. If the rate of VAT was 12.5% we would do the calculation by multiplying by (125 DIV 1000) rather than 12.5/100, which requires real arithmetic.

When and where we do any rounding of values can be vitally important if we are doing something like compiling an itemised bill, which includes the individual VAT for each item and a grand total at the end. Suppose, for example, that there were two items costing 11 pence. The VAT on each would be 1.65 pence and so, if we round at this stage, each will be charged at 13 pence, giving a grand total of 26p. If, however, we total up the cost, to give 22p and then calculate the VAT we get 3.3p, leading to a grand total of 25p. Here the error is quite small, but on a much longer list of items it could rapidly become enormous. Obviously there will be an element of "swings and roundabouts" with some rounding errors going one way and some the other, but I do not think you will have a very receptive audience if you try this argument on your customers! The important thing is to know what protocol you are to use, or decide on one for yourself, to deal with these rounding problems. The scientist who is pleased at having measured the distance of Mars to within 1% will be far less delighted by a similar error in the bill for servicing his car!

The problem is not unique to VAT; any financial calculations involving division require the same sort of consideration. The authorities who deal with taxation, such as the Inland Revenue, are well aware of this and will normally be able to supply a leaflet to give guidance as to how, and at what point, you should do the rounding. It is not generally satisfactory to take something like a taxation rate and just plug it into the program without considering the rounding problems. Taxation tables, for example, are so constructed as to ensure that, taking the whole year together, the correct amount of tax is paid. They have their own way of spreading out rounding errors, different rules apply to computerised systems and you will need to find out what they are.

The whole question of the security of information held on computers is a burning issue at the moment. The micro is less vulnerable than the mainframe which is often accessible from the telephone lines and usually holds its long term data store permanently online in a hard disc system. At least on the micro you can take out the tape or disc and lock it away. It is still desirable, however, that sensitive data should only be accessible to those authorised to use it and the password is a convenient means of attempting to ensure this.

Listing 5.3 shows a typical routine for password protection of, for example, a database. Here the password is just defined in the program, at line 20, but this would not be a very clever way of implementing the system, in practice, you would normally be retrieving the password from the protected storage medium such as the tape or disc. The function at line 130 actually takes in the password. I have allowed only capital letters and limited the length to 20 characters. There is no particular reason for the former restriction, except that non-alphanumerics are harder to remember and the restriction to a minimum of 20 characters is largely for screen tidiness. Most people will have a job to remember as many as 20 characters and if they write the password down most of its value is lost.

The password does not, for obvious reasons, appear on the screen, but it is helpful to let the user know how many characters have been typed and so line 160 prints a star for each character entered. Notice that at lines 50 and 90 I have made sure that the password is erased from the memory as soon as its job is done, just in case someone manages to get into the program.

Line 80, which responds to invalid passwords, also counts the number of unsuccessful entries and the program ends, at line 50, after three tries. This copes with attempts to guess the password but allows scope for genuine users to make mistakes. There will be nothing to stop the potential hacker going round for another try unless you take more drastic action than that in line 50, which simply ends the program. You could, for example, delete the data file from the disc, by making line 50 execute \*DELETE DATA but this is dangerous unless you have a reliable regime for keeping back-up copies of your discs! If you want to adopt this method you will also need to ensure that the program will not run at all if the disc is write protected by starting it with a sequence such as:

1 ON ERROR IF ERR=&C9 THEN 5

2 \*SAVE TEST 100+10

3 PRINT"REMOVE WRITE PROTECT TAG AND TRY AGAIN":END

5 ON ERROR OFF

10 Main Program starts

Error &C9 is Disc Write Only will be invoked at line 2 if the tag is present, line 2 does not actually do anything useful, its job is merely to test for the write protect tag by inducing the error.

#### Listing 5.3

10 MODE 6

20 correct$="JESSICA"

30 tries%=0

40 REPEAT CLS

50 IF tries%=3 correct$="":CLS:PRINT"NAUGHTY, NAUGHTY !":END

60 pass$=FNpassword(0,10,"Password ?")

70 valid%=FNvalidpass(correct$,pass$)

80 IF NOT valid% PRINT'"INVALID PASSWORD-PRESS SPACE TO RE-TR

Y":tries%=tries%+1:REPEAT UNTIL GET=32

90 UNTIL valid%:pass$="":correct$=""

100 CLS:PRINT"CORRECT PASS"

110 END

120

130 DEF FNpassword(x%,y%,prompt$)

140 LOCAL in$,a$

150 REPEAT

160 PRINT TAB(x%,y%)SPC(40)TAB(x%,y%);prompt$;" ";STRING$(LEN(i

n$),"\*");

170 REPEAT \*FX15,1

180 a$=GET$

190 UNTIL INSTR(CHR$(13)+CHR$(127)+"ABCDEFGHIJKLMNOPQRSTUVWXYZ"

,a$)

200 IF a$<>CHR$(13) AND a$<>CHR$(127) AND LEN(in$)<20 in$=in$+a

$

210 IF a$=CHR$(127) in$=LEFT$(in$,LEN(in$)-1)

220 UNTIL a$=CHR$(13)

230 =in$

240

250 DEF FNvalidpass(correct$,offered$)

260 IF offered$=correct$ THEN =TRUE ELSE =FALSE

#### Listing 5.4

10 MODE 6

20 REPEAT CLS

30 pass1$=FNpassword(0,10,"Enter your chosen password")

40 pass2$=FNpassword(0,10,"Enter your password again")

50 valid%=FNvalidpass(pass1$,pass2$)

60 IF NOT valid% PRINT'CHR$(7)"BOTH VERSIONS SHOULD MATCH. PRE

SS THE SPACE BAR TO TRY AGAIN":REPEAT UNTIL GET=32

70 UNTIL valid%

80 CLS:PRINTTAB(0,0)pass1$

90 END

100

Listing 5.4 shows an example of a routine for setting up a password, it requires the function and procedure in listing 5.3 to be entered as well. I have used the normal method of having the user type the chosen password twice and only accepting it if both versions are the same, since it is easy to make a mistake when you are not getting a screen echo of what you type. If you want to let the user change the password make sure that you check that he knows the existing one before allowing a change!

If you use password protection, one potential source of weakness is that the password has to be stored, along with the data, on tape or disc. Since it is possible to read the contents of the medium directly without having access to the program which was designed to retrieve the data it is possible for someone to find the password and even to extract the actual data, although the latter might not necessarily make much sense without a knowledge of what it refers to. It is possible to encrypt the data on the disc or tape, however, and so make it more difficult to do this kind of thing. The data is decoded when retrieved by the legitimate program, of course. It would clearly not be very sensible for me to go into methods of this kind in too much detail here, but I offer Listing 5.5 as a simple example of the sort of thing possible.

#### Listing 5.5

10 MODE 6

20 REPEAT CLS

30 INPUTLINE"Enter a string ";a$

40 coded$=FNencrypt(a$)

50 PRINT"You entered ";a$

60 PRINT"The encrypted version is ";FNencrypt(a$)

70 PRINT"The decoded version is ";FNencrypt(coded$)

80 REPEAT UNTIL GET=32

90 UNTIL FALSE

100

110 DEF FNencrypt(a$)

120 LOCAL char$,coded$,char1$:char1$=LEFT$(a$,1)

130 FOR i%=1 TO LEN(a$)

140 char$=MID$(a$,i%,1)

150 IF i%<>1 char$=CHR$(ASC(char$) EOR ASC(char1$))

160 coded$=coded$+char$

170 NEXT

180 =coded$

This uses the property of the EOR logical operation that when it is applied twice with the same operand to the same value, the second operation exactly reverses the effect of the first. The FOR...NEXT loop, at line 130 works its way through a password and EORs the ASCII code of each letter with that of the first character. The encrypted version would then be stored, only the first letter appearing in clear.

Decoding is carried out by a simple repeat of the coding operation. There is no reason, of course, why the EOR should be performed with the first character of the keyword, any value would do, but it is essential to know what it is or we cannot break the code ourselves. If you try Listing 5.5, by the way, do save it before you run it since some of the encrypted characters may be control codes which will have some interesting effects when sent to the VDU drivers at line 60. You may find that you are changing mode with VDU 22, for example, and this may crash your machine.

No password protection or data encryption technique will be of much use if someone can take a look at the actual program and so see exactly how it works, so it is essential that you arrange to make the program itself inaccessible. I shall be dealing with some techniques for achieving this in Chapter 8.

### Dealing With Dates

Whatever your business or financial application you are almost certainly going to want to handle dates at some point in the program. Simply recording dates is easy enough but you will sometimes need to do calculations on dates, perhaps to determine the number of days between two dates or to decide what the date will be so many dates hence. We must also consider the problem of sorting information into chronological order. This type of manipulation of dates is often far from being as simple as it might first appear. Because we can express a date in numerical form it is easy to suppose that it can be dealt with in the same way as any number, but its appearance is deceptive because it does not conform to the normal rules of numbers, largely because of the varying lengths of months. Add the complications of leap years and you have quite a bit to think about. Let us begin, however, at the beginning, with the entry and validation of dates at the keyboard.

#### Listing 5.6

10 MODE 6

20 REPEAT

30 REPEAT

40 INPUTLINE"Please enter a date "date$

50 flag%=FNdate\_OK(FNstrip(date$))

60 IF NOT flag% VDU7:PRINT"\*NOT VALID\*"

70 UNTIL flag%

80 PRINT;day%;"/";month%;"/";year%;" is a valid date"

90 UNTIL FALSE

100

110 DEF FNdate\_OK(date$)

120 REM\*CHECK LENGTH RANGE\*

130 IF LEN(date$)>8 OR LEN(date$)<5 THEN =FALSE

140 LOCAL i%,sep%,count%,mon%,maxdays%

150 REM\*COUNT SEPARATORS AND CHANGE TO /\*

160 FOR i%=1 TO LEN(date$)

170 IF INSTR("1234567890",MID$(date$,i%,1))=0 sep%=sep%+1:date$

=LEFT$(date$,i%-1)+"/"+MID$(date$,i%+1)

180 NEXT

190 REM\*MUST BE TWO SEPARATORS, NOT TOGETHER AND NOT FIRST OR L

AST CHARACTERS\*

200 IF sep%<>2 OR INSTR(date$,"//")<>0 OR LEFT$(date$,1)="/" OR RIGHT$(date$,1)="/" THEN =FALSE

210 REM\*EXTRACT DATE\*

220 day%=VAL(date$)

230 REM\*DATE MUST BE 1 TO 31 INC\*

240 IF day%<1 OR day%>31 THEN =FALSE

250 FOR i%=1 TO LEN(date$)

260 REM\*EXTRACT MONTH\*

270 IF MID$(date$,i%,1)="/" AND count%=0:count%=1:month%=VAL(MI

D$(date$,i%+1))

280 REM\*EXTRACT YEAR\*

290 IF MID$(date$,i%,1)="/" AND count%=1:year%=VAL(MID$(data$,i

%+1))

300 NEXT

310 REM\*VALIDATE YEAR\*

320 IF year%>99 THEN =FALSE

330 RESTORE

340 REM\*READ MAXIMUM NUMBER OF DAYS ALLOWED AND VALIDATE MONTH\*

350 REPEAT

360 READ mon%,maxdays%

370 DATA 1,31,2,28,3,31,4,30,5,31,6,30,7,31,8,31,9,30,10,31,11,

30,12,31,100,100

390 REM\*MONTH MUST BE IN DATA LIST\*

400 IF mon%=100 THEN =FALSE

410 REM\*LEAP YEAR\*

420 IF month%=2 AND (year% MOD 4)=0 maxdays%=29

430 REM\*CHECK DATA AGAINST ALLOWED MAXIMUM\*

440 IF day%>maxdays% THEN =FALSE

450 =TRUE

460

470 REM\*STRIP LEADING AND TRAILING SPACES\*

480 DEF FNstrip(a$)

490 IF LEFT$(a$,1)<>" " THEN 530

500 REPEAT

510 a$=RIGHT$(a$,LEN(a$)-1)

520 UNTIL LEFT$(a$,1)<>" "

530 IF RIGHT$(a$,1)<>" " THEN =a$

540 REPEAT

550 a$=LEFT$(a$,LEN(a$)-1)

560 UNTIL RIGHT$(a$,1)<>" "

570 =a$

The most convenient form of presenting dates to a computer is the familiar '7/4/48' layout and Listing 5.6 accepts and validates dates in this format. It does not, actually, insist on the slash as a separator; it merely requires that there are two non-numeric characters as separators, they can even be spaces.

The function FNdate\_OK, at line 110, returns TRUE if the data is valid but otherwise it is FALSE. Line 130 uses the fact that any date in this form must have at least 5 characters and can have no more than 8, further checks would be a waste of time if the length were outside this range. The FOR...NEXT loop starting at line 160 goes through the date and counts the separators, at the same time it replaces them all with the slash. Having a standard type of separator makes the subsequent coding easier.

Line 200 looks at the number and position of the separators, rejecting the date if there are other than two separators and if they are clearly wrongly placed, by being together or at either end of the input. If all the tests applied so far are passed we have the correct format for the data and so line 220 extracts the day number, which must be between 1 and 31; this is checked at line 240.

The FOR...NEXT loop at line 250 now looks through the string and finds the separators so that it can extract the month and year numbers. The latter must be less than 100, hence line 320. The next piece of code reads along the DATA statement, at line 380, and tries to find the month number. The latter must be less than 100, hence line 320.

The next piece of code reads along the DATA statement, at line 380, and tries to find the month number, which is accompanied by the number of days permitted. If the month number is not found then the data read runs off the end of line 380 and returns the nonsensical month number of 100.

Line 420 deals with the leap year. The rule for a leap year is that it is a year divisible by four but with the proviso that century leap years are only leap years if divisible by 400 as well. Line 420 deals with the practical situation for dates at present and up to the year 2100; 2000 being a leap year. If you are wanting to work with dates in the past, or you intend your software to be used in the 22nd century, then you will need to extend the routine to accept the century as well and to add the check:

425 IF maxdays%=29 AND (year% MOD 100)=0 THEN IF (year% MOD 400

)<>0 maxdays%=28

For most business purposes, however, the year number in the century is all that is required.

The final check at line 440 ensures that the day number is in range for the month. After the function has been executed we have a valid day, month, year combination in the appropriate integer variables which are printed out at line 80. The function at line 480 merely strips any redundant spaces from the back and front of the input since these would invalidate an otherwise perfectly satisfactory date and confuse the user.

Once we have a valid date entered and stored in the three variables day%, month% and year% we may want to do various calculations based upon it. We might, for example, want to find out what the date is 30 days hence so that we know when to send out a reminder for a bill or perhaps we want to calculate the time lapse between two dates. This is rather complex to do using the conventional method of dating since the relationship between the numbers of days and the time between dates is complicated by the differing lengths of months and the possibility of changing the year. The best way to tackle the problem is to set up a simpler and more logical dating system for use by the computer internally, and to have subroutines for translating between conventional dates and the internal system at input and output.

#### Listing 5.7

20 MODE 6

30 REPEAT

40 PRINT"Enter the date, use commas as separators";:INPUT'day%

,month%,year%

50 PRINT"This is day ";FNdayno(da%,month%,year%)

60 UNTIL FALSE

70

80 DEF FNdayno(day%,month%,year%)

90 REM\*MONTH ONE\*

100 IF month%=1 THEN =day%

110 LOCAL dayno%,i%,x%

120 RESTORE 250

130 REM\*TOTAL UP COMPLETE MONTHS\*

140 FOR i%=1 TO month%-1

150 READ x%

160 dayno%=dayno%+x%

170 NEXT

180 REM\*LEAP YEAR CORRECTION\*

190 IF month%>2 AND (year% MOD 4)=0 THEN dayno%=dayno%+1

200 REM\*ADD IN DAYS FROM CURRENT MONTH\*

210 dayno%=dayno%+day%

220 =dayno%

230

240 REM\*DAYS IN MONTHS\*

250 DATA 31,28,31,30,31,30,31,31,30,31,30,31

Listing 5.7 shows how such an internal dating system might be set up, the method used being to number the days consecutively throughout the year from 1 to 365, or 366 in leap years. Each internal date then consists of this day number and the year, the months are ignored.

#### Listing 5.8

5 MODE 6

10 REPEAT

20 INPUTLINE"Enter the day number "dayno%

30 INPUTLINE"Year ? "year%

40 PROCdate(dayno%,year%)

50 PRINT"DATE ";day%;"/";month%;"/";year%

60 UNTIL FALSE

70

80 DEFPROCdate(dayno%,year%)

90 REM\*WITHIN JANUARY\*

100 IF dayno%<32 month%=1:day%=dayno%:ENDPROC

110 LOCAL days\_this\_month%,days\_next\_month%

120 REM\*START AT JANUARY\*

130 month%=1

140 days\_next\_month%=31

150 RESTORE 310

160 REPEAT

170 REM\*MOVE TO NEXT MONTH\*

180 days\_this\_month%=days\_this\_month%

190 READ days\_next\_month%

200 month%=month%+1

210 REM\*LEAP YEAR CORRECTION\*

220 days\_this\_month%=FNleap(days\_this\_month%,year%)

230 days\_next\_month%=FNleap(days\_next\_month%,year%)

240 REM\*SUBTRACT WHOLE MONTHS\*

250 REM\*UNTIL LESS THAN A MONTH LEFT\*

260 dayno%=dayno%-days\_this\_month%

270 UNTIL dayno%<=days\_next\_month%

280 day%=dayno%

290 ENDPROC

300

310 DATA 28,31,30,31,30,31,31,30,31,30,31

320

330 DEF FNleap(d%,year%)

340 IF d%=28 AND (year% MOD 4)=0 THEN =29 ELSE =d%

Listing 5.8 reverses the process, taking the year and day number and converting it back to date, month, year format.

Let us first look at Listing 5.7, which converts into the internal system. Line 100 deals with the simple case of a day in January, the date and day number are the same in this case. We then enter the FOR...NEXT loop at line 140 which reads through the DATA in line 250 until it gets to the current month and totals up the days in the months up to the current one, which is incomplete, of course, and then line 210 adds in the days for the current month. Any correction required for leap years is done at line 190.

Reversing the process, that is, taking the day number and extracting a conventional date from it, is a little more complex as you can see by the length of Listing 5.8. The principle of the method is to take the day number and to subtract, successively, the lengths of months until we reach a stage where the number of days left is less than the length of the next month. If we keep track of how many months we have subtracted we then have the month number and the date is the remainder. Line 100 again disposes of the relatively trivial case of a day number in January before we enter the main REPEAT...UNTIL loop between lines 160 and 270 which does the subtractions, taking the month lengths from the DATA at line 310. Lines 220 and 230 adjust the length of February for leap years. At the end of the procedure we are left with the conventional date, in the variables day%, month% and year%, ready for output. The routines in Listings 5.7 and 5.8 are both intended to deal only with valid dates and so any input to them should normally have been vetted by a routine such as that in Listing 5.6. For the sake of brevity I have not carried out input-checking in the examples so when you are trying them out, remember to enter sensible numbers.

Listing 5.9 is an example of the use of the internal format to calculate the date which will be reached so many days after the input date. It is not self-contained, of course and will require the appropriate procedures and functions from listings 5.7 and 5.8. We simply convert the input into a day number and a year and then add the number of days to elapse, all at line 70, so getting the new day number. A check has to be made that we have not moved ahead into another year, which requires 365 to be subtracted, or 366 if the current year is a leap year, and then we must allow for the possibility that a new century has begun, returning the year count to zero. We can now go on, if we wish, to extract a conventional date from the day number and the year.

The internal date format that I have described here also lends itself very well to tackling the problem of sorting information into chronological order, which process requires a determination of the relative order of pairs of dates. Two dates can easily be compared, first on the basis of the year, and if the years are the same, a second comparison can be made; this time on the basis of the day numbers. If you try to do a similar comparison with the conventional date *three* checks are needed on the year, month and date.

The routines that I have described in this section should give you a basis for constructing any type of manipulation on date and time lapses. I have, throughout, simplified the leap year problem, as already explained, but this should create no difficulties in software for use with current dates. I have as much faith in the BBC as anyone, but I think that it will have changed quite substantially by the year 2100!

#### Listing 5.9

10 MODE 6

20 REPEAT

30 INPUT"Enter date "day%,month%,year%

40 REPEAT

50 INPUTLINE"How many days, Max 365 "elapse%

60 UNTIL elapse%>0 AND elapse%<366

70 newdayno%=FNdayno(day%,month%,year%)+elapse%

80 REM\*CORRECT FOR YEAR CHANGE\*

90 IF (year% MOD 4)=0 THEN IF newdayno%>366 newdayno%=newdayno

%-366:year%=year%+1

100 IF (year% MOD 4)<>0 THEN IF newdayno%>365 newdayno%=newdayn

o%-365:year%=year%+1

110 REM\*CORRECT FOR NEW CENTURY\*

120 IF year%=100 year%=0

130 REM\*EXTRACT DATE\*

140 PROCdate(newdayno%,year%)

150 PRINT"NEW DATE "day%;"/";month%;"/";year%

160 UNTIL FALSE

### Checking the Validity of Data Entry

The data used by any piece of software usually originates from a human operator at the keyboard and any error at this point will make a nonsense of the output of the program. In this chapter and in Chapter 2 I have discussed various techniques for minimizing the possibility of erroneous entries, but only in cases where the data being keyed has some relevance in the outside world and hence covers a range of sensible values or characters. Not all data entered falls into this category, however. Sometimes a string of apparently arbitrary numbers or letters has to be entered, for example account or catalogue numbers. Here the chances of an operator making a mistake are quite high since what is being entered has no logical pattern and characters are easily mistyped or transposed. The consequences of a mistake could be horrific, however. You might deliver an elephant instead of a teaspoon or send some pensioner the electricity bill for the British Broadcasting Corporation!

There is no absolutely foolproof solution to this problem but it is possible to reduce the probability of error to a very small level by the use of "check digits". A check digit is an extra digit added to the end of a number, which is actually not part of the number, but whose value is determined by carrying out a calculation with the remaining digits. The calculation chosen normally determines a check digit which, when combined with the actual number, gives a value that always conforms to some particular, predetermined, arithmetical rule.

All our account numbers, for example, would have the appropriate check digit and all would, therefore, conform to the rule so any number entered that did not do so must have been mistyped. We might, to take a simple example, have a set of accounts numbered from 1 to 9 but make the rule that all the digits in the account numbers should add up to 10. Account 1 would thus be represented as 19, 2 as 28 and so on, the 9 and 8 being check digits. An entry of 18, for example, must of necessity be a mistake as the digits only add up to 9. This particular system would not be a very good one, in practice, since it is not very robust. Errors which cancel out could easily be made, but it would still cut down the possibility of an error compared to a completely unchecked system.

No check digit system can be absolutely perfect; there is always the possibility that a series of errors will be self-cancelling and so a random selection of errors can never be completely trapped, but this is rarely the problem in practice. On the other hand, a single digit mistake or the transposition of two digits is very common and it is possible to eliminate this kind of error almost completely. Two types of check digit system are in common user, the '2-1-2 modulus 10' and the 'geometrical' methods. The latter is the more robust and completely eliminates single digit and transposition errors whilst reducing the possibility of a random selection of errors being overlooked to around three chances in a 100.

The geometrical method computes the check digit by starting at the least significant, right hand, end of a number and multiplying each digit by two to the four, the hundreds by 8 and so on. The individual results are then added together to produce a total. The check digit is then taken as the number which must be added to this total to give a number exactly divisible by eleven. An example will perhaps clarify this. Let us take the number 1234. The total mentioned above will be:

(4\*2)+(3\*4)+(2\*8)+(1\*16)=52

which leaves remainder 8 when divided by eleven, and so the check digit will be 11-8 = 3, making the full number 12343. To check the number at input we simply repeat the calculation above and add in the check digit to get 52+3=55, which is exactly divisible by eleven and hence the number 12343 can be accepted as valid input. The value of eleven used as divisor has been shown to be that which gives the most reliable error checking. One problem with using eleven is that it can give 10 as a check digit, which is rather untidy, and it is usual to replace it with a letter of the alphabet, I have used A in my examples.

#### Listing 5.10

10 MODE 6

20 REPEAT

30 REPEAT

40 INPUTLINE"Enter a Number "n$

50 UNTIL VALn$>0

60 PRINT"Number+Check Digit "n$+FNcalculate\_cd(n$)

70 UNTIL FALSE

80

90 DEF FNcalculate\_cd(n$)

100 LOCAL i%,cd%,total%

110 FOR i%=LENn$ TO 1 STEP-1

120 total%=total%+VAL(MID$(n$,i%,1))\*2^(LENn$+1-i%)

130 NEXT

140 cd%=11-(total% MOD 11):IF cd%=11 cd%=0

150 IF cd%=10 THEN ="A" ELSE =STR$(cd%)

Listing 5.10 shows a routine for calculating the geometrical, base 11, check digit required by any number on its first entry into the computer program. This first entry, before the check digit is added, must be done with great care so that an error is not made at this point, so negating any subsequent checking. The last part of line 140 is worth nothing, this deals with the case where no check digit is actually necessary to meet the arithmetical conditions of the check. Under such circumstances the logical check digit is zero, rather than eleven, as would be returned without this statement.

#### Listing 5.11

10 MODE 6

20 REPEAT

30 INPUTLINE"Enter a number "n$

40 IF FNvalidate(n$) PRINT"\*VALID INPUT\*" ELSE PRINT"\*ERROR";C

HR$(7)

50 UNTIL FALSE

60

70 DEF FNvalidate(n$)

80 IF VAL(n$)=0 THEN =FALSE

90 LOCAL total%,i%,digit%,digit$

100 FOR i%=LENn$ TO 1 STEP-1

110 digit$=MID$(n$,i%,1)

120 IF digit$="A" THEN digit%=10 ELSE digit%=VAL(digit$)

130 total%=total%+digit%\*2^(LENn$-i%)

140 NEXT

150 IF total% MOD 11=0 THEN =TRUE ELSE =FALSE

Listing 5.11 illustrates input checking using the check digit generated in Listing 5.10. I have deliberately excluded zero as valid input, with line 80, since no distinction can be drawn between one zero and a whole sequence of zeros. In this sense, zero is a special case and is best completely excluded as valid input.

There is no doubt that the geometrical method of check digit generation is the most robust as regards error trapping but it does have the drawback that it involves quite a long calculation and can slow up the input. This may not be noticeable by someone typing with two fingers, as I am doing now, but can be irritating to a touch typist. The situation will become worse as the numbers being entered become longer. For this reason, the '2-1'2 modulus 10' system is worth considering since it gives a rather faster input. The technique is similar to the geometrical method in that it involves position weighted multiplication of the digits in a number and then the extraction of the remainder, modulus, on division by ten. In this case, however, the multiplication of the digits is done, alternately, by 2, 1, 2 and so on, again starting at the least significant end of the number. Taking the same example as before, the number 1234:

(4\*2)+(3\*1)+(2\*2)+(1\*1)=16

Division by ten gives a remainder of 6 and so the check digit is 10-6=4. The complete number will then be 12344. Listing 5.12 demonstrates the calculation. Line 150 again ensures that zero rather than 10 is returned for the case where no check digit is needed by the arithmetic. Input checking is illustrated in listing 5.12 and again I have excluded the special case of zero. The 2-1-2 method is not quite as reliable as the geometrical technique but will detect transpositions and single digit transcription errors almost as surely, and it is, as I have already noted, these two types of error which are the most common.

#### Listing 5.12

10 MODE 6

20 REPEAT

30 INPUTLINE"Enter a Number "n$

40 PRINT"Number+Check Digit="n$+FNget\_cd(n$)

50 UNTIL FALSE

60

70 DEF FNget\_cd(n$)

80 LOCAL n%,i%,total%,cd%

90 n%=1

100 FOR i%=LENn$ TO 1 STEP-1

110 IF n%=1 n%=2 ELSE n%=1

120 total%=total%+VAL(MID$(n$,i%,1))\*n%

130 NEXT

140 cd%=10-(total% MOD 10)

150 IF cd%=10 cd%=0

160 =STR$(cd%)

The use of check digits is essential if any kind of security is to be associated with input of arbitrary strings of characters. I have illustrated the technique for numbers but it can equally be applied to alphabetic characters or indeed to a mixture of alphanumerics, by deriving the check digit from the ASCII codes for the characters rather than from their actual values. Try out the two input routines, in listings 5.11 and 5.12 on your bank account and credit card numbers, just out of interest.

#### Listing 5.13

10 MODE 6

20 REPEAT

30 INPUTLINE"Enter a Number "n$

40 IF FNvalid(n$) THEN PRINT"\*VALID INPUT\*" ELSE PRINT "\*ERRO

R\*";CHR$(7)

50 UNTIL FALSE

60

70 DEF FNvalid(n$)

80 IF VALn$=0 THEN=FALSE

90 LOCAL n%,i%,total%

100 n%=2

110 FOR i%=LENn$ TO 1 STEP-1

120 IF n%=1 n%=2 ELSE n%=1

130 total%=total%+VAL(MID$(n$,i%,1))\*n%

140 NEXT

150 IF total% MOD 10=0 THEN =TRUE ELSE =FALSE

### Conclusion

The techniques covered in this chapter are not unique to business applications but are most commonly found in this type of software. If you are going to write business software, either for yourself or other people, it must be good. Remember my comment in the introduction that computers never make mistakes, your livelihood may well depend on your software behaving likewise.

# Chapter 6.

# Filing Systems

There are three main types of filing system commonly used on the Acorn computers which run BBC BASIC, ROM, tape and disc. There are, in addition, Econet, which is essentially an extension of the disc system, and the Teletext filing system both of which are outside the scope of this book.

The BBC with the Z80 second processor fitted does not use the Acorn DFS directly but the whole machine runs under CP/M, which provides its own standardized filing system. I shall not be dealing with the full range of facilities available under CP/M in any great detail in this book, but I will be commenting on software compatibility between the DFS and CP/M and discussing the CP/M filing system as used from BBC BASIC.

The ROM system is rather different from the other two in that it is 'read only', you cannot write to ROMs with the computer unless you add an extra piece of hardware, an EPROM programmer. Files which are to be resident in ROM must be in a special format, known as a "ROM image", which is similar to that used by the tape filing system but not identical. A listing of a program for converting a disc file into this format is contained in an application note on the ROM filing system of the BBC micro available from Vector Marketing Ltd, whose address appears in Appendix 2.

The tape filing system is standard throughout the machines but the disc filing system comes in two variants, the Acorn DFS, which is used on the BBC and the Advanced Disc Filing System, ADFS, which is used by the Electron and the ABC series of computers.

The Acorn DFS has been issued in two main versions, 0.90 and 1.20, the latter being required by machines with a second processor. The 1.20 appears to have become the standard version fitted to BBCs from around the end of 1984 but there is no difference between the two issues from the programmer's point of view, provided standard, documented commands are adhered to. The 1.20 version is markedly faster, however, in a number of operations. The model B Plus, which appeared in May 1985, has a modified version of the Acorn 1.20 DFS designed to accommodate the new disc controller chip and referred to as version 2.10, but the programmer will find no incompatibility provided that official routines are adhered to and no attempt is made to program the disc controller directly.

The ADFS uses completely different hardware from the DFS and is much more versatile and sophisticated than its forerunner. The file structure is of a 'tree' type, permitting nesting of directories within directories in a hierarchical system and breaking completely away from the DFS's limitation of 31 files per disc. The DFS and ADFS are completely incompatible in the sense that they cannot read ach other's discs but there is no difference in the software required to handle data files from BASIC programs so software translates easily between the two. If filing system commands, that is 'star' commands, are used, however, some modifications may well be needed since the ADFS has some commands with no equivalents in the DFS and such operations as changing drive are not accomplished in the same way in both systems. Equivalent error messages generally have the same error numbers so error traps will often be transferrable but there are, inevitably, some errors that have relevance in only one of the systems. In this chapter I will comment only on modifications to the programs that are required when changing between DFS and ADFS; in the absence of such comment the routines work on either.

### The User and the Filing System

Probably the most daunting aspect of a microcomputer for the novice user is the filing system and it is particularly important that software is explicit and clear as to what manipulations of tapes or discs are required. This is especially vital if the user has to create and use data files, since a mistake can result in the loss of a great deal of work and possibly valuable information.

In the best software, the user should be largely unaware of the filing system; once he has loaded the tape or disc it should get on with its job as unobtrusively as possible. This is a situation which is much easier to achieve with discs that with tape, since the disc drive itself is controlled by the computer whereas the user has to press buttons on the tape recorder. Both discs and tapes have their own hazards for the user and it should be in the nature of good software that he is protected as fully as possible from losing his data.

### Tape Disc Compatibility

BBC BASIC and the Acorn disc and tape filing systems have been very carefully constructed to allow upward compatibility, in the sense that software written for tape will run on the disc system, equivalent commands being used. This does not mean, however, that software will transfer in a friendly manner, at the very least the user prompts required for tape will not be the same as with discs and it will normally be necessary to produce separate versions of your software for the two filing systems. The differences will only need to be cosmetic in most instances and the actual nuts and bolts can be left the same but often the disc user will need some facilities that the tape user can manage without. He may need to see the disc catalogue, for example, especially if he has to enter the name of a file to be loaded; a null filename is not acceptable to DFS, and you might want to offer the possibility of locking and unlocking files on the disc to give some data protection.

It is sometimes useful to be able to find out, from within a program, which filing system is in use. An operating system routine, OSARGS, is provided to allow this to be done. OSARGS is called at location &FFDA and if it is entered with zero in the accumulator and Y registers it returns with a number in the accumulator which depends on the filing system selected. The code required to abstract the information is:

10 Y%=0:A%=0:!&70=USR(&FFDA)

20 file\_system\_type%=?&70

The significance of the values returned is:

1 – 1200 baud cassette

2 – 300 baud cassette

3 – ROM filing system

4 – DFS

5 – Econet

6 – Teletext

7 – IEEE interface

8 – ADFS

### Tape Files

Most of the prompts required by the tape user which refer to the tape recorder operation are issued automatically by the operating system and the main job of the software writer is to prompt for tapes, appropriately blank or containing data, to be loaded. Prompts such as "Please Load a Tape with the Data File – Press the Space Bar When Ready" are particularly helpful since the user is made to feel in control; the infernal machine is awaiting his command!

If you are writing software which uses data files you may want the user to be able to enter filenames both for saves and loads. The filename is actually optional on a load, a null being accepted as a command to load the first file found on the tape, but a save requires an explicit name. User-entered filenames are only really necessary if you want there to be more than one file on each tape and with the ready availability of cheap computer cassettes it is certainly worth considering asking your users to use a separate tape for each file. There will then be no need for a distinction to be made by the program between files generated at different times by the software and so you can supply some standard filename. If you do let your user enter a filename then the input must be trapped carefully before it is sent to the tape filing system; remember that the maximum length of the name should be nine characters and I would suggest that you restrict its contents to capital letters only. Punctuation and numbers appearing in the wrong place can cause problems.

One of the biggest hazards with tape filing is that the wrong tape may be loaded or the user may spell a filename wrongly. In either case the file required will never be found and the tape will run forever. Pressing the ESCAPE key will stop the tape sequence but will also break the program unless you have an effective ON ERROR trap which picks up the ESCAPE, such a trap is, therefore, absolutely essential. You should also be prepared for ESCAPE to be pressed *during* tape saves, since a user may suddenly realize that the wrong tape has been loaded, or that he pressed only PLAY and not RECORD. A safe exit is especially important here as otherwise the data may be hopelessly lost.

Errors resulting from faulty tapes or badly adjusted tape recorders are inevitable with tape filing. Write errors, which usually result from faulty tapes are not retrievable, and users should be encouraged to make more than one copy of a data tape. Indeed, for important applications you might well arrange to build this into your software. The tape file can be written twice to the same tape, or if the file is too long for this, a second tape can be prompted for. This will, of course, take time but the user will see its value the first time that he has a write error. I would go as far as to say that tape files are far too risky a medium for important data and any serious application should use discs for this reason alone.

Read errors from a perfectly written tape are usually a result of a wrongly set up tape recorder or dirty heads. They most commonly occur when you are trying to load a tape which was written on another machine or perhaps if the settings on your tape recorder have been accidentally altered. The action taken by the filing system when it detects a read error in a data file can be changed by the software using the \*OPT 2,n command. If n=0 then the computer ignores all errors and bashes on regardless, although the error messages will still appear. The default condition is n=1 which prompts the user to rewind the tape to a point before the error and replay it. The on-screen message is not very explicit here, and it is worth pointing out in your documentation that it is only necessary for the tape to be rewound a short distance; it need not be wound back to the start of the file.

If n is set to 3 then a read error has the same effect as any other error in the program and is passed, with a suitable error message and number, to the BASIC error handling routine for report or trapping by the ON ERROR statement with the tape loading operation being aborted. It is normally best to leave \*OPT 2,2 in charge. This is what is typically done and most tape users will rapidly learn the action necessary when such an error occurs. You do not normally want to ignore a tape error and there is really nothing that can usefully be done in an error trap, other than to ask the user to try again, which \*OPT 2,2 does anyway.

As well as \*OPT 2 there is also \*OPT 1,n available to the programmer. This is used to control the messages sent whilst tape files are being handled. The default for data files using INPUT# and PRINT# is that no messages are given at all, but it is more comforting for the user to have an indication that something is actually happening during a data save or load and so I would suggest that you invoke \*OPT 1,1 which will result in messages similar to those seen when programs are loading.

### Disc Filing

The addition of a disc drive to the BBC micro or the Electron is now a relatively inexpensive upgrade and the enhancement in the machine's capabilities and convenience of use is so dramatic that it is hard to believe that there will be very many tape-based machines around for much longer. At the simplest level discs provide very fast loading of programs and data files but if this is the only way that you are using your discs then you are not getting value for money. The DFS and ADFS support random access filing so it is possible to use a disc drive as an extension to the computer's memory and to access it in much the same way as you would the RAM. I shall be returning to the topic of random access filing in detail later.

When using discs it is absolutely essential that you have an ON ERROR trapping mechanism in operation which will report any problems to the user and then salvage the problem. A media fault may occur, the user may try to use an unformatted or full disc or simply try to update a locked file. Problems of this type should always be recoverable and cannot be dealt with by any type of specific protection within the software itself other than ON ERROR.

It is much easier to hide the filing system from a user on a disc machine than it is with tape since once a disc is loaded in the drive the software can take full control. You should try to achieve this as much as possible in your programs. The Acorn DFS can support up to two physical drives but each can be double-sided, with each side being treated as a separate drive. The system is said to have the capability of running two physical drives, but four virtual drives. The ADFS and the CP/M systems only support two physical drives, regarding both sides of a double-sided disc as being the same drive. Indeed CP/M requires two eighty track double-sided drives to function at all. The drives on the DFS are numbered 0 to 3. The ADFS regards the drives as 0 and 1, although they can alternatively be called with A and B, which is also the method used by CP/M.

### User Disc Drive Selection

Before I start on this section I should like to define some terminology which will make the discussion easier. I will use the expression 'a non-existent drive' to refer to a drive designation which, although supported by the filing system, is not actually present in the hardware, whereas the term 'invalid drive' will be used to indicate a drive number which is not supported by the system at all. An example of a non-existent drive would be drive 2 on a single drive system and an invalid drive would be drive 4 on any of the filing systems.

When writing software for the disc environment you will normally want it to be capable of running on a single drive machine, but it should also allow people with multiple drives to exploit the convenience of their investment. This will probably mean asking the user to load a disc and then to enter the drive number. On the DFS any number between 0 and 3 will have to be accepted, but this can cause problems if a number corresponding to a non-existent drive is entered and then an attempt is made to access that drive.

Exactly what happens depends on the set-up of individual drives. A single drive system will normally hang up if a drive other than zero is accessed. In a twin drive system the drives are numbered 0 and 1, and trying to use drive 2 or 3 may hang the machine. Some drives that I have encountered, however, are set up so as to ignore the disc side select signal – so setting the drive number to 2 actually selects drive 0 and 3 gives you drive 1. If you know that your software is only going to be used on a twin, single-sided, drive system at most, then you might usefully restrict the drive number input to 0 and 1. This will only cause irritation for the relatively small number of people who have a single, double-sided drive, where the two sides are normally regarded as drives 0 and 2. Users will need to be warned of the hazards of entering drive numbers that are not relevant for their hardware – but there is no way that you can protect them completely from the consequences on the 0.90 DFS. On the later, 1.20, version however, pressing the ESCAPE key will usually be recognized if the machine hands, so you should arrange your ON ERROR trap to deal safely with this. The 0.90 DFS will require a BREAK to retrieve the situation so data loss may well be an inevitable consequence. Your only totally safe recourse is to use only drive 0 and to prompt for appropriate disc changes, but this will deny the multiple drive owner the full use of his system.

The drive is set in the DFS by the command \*DRIVE n, where n is the drive number. Since this is a star command, if you want n to be derived from BASIC variable, as it must be if entered by a user, then the command will need to be passed directly to the command line interpreter. In BASIC 2, the OSCLI command can be used in the form OSCLI("DRIVE "+STR$(n)) but in BASIC 1 you will have to simulate OSCLI by a direct call to the command line interpreter as illustrated in listing 6.1.

#### Listing 6.1

10 MODE 6

20 REM\*SET UP BUFFER\*

30 DIM buffer%

40 REPEAT

50 REPEAT

60 INPUTLINE"Command ? "command$

70 PROCoscli(command$)

80 REM\*DO NOT ALLOW OVERFLOW OF BUFFER\*

90 UNTIL LENcommand$<32

100 UNTIL FALSE

110

120 DEFPROCoscli(command$)

130 REM\*INSERT COMMAND IN BUFFER\*

140 $buffer%=command$

150 REM\*POINT X AND Y REGISTERS AT BUFFER\*

160 X%=buffer% MOD 256:Y%=buffer% DIV 256

170 REM\*CALL COMMAND LINE INTERPRETER\*

180 CALL &FFF7

190 ENDPROC

User drive selection routines will need to be completely rewritten if you want to transfer your software to CP/M or the ADFS, although with the former the question of user drive selection does not normally arise. The ADFS does not use \*DRIVE at all, the drives are changed as part of the directory designation by the \*DIR command. I shall be returning to this topic later since the concept of the directory is rather different in the ADFS as compared to the DFS and needs some comment.

### User Entry of Full File Specifications

In all the filing systems it is possible to allow a user to enter the full file specification when selecting a file; this includes the drive, directory, and the filename. I would strongly recommend that you refrain from offering this facility except in applications software which is intended for use by people who have a very good understanding of the filing system. The ordinary user will probably find the full file specification rather awkward to handle; certainly at first, and may well have problems in interpreting the error messages that result from a wrong entry. Needless to say, if you allow entry of full file specifications you must have an effective ON ERROR trap to cope with the inevitable mistakes.

### User File Name and Directory Entry on the DFS

There are several hazards attendant on allowing the entry of filenames by the user. The problem of a totally invalid name, which is too long or contains illegal characters, is easily dealt with by normal input protection methods but it is also prudent to restrict the user in some way as to the specifications he can make in his filename. A technique that I would recommend is to insist that the filename contains only capital letters and, of course, that it does not exceed the maximum allowed length. This will permit adequate flexibility in naming of files without the hazard of numbers in the wrong place or invalid characters. If you want to allow the use of directories then in the DFS you will need some error trapping to ensure that the filename is not more than 9 characters long and that the first character is a letter and the second a full stop. In most instances there is no need for the user to be able to specify a directory in the DFS since the directory letter is only really a filename extension; it does not actually offer any extra facilities or file space. In the ADFS, however, the situation is rather different and I will come to the question of directories in the ADFS in the next section. Listing 6.2 is a general input protection routine that will ensure that the user can only enter a valid filename and directory on a machine fitted with the DFS.

#### Listing 6.2

10 MODE 6

20 REPEAT

30 INPUTLINE"Enter a File Name "f$

40 valid%=FNfilenameOK(f$)

50 IF NOT valid% THEN PRINT"\*INVALID FILE NAME\*";CHR$(7)

60 UNTIL FALSE

70

80 DEF FNfilenameOK(f$)

90 REM\*MUST BE AT LEAST 1 AND NO MORE THAN 9 CHARACTERS\*

100 IF LENf$>9 OR LENf$<1 THEN =FALSE

110 REM\*FULL STOP MUST BE SECOND CHARACTER IF USED\*

120 IF INSTR(f$,".")<>0 THEN IF INSTR(f$,".")<>2 THEN =FALSE

130 REM\*IF MORE THAN 7 CHARACTERS THERE MUST BE A FULL STOP FOR DIRECTORY\*

140 IF LENf$>7 AND INSTR(f$,".")=0 THEN =FALSE

150 LOCAL i%,char$,point%

160

170 FOR i%=1 TO LENf$

180 char$=MID$(f$,i%,1)

190 REM\*ONLY ONE FULL STOP\*

200 IF char$="." THEN point%=point%+1:IF point%>1 THEN i%=1000:GOTO 230

210 REM\*ONLY FULL STOP OR LETTERS\*

220 IF char$<>"." THEN IF char$<"A" OR char$>"Z" i%=1000

230 NEXT

240

250 IF i%>100 THEN =FALSE

260 =TRUE

### User Supplied Filenames and Directories with the ADFS

If you preclude the use of a full file specification as suggested above then there will be no difference in the problems encountered by the DFS and the ADFS as regards user-supplied filenames. The latter can support up to ten characters in a filename by the way, rather than the seven permitted by the DFS. Where you will have a potential problem with the ADFS is when you let a user select a directory.

In the DFS the directory is little more than an additional character added to the front of a group of files to allow certain operations on them to be done collectively. In the ADFS directories have a completely different significance. The term now applies to a file on the disc, which is not of the normal type. It does not hold data but information about a group of files. The directory file is really a subsidiary catalogue and before a file can be added to a directory the directory itself must be created and named with the \*CDIR command. Attempts to load files from a non-existent directory will, of course, result in a "Not Found" error, but so will attempts to write files to a directory that has not been created. For this reason allowing the user to specify the directory on the ADFS requires some careful coding with suitable error traps.

#### Listing 6.3

10 MODE 6

20 REPEAT

30 REPEAT

40 INPUTLINETAB(0)"Please enter a directory "dir$

50 UNTIL LENdir$>0 AND LENdir$<11

60 ON ERROR GOTO 110

70 OSCLI("DIR "+dir$)

80 UNTIL FALSE

90

100 REM\*ESCAPE TO EXIT\*

110 IF ERR=17 THEN CLS:END

120 REM\*ALL ERRORS EXCEPT 'NOT FOUND'\*

130 IF ERR<>&D6 CLS:REPORT:E$=INKEY$(1000):GOTO20

140 REM\*DIRECTORY NOT FOUND\*

150 PRINT"THIS DIRECTORY DOES NOT EXIST IS IT TO BE CREATED ?

";

160 REPEAT \*FX15,1

170 a$=GET$

180 UNTIL INSTR("YyNn",a$)

190 IF INSTR("Yy",a$) THEN OSCLI("CDIR "+dir$)

200 PRINT"DIRECTORY ";dir$;" CREATED"

210 GOTO 20

Listing 6.3 shows the sort of thing that will be required if a user is being allowed to enter a directory into which a file is to be saved. The "Not Found" error is number &D6 and I have trapped this specifically at line 130. The user has to be given the opportunity of either creating a new directory or escaping, since he may have actually been trying to add the file to a directory which already exists and have made a mistake in the entry. Attempts to load from a non-existent directory require no special error trapping, other than the possible amplification of the "Not Found" message. I would suggest that, if you want the user to be able to select a directory on the ADFS, you include this as a special routine with its own error trap, rather than allowing an extended file specification. This will allow the maximum amount of protection for the user.

The concept of changing drive on the ADFS is rather different from the DFS in that the drive specification has to be included as part of the directory, so \*DIR:1 will select drive one and the default directory. Selecting a non-existent drive on the ADFS on the Electron is catastrophic; the system hangs up completely and BREAK is required to restore control. User selection of the drive will normally be unnecessary, however, since the standard Plus 3 as supplied for the machine has only one drive. The second drive is an external add on unit which is not commonly in use.

### Friendliness in User File Name Entry

Assuming that we allow the user to enter a data filename and that any errors resulting from problems highlighted in the previous two sections are trapped then there are still other things to be anticipated. If no file of the same name as that entered already exists on the disc then there should be no difficulty unless the disc is full. Remember that on the DFS the disc will be regarded as full, even if there is physical space, when the catalogue contains 31 files. On the ADFS the only limitation is that a directory is restricted to a maximum of 47 files, the number of files on the disc is effectively unlimited except by the available space.

If the new filename does correspond to a file which already exists on the disc then there are two things that we need to guard against; both arising from the fact that the previous file will be overwritten by the new one of the same name. Firstly, does the user really want to obliterate the old file; in other words, was the entry of the same name deliberate? We need a routine to warn the user that a file is about to be overwritten and give him a chance to escape. The second problem arises only with the DFS and if a user does choose to overwrite an old file. It results from the fact that all files on the Acorn DFS must occupy contiguous parts of the disc. If you save a data file onto the disc it occupies a certain space and if this file is closed and then some other information written to the disc, the new file will start at the beginning of the next sector. This means that you cannot replace the first data file with another file of the same name if this is much longer than the original. An attempt to do so will give a "Can't extend" error. This problem does not arise with the ADFS and CP/M; neither of which require files to be contiguous on the disc.

Ideally you should protect the user from ever running into the problem of not being able to extend a file. If all the data files that can be written are of the same length then this will not happen. This is the case, for example, in the key strip program in Appendix 1, and so replacement of an old file by an update under the same name will always be possible. If this is not the case, however, there are two ways in which you can arrange things so the user is not troubled. The simplest way is to pad out all files with spaces or dummy numbers so as to make each one as long as the largest. This may well use up a lot of disc space and is only really to be recommended when files are not too disparate in length, otherwise you could spend most of the disc access time writing the padding for some files. A far more satisfactory solution is to detect the fact that a file is about to be overwritten and then to delete it before writing the new version. This will result in the new version being added at the end of the free disc space rather than in the place occupied by the previous file, and so there will be no problems in extending the length. The only snag with this approach is that the disc will rapidly begin to run out of free space because the deleted files will generally leave unusable holes. The space can be recovered by compacting the disc and if you adopt this technique you should ask your users to compact their data discs regularly.

It is a key element of all the techniques that I have been discussing in this section that we should be able to detect whether a file with the same name as the one the user has just entered already exists on the disc. This can most conveniently and generally be achieved with the sequence shown in listing 6.4.

#### Listing 6.4

10 MODE 6

20 REPEAT a$="Y"

30 INPUTLINE"Enter a filename "f$

40 IF FNfile\_exists(f$) THEN PRINT"\*THIS FILE ALREADY EXISTS D

O YOU WISH TOOVER-WRITE IT WITH A NEW VERSION ?(Y/N)\*";:REPEAT a$

=GET$:UNTIL INSTR("yYnN",a$)

50 UNTIL INSTR("yY",a$)

60 PROCsave(f$)

70

80 DEF FNfile\_exists(f$)

90 C$=OPENUP f$

100 CLOSE #C%

110 IF C%=0 THEN =FALSE

120 =TRUE

What is done here is to open the file for update by assigning a channel number to it and then close it again immediately. BASIC 1 will require OPENIN rather than OPENUP. A non-existent file is always given a channel number of zero, but no error message is issued until you try to use the channel, so this provides a convenient way of detecting the previous use of a filename. The technique will work equally well on the DFS, ADFS and CP/M, since it makes use of the BASIC rather than the filing system. If you are using the DFS and want to delete the old file before re-writing it all that needs to be done is to add the extra line:

55 OSCLI("DELETE "+f$)

With the ADFS there is a possible pitfall in listing 6.4 if the user enters a filename that corresponds to an existing file that does not have read/write status. An attempt OPENUP such a file will result in the filing system error "Access Violation", error number &BD. You will need to make sure that your general error trap will cope with this and I would recommend that you change the error message to something like "The filename just entered is being used elsewhere and is not available to you – Please try again". This will cope with the possibility that someone may enter a filename that corresponds to a directory file on the disc, which you would certainly not want to overwrite. Directory files have read only status when created and should not be tampered with.

### Alternatives to User Input of Data Filenames

The usual situation in which you will want a user to enter filenames is to enable him to create a library of identifiable files for retrieval by a particular piece of software. The advantage of the user being able to choose filenames for himself is that he can use names that are relevant to the file contents and so make the retrieval of specific files easier. In a wordprocessor, for example, all letters might be saved into files whose names end in L.

The utility of this approach is limited, however, by the fairly restricted length of filename allowed by most disc filing systems. There is added to this, from the programmer's point of view, the precautions which are needed as a result of allowing the user to enter strings that are used directly as filenames. Both of these limitations can be overcome, however, by completely changing the strategy of data file identification.

The simplest approach is to allow the user only to enter an identifier for each data file and then to supply the rest of the filename from within the program. For example, instead of prompting for a filename the user can be asked for a file number, N%, and then the filename can be set to the string DATA+STR$(N%). The problem here, however, is that the number gives the user no scope for a mnemonic identification of files but this can be provided separately and in a form which is far more flexible than is permitted by direct filename entry. What is required is that when the user saves a file he is prompted for the file number and then for a file description, which can be a string of any length that you choose to offer. The file is saved under a name derived from the number and the description is read into a variable which can be linked to the file number. The most obvious way to do this is to use a string array with the descriptor string for file N% being held in array element N%. The file descriptions can then be saved on the same disc as the data and retrieved for display to the user instead of the disc catalogue whenever files are to be loaded or saved. Moreover the list of file descriptions can be used to construct a menu so that the user can be presented with a numbered list of the file descriptions, from which he can select the required data file by entering its number. This technique provides by far the most friendly method of data file handling available. The user is almost completely isolated from the filing system and yet can identify his files clearly.

#### Listing 6.5

10 MODE 4

20 DIM descript$(29):FOR i%=0 TO 29:descript$(i%)=STRING$(25,"

"):NEXT

30 PRINTTAB(0,10)"PLEASE LOAD A DATA DISC IN DRIVE 0 OR A BLANK

BUT FULLY FORMATTED DISC IF YOU WANT TO PREPARE A NEW DATA DISC

. PRESS THE SPACE BAR WHEN READY TO CONTINUE."

40 REPEAT UNTIL GET=32

50 IF NOT FNfile\_exists("DESCRIP") PROCsave\_des ELSE PROCload\_d

es

60 ON ERROR CLOSE#0:IF ERR=17 THEN 70 ELSE CLS:REPORT:E$=INKEY(

1000):GOTO70

70 REPEAT:CLS

80 PRINTTAB(18)"MENU"

90 PRINTTAB(8,5)"1. SAVE DATA TO DISC."

100 PRINTTAB(8,7)"2. RETRIEVE DATA FROM DISC."

110 PRINTTAB(8,9)"3. SEE LIST OF FILES."

120 PRINTTAB(8,11)"4. END PROGRAM."

130 REPEAT

140 INPUTLINETAB(8,13)"Please Choose "a%

150 UNTIL a%>0 AND a%<5

160 IF a%=1 THEN PROCsave

170 IF a%=2 THEN PROCselect\_file

180 IF a%=3 THEN CLS:PROCdisplay:PRINTTAB(0,31)2\*PRESS SPACE BAR TO RETURN TO THE MENU\*";:REPEAT UNTIL GET=32

190 UNTIL a%=4

200 CLS:END

210

220 DEFFNfile\_exists(n$)

230 C%=OPENUPn$

240 CLOSE #C%

250 IF C%=0 THEN =FALSE ELSE =TRUE

260

270 DEFPROCdisplay

280 LOCAL i%:flag%=FALSE

290 FOR i%=0 TO 29

300 IF FNfile\_exists("DATA"+STR$(i%+1)) THEN PRINT,i%+1;" ";desc

rip$(i%):flag%=TRUE ELSE PRINT

310 NEXT

320 ENDPROC

330

340 DEFPROCselect\_file

350 LOCAL n$,n%

360 CLS:PROCdisplay

370 IF NOT flag% THEN PRINTTAB(0,31)"\*NO FILES EXIST ON THIS DIS

C-PRESS SPACE BAR\*";:REPEAT UNTIL GET=32:ENDPROC

380 ENDPROC

390 ENDPROC

400 PRINTTAB(0,30);SPC(40);TAB(0,30);

410 INPUTLINE"Choose File by Number "n$

420 UNTIL FNvalidno(n$):n%=VALn$

430 UNTIL FNfile\_exists("DATA"+STR$(n%))

440 OSCLI("LOAD DATA"+STR$(n%))

450 ENDPROC

460

470 DEFPROCsave

480 LOCAL temp$,n$,n%

490 CLS:PROCdisplay

500 REPEAT

510 PRINTTAB(0,30);SPC(40);TAB(0,30);

520 INPUTLINE"Number for New File ? "n$

530 UNTIL FNvalidno(n$):n%=VALn$

540 IF FNfile\_exists("DATA"+STR$(n%)) THEN PRINTTAB(0,30)"Overwr

ite Existing File ?";:REPEAT :a$=GET:UNTIL INSTR("YyNn",a$):IF IN

STR("Nn",a$) THEN 500

550 REPEAT CLS

560 PRINT"File No. "n$

570 PRINTTAB(0,10)"Enter file description-25 Characters Max"

580 INPUTLINE'temp$

590 UNTIL LENtemp$<26:descript$(n%-1)=temp$

600 CLS

610 PROCsave\_des

620 OSCLI"SAVE DATA"+STR$(n%)+" 8000+100")

630 ENDPROC

640

650 DEFPROCsave\_des

660 LOCAL i%

670 C%=OPENOUT("DESCRIP")

680 FOR i%=0 TO 29

690 PRINT#C%,descript$(i%)

700 NEXT

710 CLOSE#C%

720 ENDPROC

730

740 DEFPROCload\_des

750 LOCAL i%

760 C%=OPENUP("DESCRIP")

770 FOR i%=0 TO 29

780 INPUT#C%,descript$(i%)

790 NEXT

800 CLOSE#C%

810 ENDPROC

820

830 DEFFNvalidno(n$)

840 IF LENn$>2 OR LENn$<1 THEN =FALSE

850 IF VALn$<1 OR VALn$>30 THEN =FALSE

860 =TRUE

Listing 6.5 illustrates the coding needed for a maximum of thirty files. This is what would be required for the Acorn DFS since one file is needed to hold the descriptor strings. An alternative to using an array to hold the descriptions is to store them in a random access file on the disc. Random access disc filing is described in a later section in this chapter.

In Listing 6.5 descriptions of the files are held in the array descript$, which is stored on disc in the file DESCRIP. Line 50 checks to see if this file is present on the disc and if this is not the case it is created. If the file does exist then it is loaded into the array. In either case the contents of array and disc file are made identical and this situation is maintained at all times during the program. When a file is to be loaded or saved PROCdisplay, line 270, lists all the descriptions of files on the disc in the form of a menu with numbers from 1 to 30. The user merely selects the appropriate number to load a particular file.

During saves the user has to enter the file description, of up to 25 characters, and line 540 warns of attempts to overwrite an existing file offering an escape if required. If the files being manipulated are likely to be of differing lengths then you should take steps to avoid "Can't Extend" errors when replacing files with longer ones of the same name on the DFS. The techniques for doing this are discussed in the previous section. Line 430 deals with attempts to enter a number which does not correspond to a file which is currently on the disc.

A menu-driven data file handler of this type, although it requires a fairly substantial amount of coding to implement, is well worth considering, particularly for inclusion in programs which are intended for novice users.

### User Input of Filing and Operating System Commands – Star Commands

One way in which you can give users access to facilities in the filing system is to allow the direct, keyboard, entry of star commands. If such access is not limited in any way then the user can also send commands to the operating system, using the \*FX sequence, program the function keys with \*KEY and also communicate with any service ROMs in his machine. This latter facility can be extremely useful, particularly in the case of printer control ROMs for example. Some ability to access the operating system is especially useful to those with non-standard hardware, perhaps a printer with an unusual Baud rate, since it allows them to re-configure their system to suit the peripherals.

There are, of course, many potential problems for the user who plays about with operating system calls from within the program; he could turn off vital functions, such as the ESCAPE key or the keyboard as a whole for that matter, but this is only likely to happen to the compulsive fiddler. You cannot really allow for this kind of thing, not, at least, without restricting access to the operating system for legitimate purposes. What you must do, however, is to protect the user from attempting what might seem to be perfectly legitimate operations, but which will, in practice, be disastrous. Some of the filing system commands, for example, \*COMPACT and \*COPY, require the use of memory and if they are invoked from within a program corruption of the program itself or its data is very likely to occur, so such commands should be precluded.

Another area in which problems could arise is if the user changes disc drive or directory in such a way that the program can no longer access the files that it requires. This will be particularly disastrous if random access filing is in use and files are open. You should ensure, either that the user cannot carry out any damaging changes or that the system is always reset to the required status on exit from the star command handling.

**Listing 6.6**

10 MODE 6

20 ON ERROR IF ERR=17 THEN CLS:END ELSE PRINT'"\*ERROR";CHR$(7);

:REPORT:E$=INKEY$(500):GOTO40

30

40 REPEAT

50 REPEAT

60 INPUTLINE'"Command ? "comm$

70 valid%=FNallowed\_command(FNstrip(comm$))

80 IF NOT valid% THEN PRINT"\*BAD COMMAND\*";CHR$(7)

90 UNTIL valid%

100 OSCLI(comm$)

110 UNTIL FALSE

120

130 DEF FNallowed\_command(f$)

140 REM\*MUST BE AT LEAST ONE CHARACTER\*

150 IF LENf$<1 THEN =FALSE

160 REM\*NO ABBREVIATIONS\*

170 IF INSTR(f$,".") THEN =FALSE

180 LOCAL t$,i%,temp$

190

200 REM\*CONVERT TO HIGH CASE\*

210 FOR i%=1 TO LENf$

220 char$=MID$(f$,i%,1)

230 IF char$>="a" AND char$<="z" THEN char$=CHR$(ASC(char$)-32)

240 temp$=temp$+char$

250 NEXT:f$=temp$

260

270 RESTORE 350

280 REPEAT

290 READ t$

300 REM\*CHECK FOR DISALLOWED COMMAND\*

310 UNTIL LEFT$(f$,LENt$)=t$ OR t$="SLURMP"

320 IF t$<>"SLURMP" THEN =FALSE

330 =TRUE

340 REM\*DISALLOWED COMMANDS\*

350 DATA DRIVE,RUN,COMPACT,COPY,SLURMP

360

370 REM\*STRIP LEADING \*'s AND SPACES AND TRAILING SPACES\*

380 DEF FNstrip(a$)

390 IF LEFT$(a$,1)<>" " AND LEFT$(a$,1)<>"\*" THEN 420

400 REPEAT a$=RIGHT$(a$,LENa$-1)

410 UNTIL LEFT$(a$,1)<>" " AND LEFT(a$,1)<>"\*"

420 IF RIGHT$(a$,1)<>" " THEN =a$

430 REPEAT a$=LEFT$(a$,LENa$-1)

440 UNTIL RIGHT$(a$,1)<>" "

450 =a$

Listing 6.6 demonstrates one way of restricting user access to star commands and you can extend the range of prohibition to suit your application by simply adding commands that are not permitted into the data statement at line 350. In order to simplify the coding I have also stopped the user of any abbreviated form of commands, terminated by a full stop, by the inclusion of line 170. This is not a major limitation and if it is not included then the list of prohibited commands becomes very long. \*CAT, for example, can equally be invoked by \*., \*C. and \*CA. and all would need to be included in the data. Leading spaces and leading stars have to be stripped from the input line since the command line interpreter will itself do a similar strip and so, for example, \*DRIVE would not be intercepted by would be correctly dealt with by the command line handling and so would elude our trap. Needless to say, you will have to ensure that your ON ERROR trap can cope with any completely erroneous commands which may be entered.

One problem with the user entry of star commands is that if a command which is not represented in the operating system, the filing system on in a utility ROM is entered, it will be looked for on the discs and an attempt made to run a file of that name as a program. This is fine provided that there are discs in the currently selected drive, and in the drive specified as the library, and if there is *no* file with a name corresponding to the \* command entered. If this is the case, a 'Bad Command' message will be given and the ON ERROR trap triggered.

If, however, there is no disc in either one of the appropriate drivers then the machine will hang with possibly disastrous results as far as any data is concerned. A similar catastrophe will occur if the user inadvertently enters an unrecognized star command which corresponds to the name of a file on the disc, especially if it is not actually a runnable file. The machine will have a go anyway and the result will often be a complete crash. There is nothing you can do to provide protection from either of these pitfalls, they are really the price to be paid for the rather dubious advantage of being able to use \*FILE as an abbreviation for \*RUN FILE or \*EXEC FILE, the latter being available only on the ADFS. If it is any consolation, I have never actually fallen foul of these problems in practice, that is until I tried to do so deliberately when writing this chapter, but any user could be rather less lucky!

### Random Access Disc Filing

If you have a data file that is to be accessed, perhaps for amendment, the most direct strategy is to load the entire file into memory, perform the required manipulations and then rewrite the entire file back to storage. If you are operating with tape then this is the only option, but with discs there is an alternative. If you know exactly how the file is organized on the disc you can simply position the disc read/write head at the place where you want to make alterations and change the file contents directly on the disc, without having to read it into memory. This technique is called random access filing and is only possible with a disc system since you cannot locate a specific point on a tape. The major advantage of the method is that since the entire file never has to be stored in memory then its size is only limited by the space available on the disc and so files of several hundred kilobytes are possible.

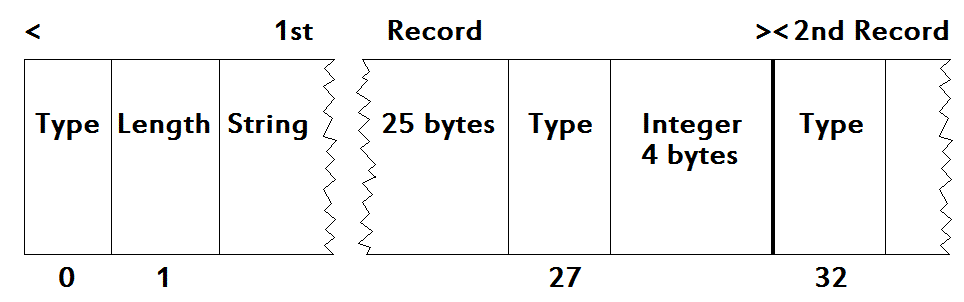
BBC BASIC, unlike many other versions of the language, does not have a pre-packaged random access filing capability. Rather it provides you with the tools to build your own system tailored to whatever specific application you have. In order to access files in this way you use a pointer, which can be moved to any place in the file, and then you can read from, or write to, the file at the pointer position. In BBC BASIC the pointer can be moved one byte at a time and facilities are provided to read and write either individual bytes or entire variables. In order to be able to operate the system at all it is clearly going to be necessary to organize the file in some predetermined and regular pattern so that any particular piece of information can be found. The file is normally regarded as a series of records, with each record being subdivided into fields.

A file that is to be handled in this way must be simultaneously open for reading and for writing. The command to do this is C%=OPENUP(filename$) in BASIC 2. In BASIC 1 OPENIN is used instead. The channel assigned to the file is stored in C% and the pointer associated with the file is moved by setting the pseudo-variable PTR#C% to a value which represents a number of bytes from the top of the file. When the file is first opened, the value of the pointer is zero, meaning that it points to the first byte in the file. The pointer moves either in response to the PTR# command or whenever a read or write takes place, where it is left at the position in the file immediately after the last byte read or written. The pointer is a little like the cursor on a screen with the direct cursor movement commands corresponding to resetting the value of PTR#. The cursor also moves to the next free position when you write to the screen.

The first job to do when setting up a file to be used in random access mode is to decide upon the exact format and length of each record and so determine the resulting layout of the file's contents. Different types of variables occupy differing amounts of disc space and so this must be taken into account when performing the calculations and each entry in the file is preceded by a 'type' byte, which specifies whether it is a real number, an integer or a string. In addition, a string entry has, as its second byte, after the type, a length byte. To read a variable from the file the pointer must be set at its type byte and if you try to read the information into the wrong type of variable a 'Type Mismatch' error will be signalled. The type compatibility rules are the same as in BASIC as a whole, you cannot read numbers into strings, or vice versa, but reals and integers are mutually compatible.

Integers occupy five bytes of file space, four for the number plus the type byte, reals require six bytes and the space occupied by strings is equal to their length plus 2 bytes, for the type and length indicators. A file that consists only of numbers is, therefore, fairly easy to deal with. If they are all integers, for example, the Nth file entry will be accessed by moving the file pointer to position (N-1)\*5. Notice that the pointer starts at zero so the first number is at position zero in the file, the second at 5 etc. If the numbers were reals then the expression would need to be changed to (N-1)\*6. Strings are a little more complex to deal with because of their variable length and in practice it is best to simplify the problem by padding out strings, if necessary, to a standard length. If you do not do this it is almost impossible to operate the random access file since you cannot predict where a string will start. Moreover, if you replace on string by another that is longer, then the new string might very well overlap the beginning of the next entry and so completely corrupt it. The padding technique may mean that you are consuming substantial amounts of disc space with non-information but there is really no alternative. A file of strings of length 25 characters each will be read by setting the pointer to (N-1)\*27.

There is no reason of course why you should not mix variables of different types in the same file, but you must still keep track of the file structure and it is here that the idea of division into records and fields comes into its own. The record is a repeating unit in the file, each of exactly the same length, and the field is a subdivision of a record organized in some way that the internal structure of each record is exactly the same as that of any other record. Figure 6.1 shows a simple example of this idea with a record which consists of a 25 character string followed by an integer. The record length is 27 + 5 = 32 bytes and the string is always found at a position 27 bytes beyond the start of each record. The string would be regarded as field 1 and the integer as field 2 so that the pointer can be set to the Fth field of the Rth record by making PTR# equal to (R-1)\*32+(F-1)\*27, in this example. In BBC BASIC it is very easy to set up a function which will return the correct pointer value for any field and record.



**Figure 6.1.** Disc File Structure

**Listing 6.7**

10 MODE 6

20 ON ERROR CLOSE #0:IF ERR=17 CLS:END ELSE REPORT:END

30 REM\*CREATE FILE\*

40 C%=OPENOUT"DATA"

50 FOR I%=1 TO 31

60 data$="THIS TEXT IN RECORD-"+STR$(I%)

70 IF LENdata$<25 THEN REPEAT data$=data$+" ":UNTIL LENdata$=25

80 PRINT#C%,data$,I%

90 NEXT

100 CLOSE #C%

110

120 REM\*OPEN FILE FOR RANDOM ACCESS\*

130 C%=OPENUP"DATA"

140 REPEAT

150 REPEAT

160 INPUT"Record and Field "rec%,f%

170 UNTIL rec%>0 AND rec%<32 AND (f%=1 OR f%=2)

180 REM\*MOVE POINTER\*

190 PTR#C%=FNsetpointer(rec%-1,f%-1)

200 IF f%=1 INPUT#C%,text$:PRINT text$ ELSE INPUT#C%,num%:PRINT num%

210 UNTIL FALSE

220

230 DEF FNsetpointer(rec%,f%)

240 =rec%\*32+f%\*27

Listing 6.7 implements the example that I have just been discussing, with the string and the integer both simply being record identifiers. The file must first be set up with an OPENOUT command, as at line 30 and then when this is executed the filing system reserves space on the disc. In the DFS this is initially 16K whereas the ADFS reserves 64K. All of this may well not be required but it is initially allocated, the file length being reduced to the amount of disc space actually required by the data when the file is closed at line 100. This method of allocation of space will result in a Disc Full errors being reported when the file is opened if there is not enough room for this initial allocation, even though there may well be plenty of disc space for the required data to be written. If, in the event, more space than initially allocated is required by the data then, when the end of the current space is reached, the DFS will reserve a further 16K and the ADFS 64K, and so on until the disc is full or the data ends.

Notice the padding at line 70 of Listing 6.7 to ensure that all the strings held are exactly 25 characters in length, so as to keep the record size uniform. Line 130 opens the file for random access, using OPENUP, in BASIC 1 OPENIN should be used, and then you are asked to enter a record and field number. The function at line 230 calculates the pointer setting for use in line 190 which actually moves the pointer. Line 200 does the required read from the file.

You might have been wondering why I chose to pad out the strings used in this example to 25 characters, giving a record size of 32 bytes. This is *not* accidental! I deliberately wanted to have a record length that was divisible, exactly, into 256. To understand the advantage of this, we must look a little more closely at the way in which the DFS and ADFS actually handle open files. I initially implied that moving the pointer in a file was essentially akin to moving the read/write head on the disc surface, but if this were literally true then every time the pointer moved, the discs would have to be accessed. This would not only be rather slow but would result in a lot of wear and tear on the drives. What actually happens is that the system employs a buffer, in the computer memory, and loads the data from the disc into that buffer in 256 byte blocks. Provided, therefore, that you do not move the pointer outside the current block in the buffer, the drives will not actually need to come on. If you run the program in listing 6.7 and ask for record 1, field 1, the drives will run to load the first 256 bytes. If you then work your way up the file a record at a time, you will not hear the drives accessed again until your reach record 9, when the next 256 byte block is required. If you now return to record 8 the drives will run again, to go back to the previous block if you are using the DFS, but you will hear no drive activity with the ADFS. This is because the ADFS actually holds up to five 256 byte blocks of data in memory at any one time, always the last five which you accessed, so when you initially went to record 9 and the new block was loaded the previous block was not discarded and is still available when you want it again for record 8. Random access filing is, therefore, potentially much faster on the ADFS.

Both DFS and ADFS, however, handle their buffers in 256 byte blocks and, for this reason, it is a good idea to make the length of each record in your file either divisible into 256 or a multiple of 256, even if some padding is necessary. If you do not do this, you might find that the drives have to be activated in the middle of the retrieval of a variable, creating a delay in the program operation.

In Listing 6.7 we created the data file in a rather artificial manner within the program. In reality you will probably be entering the data from the keyboard and, moreover, the entries will not all be carried out in the same session. You will start off with a file of a particular length and then, at the next usage, you will want to open it again and add new data onto the end. This is a similar situation to the one which we discussed earlier with respect to the replacements of a file with a longer one with the same name. On the ADFS and CP/M there is no problem, but with the DFS there is the risk of a "Can't Extend" error if another file has been written to the disc in the meantime. The best way to avoid this problem with the DFS is to create a padded out dummy file of the greatest length that will ever be required by your data. There will then be no need to extend the file, the growing data can merely take over the reserved space. To illustrate this, let us suppose that the data in the file created in listing 6.7 represents only the initial entries and that we are going to add a further 50 entries at a later stage. The space can be made secure by including an extra line:

95 FOR I%=1 TO 50:PRINT #C%,STRING$(25," "),I%:NEXT

to write dummy data. If you have software to be used with the DFS it is often a good idea to include a disc preparation program to which the user must submit any disc before using it for data. The program should write the dummy files and so preclude any problems associated with extending the data at a later stage. This kind of ruse will, of course, be quite unnecessary with the ADFS and CP/M which do not require files to occupy contiguous areas of the disc.

### Manipulating The Contents Of Random Access Files

If you have a data set resident in a random access file then you will certainly be interested in performing manipulations such as sorts, insertions and deletions. Let us, for the moment, consider the case of a deletion of an entire record. One way of doing this would be to move all the following records up the file by an amount equal, in bytes, to the record length. This is rather like the way you would delete from an array by 'rolling up', but would be a very cumbersome approach requiring the records to be taken into memory and then rewritten to the disc one at a time. It would be very slow due to the disc access required, especially if the deletion was being performed on a record early in the file. There is, fortunately, a far more satisfactory solution involving leaving the disc format unchanged but taking over the file management by creating a set of 'virtual' record numbers which are indirectly related to the record numbers as represented in the actual disc file. Before returning to the rather more complex example of a deletion let me try to explain the way the technique works with a rather simpler case.

Suppose, for the sake of argument, we had a file of 50 records stored on disc and we wanted to reverse their order, so that record 50 became record 1, record 49 becomes record 2, etc. If we do not want to move the records around on the disc then we can effectively achieve our object by making the software respond to a request to retrieve record 1 by going to the end of the file and actually getting record 50. As far as the user is concerned, record 50 has become record 1. To do this we must construct a 'look-up' table which relates the ordering of the records, as we want to see if from our point of view, to their actual positions on the disc. Such a table is best held as an array, with the array indexes representing the virtual record numbers and the actual contents of the array being the true record numbers. In the example given the array element zero would contain 50 and we would call up the record not by setting the pointer using (R-1), where R is the record number, but with (array%(R-1)-1). Listing 6.8 illustrates this with a modified form of Listing 6.7.

**Listing 6.8**

10 MODE 6

20 REM\*SET UP POINTER ARRAY\*

30 DIMpointer%(30)

40 REM\*REVERSE NUMBERING\*

50 FOR I%=0 TO 30:pointer%(I%)=31-I%:NEXT

60 ON ERROR CLOSE#0:IF ERR=17 CLS:END ELSE REPORT:END

70 REM\*CREATE FILE\*

80 C%=OPENOUT"DATA"

90 FOR I%=1 TO 31

100 data$="THIS TEXT IN RECORD-"+STR$(I%)

110 IF LENdata$<25 THEN REPEAT data$=data$+" ":UNTIL LENdata$=25

120 PRINT#C%,data$,I%

130 NEXT

140 CLOSE #C%

150

160 REM\*OPEN FILE FOR RANDOM ACCESS\*

170 C%=OPENUP"DATA"

180 REPEAT

190 REPEAT

200 INPUT"Record and Field "rec%,f%

210 UNTIL rec%>0 AND rec%<32 AND (f%=1 OR f%=2)

220 REM\*MOVE POINT ER\*

230 PTR#C%=FNsetpointer(pointer%(rec%-1)-1,f%-1)

240 IF f%=1 INPUT#C%,text$:PRINT text$ ELSE INPUT#C%,num%:PRINT num%

250 UNTIL FALSE

260

270 DEF FNsetpointer(rec%,f%)

280 =rec%\*32+f%\*27

When this program is run you will find that requesting record 1 now gives you what was originally written, at line 120, as record 31. The array, pointer%, is initialized at line 50 as to reverse the record numbering and then line 230 uses the appropriate element of pointer% to call FNsetpointer, rather than the record number itself, the latter now having relevance only to the disc order.

Study Listing 6.8 carefully and you should be able to grasp how the virtual numbering system works. Any manipulations on the file order now only require the manipulation of the pointer array and the disc can be left untouched. Let us now return to the record deletion problem. Suppose we have 50 records and we want to delete the record with virtual number 20, which has its actual number in the 19th element of the pointer array. We simply copy the contents of this element into a temporary variable and then 'roll up' the array so that element 19 is overwritten by 20, 20 by 21, and so on until we reach the end of the array. We then record the fact that we only have 49 records active and place the value from the original element 19 into 49. This element points to a record on the disc that is no longer active, but when we want to extend the file again this space will be used for the new entry. The coding required for the deletion would be something like:

10 pointer%(19)=temp%

20 FOR i%=19 TO 48

30 pointer%(i%)=pointer%(i%+1)

40 NEXT

50 pointer%(49)=temp%

60 no\_of\_records%=no\_of\_records%-1

Do not let the discrepancy between the record numbers and the array indices confuse you. In order not to waste an array element the pointer to record 1 is held in pointer%(0). This arises from the fact that arrays start from element zero in BBC BASIC, as is the case with most versions of the language.

The idea of using an array to contain a look-up table of pointers to the disc file can be applied to the manipulation of the order of records in files for any purpose. Sorting and the insertion of records are obvious other areas of application.

### Auto Booting

This is a technique that is probably fairly well known but I include it here for completeness. The Acorn disc filing systems can both be made to carry out a set of instructions automatically when a disc is loaded into drive zero and SHIFT is held down whilst BREAK is pressed and released. You have, no doubt, encountered this as the standard method of starting commercial software.

In order to implement it yourself, you must do things: set a boot option and prepare a file called !BOOT. The boot option on the disc is set by the user of the \*OPT 4,n command, where N is a number between 0 and 3. The default setting is N=0, which means that the disc will ignore any attempt to auto boot it with SHIFT/BREAK. When N is 1, 2 or 3, however, the auto boot process initiates a search for a file on the disc called !BOOT. If N=1 an attempt will be made to \*LOAD the file, N=2 will load the file and then an attempt will be made to run it as a machine code program, whereas OPT4,3 will expect !BOOT to be a file which can be \*EXECed. The last of these options is the most commonly used and the DFS contains a utility, \*BUILD, which allows an executable file to be constructed. Alternatively, such a file can be prepared on a wordprocessor and the file spooled under the name !BOOT. The ADFS also has \*BUILD but it is not an integral part of the firmware; rather, it is on the systems disc.

To build a !BOOT file which can be executed you simply type \*BUILD !BOOT at your keyboard with the disc that is required to carry the file in the default drive. A line number will appear and you then type in the sequence of commands that you want to be executed on auto boot, just as you would if you were issuing them from the keyboard for direct action, including pressing RETURN. Every time that you press RETURN a new line is started with a fresh line number. When the command sequence is complete, press ESCAPE and the file is prepared. All that is required to complete the process is to type \*OPT 4,3 and the disc will now auto boot in a thoroughly professional manner.

The main problem with using \*BUILD to prepare your !BOOT file is that if you make a mistake you have no simple way of editing the file, and must start from scratch again. The alternative is to type the commands to be included in the file into a wordprocessor such as *View* or *Wordwise* and then save the file under the name !BOOT but using the spool option. If you make a mistake you then have a file that can be reloaded into the wordprocessor and edited.

You should always make your discs auto boot in this fashion, even if they only require the simplest of !BOOT files such as CHAIN"MAINPRG", since this will give the user the most user-friendly introduction to the software. If you do not do this he will need to be told what file is to be loaded and run.

### The CP/M Filing System on the BBC with Z80 Second Processor

When the BBC Micro is fitted with the Z80 Second Processor, the machine is expected to be run under CP/M and the Z80 package includes all the necessary systems software and utilities. The Acorn DFS is not used under these circumstances and all filing is handled by the CP/M so some modifications to software which accesses the DFS may be required if it is to run in the Z80, although changes should only be minimal since substantial effort has been expended by the designers to ensure the maximum degree of compatibility between the two filing systems as accessed from the respective BASICs.

CP/M is, of course, far more than a filing system, it provides a complete software environment and, since the CP/M itself can be mounted on any Z80-based machine with adequate hardware, then software portability is greatly improved. In this section, however, I am going to be concerned mainly with the filing system provided by CP/M and its compatibility with the Acorn DFS.

A special version of BBC BASIC is provided to run on the Z80, but to all intents and purposes it is compatible with the 6502 version. There are some technical differences which I will deal with in Chapter 7, but a correctly written program in the 6502 dialect will transfer to the Z80 with minimal change, although the actual disc file holding the program will not load directly under Z80, it will need to be translated into the CP/M format. A utility is provided with the Z80 package to do this for you. Any assembler source code or machine code will need to be completely rewritten of course since the two processors are quite different.

The handling of the normal type of sequential data file is exactly the same in the two BASICs and no changes will be necessary. The routines which I have discussed for such things as detecting the existence of a file will work equally well under CP/M. User entry of filenames can proceed in the same way, except that CP/M allows up to eight characters in a name and the directory is not implemented as such. Those familiar with CP/M will know that the filename in a CP/M system can have a three character extension. This can be included in filenames sent from BBC BASIC but if it is not supplied a default of .BBC is used. This distinguishes all files relevant to BBC BASIC and when the disc is catalogued from within the BASIC using an unqualified DIR command only files with the BBC suffix are shown. To see the whole disc catalogue you must send DIR \*.\* using the general wildcards. One way of exploiting the filename extensions is to use them to identify data files belonging to a particular piece of software by having the program supply a characteristic version of the file suffix.

All data files for a graphics program might end their names with '.PIC', for example. You can then show the user only the data files of relevance by cataloguing the disc with the command DIR \*.PIC. Do remember, however, that some filename extensions have a particular significance in CP/M and should be avoided for general purposes. A list of these appears in the *User Manual* for the Z80 second processor.

Random access filing routines written in 6502 BASIC will need to be rewritten for the Z80 version since the organization of data on the disc is quite different. There are no type bytes, so the "Type Mismatch" error never occurs, and integers and reals are not differentiated, both occupy 5 bytes. Strings have neither type byte nor length byte but are terminated with a carriage return, so a string occupies file space equal to its length plus 1 byte. Note of these differences will matter, of course, unless you are accessing a variable in a file by setting the value of PTR# and this must then point to the first byte of the variable to be read or written. Since there are no type bytes, you can, generally speaking, read any type of variable from any point in the file so if you do not position the pointer correctly you will still get a value and no error message. If, for example, you are trying to read a number and the pointer is not where you suppose it to be the system will merely read the next five bytes and interpret them as a number anyway. This can be very confusing so software using random access filing in Z80 BASIC should be coded carefully and rigorously checked.

The other major difference between filing on the DFS and under CP/M is that the pseudo-variables, EOF and EXT do not behave properly. In the DFS EOF becomes true when the end of the data is reached and EXT contains the length of the file. A CP/M file is always organized in 128 byte blocks and so EOF may still be false when the end of the data is reached, it only becomes true when the end of the last block is indicated by the pointer. Similarly EXT gives the file length to the end of the block and not the end of the actual data. This means that files which do not have lengths which are exact multiples of 128 will have a trailing area of garbage. Since, in practice, most files will be in this category it is not possible to read data from a file UNTIL EOF is true, for example, unless your application can tolerate reading this garbage.

As I have already mentioned, CP/M does not require files to occupy contiguous areas of space on the disc so there is no such thing as a "Can't Extend" error and, therefore, no need to take any steps to reserve disc space as is the advisable course with the DFS.

### Filing System Commands under CP/M

The Acorn filing system is not used under CP/M but the Z80 BASIC has been provided with an interface to the CP/M which looks, to the programmer, very like the Acorn DFS. Commands are sent prefixed by a star, in which the BBC user will regard as the normal manner. What has been done is to construct a pseudo-operating system which scrutinizes every star command sent from the BASIC. This intercepts \*SAVE, \*LOAD, \*DRIVE and \*TYPE and translates them into CP/M equivalents so that they appear to work in much the same way as with the DFS. There are syntax differences, notably that a drive must be A: or B:, the colon being essential, and the load address in \*LOAD is not optional. The pseudo-operating system also recognizes \*DIR, which is the CP/M equivalent of \*CA, \*RESET which initializes a disc and \*CPM or \*BYE which exit from BASIC to CP/M monitor level. Any other star commands are passed through the normal channels, first to the ROMs in the I/O processor and finally to the discs.

If you allow users general access to star commands from the Z80 then it is important that you check for potential disasters such as \*BYE and intercept them. There should be no need for user access to filing system commands under CP/M but you may want to leave some FX calls available for system reconfiguration.

### Filing System Errors Under CP/M

CP/M does not pass its errors back to BASIC in the way that the Acorn filing systems do. If an error occurs in the disc operating software, an error message will be sent from CP/M and displayed, irrespective of the BASIC ON ERROR routine's contents. The program will be suspended at this point and the normal procedure within CP/M is to press either return or CTRL/C to re-boot the filing system. If you do either of these in BASIC you will return to the CP/M and lose everything!

There is, in fact, no absolutely safe way of getting out of this predicament within BASIC but normally pressing ESCAPE will clear the error. It is essential, therefore, that your ON ERROR trap can pick up the ESCAPE and deal with it safely. Users should be warned not to press ESCAPE until the discs stop running otherwise there will probably be a crash. One point to be aware of, by the way, is that the drive that was logged in on the CP/M when the error occurred will be retained after the escape. This can cause a problem if, for example, the error happened when saving a file under the name B:FILENAME whilst the program requires the logged in drive to be A for most purposes. The save involves temporarily logging drive B, but normally drive A would be re-selected afterwards. If an error occurs, however, this will not happen and drive B will be the default after the error is cleared. If it is essential that you go back to drive A then it must be logged explicitly with \*DR. A: or equivalent in the ON ERROR trap.

### Program Discs and Auto Boot under CP/M

The hardware configuration under CP/M is absolutely standard, two double-sided eighty track drives are always required although these are treated by the system as two rather than four drives since both sides of the discs are addressed as one continuous surface. The drives are designated A and B and the default drive is known as the 'logged in' drive. Unless you change it, the logged in drive is drive A, which corresponds to drives 0 and 2 on the Acorn DFS. The normal practice with CP/M software would be to have the program and any necessary system software on drive A is usually referred to as the 'program disc' but It must contain not only the program itself but the CP/M system software generated by the SYSGEN command. SYSGEN only reproduces the basic set of CP/M commands which allow the disc to be catalogued and files to be erased and renamed. Other facilities are available from separate files and these will have to be present on the program disc as well if your package requires them.

When the Z80 first comes on there is no language present nor is CP/M installed. You have to load a CP/M system disc to activate CP/M and then load BBC BASIC from disc if this is the language required. You are now in the position that the ordinary BBC finds itself in at switch on and can chain in the program to be run. The process can be short-circuited to some extent by booting CP/M and the entering BBCBASIC FILENAME while FILENAME is the name of the file in which the BASIC program is stored. The effect is the same as selecting BBCBASIC and then typing CHAIN "FILENAME".

This is a rather cumbersome sequence of events for an inexperienced user to follow and your object in producing a piece of software to be run under CP/M should be to prepare a program disc that can be loaded into drive A and then booted up by pressing CTRL/BREAK. The user should then find the program up and running and apart, possibly, from responding to a prompt to load a disc into the other drive for data this should be the end of his contact with the filing system. A program disc for use in this way will need to carry tour BASIC program, the Z80 BASIC and the CP/M system files created by the SYSGEN command. The auto boot function also needs the file SUBMIT.COM, which is supplied on the CP/M disc and you will have to create your own boot file to be run at auto boot. This file must be called BOOT.COM if it is a machine code program and BOOT.SUB if it is a text file which consists of a set of CP/M commands to be executed on boot up.

The way the system works is that when the CP/M is initialized it looks at the disc and if a file called BOOT.COM is found this is loaded and run as a machine code program. If this file is not present, BOOT.SUB is looked for and the commands that it contains executed. The software to do this is in the file SUBMIT.COM, the results are exactly equivalent to setting up a !BOOT file on the Acorn systems and invoking \*OPT 4,2 and \*OPT 4,3 respectively.

The BOOT.SUB file can be created with the CP/M editor. To do this load the disc that you want the file to appear on in drive B and a CP/M system disc with the file ED.COM in drive A and then enter the editor with ED B:BOOT.SUB. The simplest BOOT.SUB file will merely contain the single line BBCBASIC FILENAME, terminated by a carriage return and the program disc will then load the language and chain in the program when the system boots.

The BBC Micro with a Z80 second processor running CP/M is a very powerful machine and is particularly suited to business applications. With carefully written software the user need not have any interaction with the filing system or computer operating system but the CP/M environment produces a whole host of facilities for exploitation by the programmer. The CP/M filing system has none of the limitations on the number of files or the way they are arranged on a disc that is associated with the Acorn DFS and so is potentially much more friendly to user and programmer alike and particularly well suited to handling large data files of the type often required in business software.

### Conclusion

Good software will always make the user feel in control and this is nowhere more important than when access to the filing system is required. Ideally your user should be quite unaware of what is going on in this department whilst being confident that all is well. Make sure that your prompts about disc or tape loading are plentiful and clear and that there is an escape route from a mistake on the part of the user or a disc or tape problem and everything should run smoothly. One of the best ways to make your software unpopular is to make it easy for the customers to lose several hours work at the keyboard by making a simple mistake!

# Chapter 7.

# On Compatibility

The BBC Micro has undergone several changes in firmware since its introduction and, in addition, there are a number of possible hardware configurations, particularly second processors, the 6502 and the Z80. Ideally, any software intended for distribution should be compatible with as many configurations as possible, and you can often also ensure that programs will interchange between the BBC and the Electron. In this chapter I want to look at the points that have to be considered in maximizing such compatibility. The question of filing system interchange has already been covered, in Chapter 6, so I will not deal with it again here, rather I shall be discussing compatibility between the various versions of BBC BASIC, different operating systems and ensuring that second processors will cope with your software.

### Operating System Differences

The BBC Micro, when first introduced, has version 0.1 of the operating system. This lacked many of the FX calls available in the current, 1.2 version and had a number of other deficiencies. It may be that there are still some machines about which contain the old chip but I would imagine that any serious user will have upgraded long ago. If you want to have your software compatible with 0.1 then the FX calls which are to be avoided are those not marked with asterisks in the list on pages 418 and 419 of the *BBC User Guide*. The 0.1 operating system does not support discs so there is no need to worry about compatibility with disc software.

The next version of the operating system was number 1.0, issued when discs first became available on the BBC, and this was rapidly followed by the current standard 1.2 version. These two later versions of the OS seem, for all practical purposes, to be identical as far as software is concerned although they do use slightly different hardware. The Model B Plus has a new version of the operating system, 2.0, with some additional commands to cope with the shadow RAM but otherwise it is the same as version 1.2 and provided only the official calls are used and the shadow RAM is not accessed there should be complete compatibility in both directions.

The Electron has always had the 1.0 operating system which is essentially the same as the current BBC version. At least it accepts, without producing an error, all the commands available on the larger machine although they may not all be acted upon. For example, \*TV has no effect on the Electron, it is simply absorbed and ignored. This strategy allows BBC software to run on the Electron with a minimum of modification.

### The BBC and the Electron

The major difference between the BBC and the Electron is the lack of mode 7 on the smaller machine. BBC software which uses mode 7 will be run using mode 6 on the Electron. This may or may not be a successful strategy depending on what is actually being attempted. Any teletext control characters will produce gibberish on the Electron mode 6 screen and there will, of course, be no graphics or colour effects. If no teletext control characters are used, then the mode 7 display should translate entirely satisfactorily into mode 6. You can always try it out on your BBC, of course.

Apart from the teletext control code problem the other snag with transposing BBC mode 7 software to the Electron is the possibility that there may not be enough memory. Mode 6 requires 8K whereas the BBC's mode 7 takes up only 1K, so it is quite possible that a successful mode 7 program will run out of space in mode 6. The situation is made even worse if you are operating in an Electron with the ADFS since that filing system takes more memory than does the DFS in the BBC.

The Electron is a slower machine than the BBC, the exact speed difference depends on what is being done, but the smaller machine takes twice as long as the BBC for some operations. The relative slowness of the electron means that routines which are marginal as regards being sufficiently fast on the BBC will almost certainly be unacceptably slow when moved to the other machine. The other point to note is that any software timing, for example delays using empty FOR…NEXT loops, will not transpose correctly and it is far better to use the hardware timers such as the BASIC TIME facility.

The standard Electron has neither printer interface nor the joystick/ADC port attempts to use these are simply ignored. The machine will not hang, for example, if you try to use the printer on an unexpanded Electron, the printer sink is used. A Plus 1 interface is required to give the Electron these facilities and then it behaves essentially like a BBC.

### The Model B and the B Plus

The only major difference between these two machines from the programmer's point of view is the shadow RAM available in the Plus and that all Plus machines will have BASIC 2. Provided the shadow RAM is not used, program will transfer between the machines in either direction. The incompatible commands are \*SHADOW itself, its equivalent \*FX 114 call and mode selection with command MODE (n+128), where n is the mode number. There are also some new operating system routines associated with the shadow memory and these will only work in the Plus, of course.

### BASIC 1, BASIC 2 and HI-BASIC

Originally the BBC Micro had a chip which has become known as BASIC 1 and it is this version that is described in the current *User Guide* but the machines sold over the past year or so have all been fitted with BASIC 2 which has a few enhancement and has had some obscure bugs removed. To find out which version of BASIC you have type REPORT after resetting your machine with CTRL-BREAK or at switch on. You will get a copyright message from the ROM and if this is dated 1981 you have BASIC 1, BASIC 2 has a 1982 copyright data. If you have the earlier version, it is possible to upgrade but it is hardly worthwhile in most instances as the differences between the two are only small and the bugs in BASIC 1 are easily avoided. Any software which runs on BASIC 1 will be perfectly alright when transferred to the later interpreter but the compatibility does not work the other way and it is necessary to be aware of what is required when writing software in BASIC 2 to ensure that it will run on BASIC 1.

If you need to find out from within a program which interpreter is in use the easiest way is to use the ? indirection operator to examine location &8015. This is the last byte of the copyright message and contains ASC"1" in BASIC 1 and ASC"2" in the newer ROM.

The Electron has a version of BBC BASIC which is exactly the same in use as BASIC 2, and the HI-BASIC issued for use on the 6502 second processor is also syntactically the same as BASIC 2 and can be regarded as identical as far as programming is concerned. I shall be returning, in a later section, to the version of BBC BASIC used on the Z80 second processor but for most purposes it too can be regarded as being BASIC 2.

### Differences Between BASICs 1 and 2

There are many machines about with BASIC 1 chips in them and it is essential that any commercial software run under either version. This means, in practice, ensuring that BASIC 1 can be used since the upward compatibility is completely satisfactory.

### The INSTR Bug

This is noted in the *User Guide* as a known bug in BASIC 1. The symptoms are that if INSTR is used to search for a longer string inside a shorter one, which it will not find of course, when the routine is exited, it leaves the stack corrupted. This is only a problem if it happens within a subroutine of some kind, be it procedure, function , etc. since the return address is stored on the stack. The result will actually be a complete crash of the program when attempting to return from the subroutine.

The solution is to avoid any comparisons using INSTR where the second string could be longer than the first. A sequence such as:

100 IF LEN a$>LEN b$ THEN u%=0:GOTO 120

110 u%=INSTR(b$,a$)

120 REST OF PROGRAM

will achieve the desired effect, or, more generally, you could replace the use of INSTR in all cases with a call to the function FNinstr(b$,a$) which should be coded as:

DEF FNinstr(b$,a$)

IF LENa$>LENb$ THEN =0 ELSE =INSTR(b$,a$)

These modifications are not essential in BASIC 2 where the bug has been corrected but will still have to be included if you want the program to run successfully under BASIC 1.

### ON…GOTO or GOSUB and ELSE

In BASIC 2 it is possible to include an ELSE clause on the end of an ON…GOSUB or ON…GOTO statement to be executed when the value being tested is not found in the list. This prevents an error being caused by out of range values. In BASIC 1, however, although you can do this it has a similar disastrous effect to the INSTR bug if used in a subroutine so should be avoided in software required to run under BASIC 1.

### Separators in INPUT

The only separator allowed in the INPUT statement in BASIC 1 is the comma whereas BASIC 2 allows the comma or semicolon to be used interchangeably. A line such as:

10 INPUT"Enter a Number";n%

will cause an error in BASIC 1 and the comma should be used instead. There is no significance in this change; it was, I think, introduced to make BBC BASIC compatible with most other versions of the language which use the semicolon.

### OPENIN and OPENUP

The keyword OPENUP is not used in BASIC 1, OPENIN performs the same function of opening a file for reading and writing. In BASIC 2 both keywords are represented but OPENIN only opens a file for reading. There is never any need to use OPENIN in BASIC 2, however, and it is not actually advisable as a program which contains it will not work, it loaded into a machine with BASIC 1. If you use OPENUP, however, it will be translated into OPENIN if used under BASIC 1 and all will be well. The same token is used for OPENUP in BASIC 2 and OPENIN in BASIC 1 to ensure this compatibility will work either way. If you only want to read from a file in BASIC 2 there is, of course, no disadvantage or significance in having it open for writing as well.

### Number Formatting With @%

The default value of @% is 10 in BASIC 1 and &09A in BASIC 2. This actually stems from some alterations in the binary number to decimal string conversion routines which allow numbers to be printed to 10 significant figures as a default setting. The main thing to remember as far as programming is concerned is that if @% is changed and then has to be reset to the default you must use the correct value for the BASIC in operation. The best way to ensure this is to store the default value of @% in another variable before you alter it and then retrieve it from there when you want to reset. This will then give the same effect in both BASICs.

### OSCLI

This is a keyword only found in BASIC 2, there is no equivalent in the earlier version. Its purpose is to pass a string to the command line interpreter and the string can be a literal or a BASIC variable, or indeed any combination of the two. So, for example, OSCLI("FX 4 "+STR$(N%)) would send FX call 4,N% and N% could contain a user-supplied value. Star commands cannot pass BASIC variables since when the star is encountered, the whole line is passed to the command line interpreter, which knows nothing about BASIC parameters. OSCLI provides a simple means of avoiding this limitation. For compatibility with BASIC 1 you will need to forget about OSCLI but the same effect can be achieved by a direct call to the command line interpreter. You can see an example of this in Listing 6.1, Chapter 6.

### Assembler Differences

There are several additional features in BASIC 2 assembler as compared to the earlier version. EQUB, EQUW, EQUD and EQUS are not available in BASIC 1. The other feature not represented in the earlier assembler is the ability to assemble code at a location other than that at which it is intended to run. In BASIC 2, OPT can be set to 4 or 7 and then code will be assembled at the location contained in O% but such that it will actually expect to run at P%. This is useful for preparing code for loading into ROMs which are to be paged into the space normally occupied by BASIC.

Anything written in the BASIC 1 assembler will be handled quite satisfactorily by the newer version, but compatibility in the other direction will require the additional facilities mentioned above to be ignored.

### General Points

The LOG, LN and all the trigonometrical functions have been rewritten in BASIC 2 so as to make them more accurate so you may find that programs using these return different numerical results depending on the interpreter used. Another point to note is that BASIC 2 has a rather less extravagant use of memory when handling strings. I referred to this in Chapter 4 and showed how to avoid the problem, which arises if the contents of string variables are altered so as to make them longer. The problem is not completely solved in BASIC 2 and the precautions outlined in Chapter 4 should still be exercised.

### Z80 Basic

The version of BBC BASIC supplied for the Z80 second processor is completely recoded, of course, but represents a quite remarkable achievement on the part of its author in that it is almost totally compatible with the 6502 version. Apart from the problem with random access files noted in Chapter 6, correctly written programs should be transferrable from BASIC 1 or BASIC 2 to the Z80. Compatibility in the reverse direction is only limited by the same problems of compatibility between BASICs 1 and 2 already noted; the Z80 BASIC being effectively BASIC 2.

You may fall foul of the Z80 BASIC if you have slipped into bad programming habits as regards leaving loops. The 6502 version of BBC BASIC will tolerate, to an extent, jumps out of FOR…NEXT and REPEAT…UNTIL loops, without proper termination. So you can get away with a piece of code such as:

10 FOR i%=1 TO 10

20 IF i%=z% THEN 40

30 NEXT

40 REST OF PROGRAM

in 6502 BASIC, although it will probably give you problems eventually. In Z80 BASIC it will certainly be a disaster and should be avoided by changing the structure completely or by rewriting line 20 as:

20 IF i%=z% THEN i%=100

so that the loop is correctly terminated.

The reason for this behaviour in Z80 BASIC is that it only uses one stack for all purposes and so if you do not leave a loop correctly the loop return address will be left on the stack to create havoc later on. For the same reason you cannot change HIMEM inside any type of structure which requires a return address in Z80 BASIC, this means within loops, subroutines, procedures and functions, since the stack is corrupted when HIMEM alters and so the return address will be lost. It is not likely that you will want to do this, however, particularly since changing mode does not affect HIMEM when working across the Tube.

The most subtle trap in Z80 BASIC is concerned with local variables. These are also stored on the stack and so you cannot use a sequence such as:

DEFPROCfatal

FOR i%=1 TO 50

LOCAL array(i%)

NEXT

since the return address for the loop will get totally lost as the values of the array elements are pushed onto the stack. The solution to this is to avoid the FOR…NEXT structure and construct the loop from IF statements and GOTOs which do not require the stack. This is, in all conscience, not a problem that you are likely to encounter since the LOCAL declaration of an array is not a widely-used technique in BBC BASIC but it can be a very confusing thing if you do try it in Z80 BASIC since the resultant crash is often difficult to understand.

Z80 BASIC has no limit on the depth of nesting of loops and subroutines except that you may run out of stack space, in which case the "No Room" error will be signalled. The equivalent errors in 6502 BASIC are "Too Many FORs", "Too Many REPEATs" or "Too Many GOSUBs". If you get a surprising "No Room" error in Z80 BASIC then look to see if you have not closed a loop or subroutine correctly.

The only incompatibility problem that this is likely to cause is if a program written on the Z80 with a very high degree of nesting is transferred to the 6502 where the depth of nesting allowed is now inadequate. Unless you go in for really convoluted structures you will probably have found that running out of nesting levels is a fairly rare happening on the 6502 so it is unlikely that you will produce any code on the Z80 which will give rise to this problem when transferred to the other BASIC.

Although machine code written for the 6502 will obviously not run on the Z80 you still need to use CALL and USR to access operating system routines. The way that these calls behave, in the Z80, has been modified for access to legal operating system entry points and so any such calls, if they only use the documented routines, should be handled correctly when transferred between the two processors.

### Compatibility with the 6502 Second Processor

The addition of a 6502 second processor to the BBC not only expands the memory but also increases the speed of the machine, quite dramatically as far as some operations are concerned. One consequence of this is that any software timing methods that are employed will not work properly across the Tube. For example, if you have used an empty FOR…NEXT loop to create a delay in a program, then the delay time will be halved with the program runs in the second processor. The correct approach is to use the TIME facility in a structure such as:

TIME=0:REPEAT UNTIL TIME=100

which will cause a one second delay irrespective of whether the software is running in the main machine or across the Tube.

Acorn's documentation is full of fire warnings that you must not access memory directly in software which is to be Tube-compatible. The reality is not quite so simple, nor so restrictive, as the Acorn injunctions would suggest. To understand exactly what you can get away with it is useful to have some grasp of the way in which the second processor interacts with the main machine. The latter is often referred to, in this context, as the I/O processor.

When the second processor is present, your program is actually loaded and running in its memory. The I/O processor, the original BBC, is only concerned with handling input and output of data. This means that any use of the indirection operators, !, ? and $, will now access the second processor's memory. If your intention in using these operators is to read or write to some area of memory for a specific purpose, such as a direct poke to the screen or an inspection of a port, then the effect you want will not be achieved when you move the software across the Tube. On the other hand, if you are simply using a memory location for storing data, or retrieving that data, then the current processor memory will be the one that you want and so the indirection operators will be quite satisfactory. If you do need to read from, or write to, the I/O processor memory, irrespective of which processor the program is running in, then OSWORD 5 and 6 will do the necessary operations. The use of these calls is fully described in the *BBC User Guide*.

Machine code written for the I/O processor will run equally well across the Tube provided that you make sure that it can be located in a safe place. In the second processor, PAGE is set to &800, since the filing system workspace is all in the I/O processor, and so if you have machine code that requires to load in one of the favourite, safe, locations in the I/O processor, e.g. &A00, the serial buffer, or &C00, the user-defined character area, then it will be loaded on top of the program when transferred across the Tube – with fatal results.

There are two solutions to this problem. The simplest one is to reset PAGE to the value normally found in the I/O processor, &1900 with the DFS, so that all the memory between this value and &800 in the second processor can be safely used. If you include the re-definition of PAGE in a !BOOT file on disc then the software will run with or without the second processor since PAGE will not be altered from its default value if the Tube is not present. The second solution is to avoid any absolute storage locations for machine code and always to use a DIM statement to reserve space within the area used by BASIC. This latter is probably the best solution.

If the software which you are writing cannot be made Tube-compatible, as is the case with some graphics manipulations which require the speed associated with direct pokes to the screen memory, then it is helpful to check for the presence of the Tube and issue a request to switch off the second processor if it is in use. OSBYTE call &EA will do this and Listing 7.1 shows how it can be implemented.

**Listing 7.1**

10 MODE 6

20 IF FNtube\_present PRINT"TUBE ON" ELSE PRINT "NO TUBE"

30 END

40

50 DEF FNtube\_present

60 A%=&EA:X%=0:Y%=&FF

70 !&70=USR(&FFF4)

80 IF ?&71=0 THEN =FALSE ELSE =TRUE

### Compatibility with the Z80 Second Processor

There is really nothing to say here that has not been covered in previous sections. The major concerns are those involved in ensuring compatibility of software between the 6502 and Z80 BASICs and copying with the filing system differences and these topics have already been covered. The question of the use of indirection operators, direct peeks and pokes if you like, and ensuring that you access the correct memory, is exactly the same with the Z80 as with the 6502 second processor and machine code or assembler source code compatibility does not arise, it is impossible since the two processors are completely different animals.

### Conclusions

Although this chapter has been concerned with compatibility I am going to end it with a recommendation not to try to push the idea too far. If you try to write one piece of code that will serve in all systems then you will almost certainly have to forego some of the advantages to be gained from running on the more sophisticated of the systems intended to be used. If you tailor down for the Electron then you may well not be exploiting possibilities available on the BBC, for example. It is frequently preferable to produce separate versions of a program for the differing machines. There are, I think, two important areas where compatibility should always be striven for, however, and these are that all BBC software should be capable of running under BASIC 1, since there are many machines with this chip still in use, and, unless there is a very good reason why this is not possible, it should also run in the 6502 second processor. Those who have invested in this valuable enhancement to the performance of their machines have a right to expect to get the benefits.

# Chapter 8.

# The Art of Concealment

Users need, of course, to be able to run your software but, on the whole, there is no reason why they should have to see what is going on in the program itself. Indeed, on occasions it may be essential to prevent access to the program, since its examination may lead to a user being able to breach the security of the data or simply to crib the answer to the problems posed in an educational program. In addition to considerations of this type, there is the whole problem of protection of software from piracy, which includes not only prevention of access to the program for the purpose of stealing ideas, but techniques to make the copying of tapes and discs difficult. I shall not, for obvious reasons, be dealing with the many sophisticated methods of commercial software protection, but there are a few simple, and fairly widely known techniques which I will describe here largely for the benefit of those in the business and educational sectors who want to make access to information obtainable by listing programs more difficult.

### Preventing 'Breaking Into' Programs

Once a program is running on the BBC or the Electron it is possible to prevent it from being stopped in such a way that it can be listed. The obvious ways of stopping a program are by the use of the ESCAPE and BREAK keys. If you are trapping the ESCAPE key with an ON ERROR statement then this will be effectively dealt with; alternatively the ESCAPE key can be switched off completely by the command \*FX200,1. \*FX200,0 will turn it on again. The action of the ESCAPE key is controlled by the least significant bit sent by \*FX200, but the next bit controls the action taken when BREAK is pressed. The normal situation is that BREAK does not clear the memory and so a program can be recovered by OLD or investigated by direct inspection of the memory with a suitable piece of software. If the second bit in \*FX200 is set, however, the memory is cleared on BREAK and so, if this condition is set and BREAK is the only way of stopping the program, the user will not be able to investigate it once it has started to run. To summarise the effects of the possible \*FX200 calls:

\*FX200,0 ESCAPE active and BREAK does not clear memory (Default)

\*FX200,1 BREAK does not clear memory but ESCAPE switched off

\*FX200,2 ESCAPE active and BREAK clears memory

\*FX200,3 ESCAPE switched off and BREAK clears memory

Setting BREAK to clear the memory gives no protection against the program being LOADed without actually being RUN, and then listed, but listing can, itself, be made difficult as we shall see in the next section.

BREAK can be programmed as function key 10 and it is common for it to be set up to produce the string OLD|MRUN|M so that BREAK itself will normally just restart the program from the beginning. CTRL/BREAK, however, will still work as normal and so programming key 10 is not a complete solution to preventing the broken being broken into.

From a number of points of view it would be useful to be able to completely neutralize the effect of the BREAK key. This would not only prevent the breaking of programs but would also protect users from loss of data as a result of clumsy fingers. The recovery techniques involving reprogramming BREAK is of no help here, of course, since all the variables are cleared when the program is RUN.

There have been several methods suggested for trapping the BREAK key in software, in a manner similar to the use of the ON ERROR statement in BASIC. These all involve resetting the BREAK vector and taking various steps to re-enter the program safely but I cannot recommend any of these methods with confidence. The BREAK keys on the BBC and Electron are hardwired directly to the processor reset line and so they must always interrupt the processor instantaneously when pressed. Any attempt at recovery from this situation is fraught with hazard since the processor can be doing absolutely anything when reset and the system really expects to restart properly. I have tried several methods of trapping BREAK, both published ones, and of my own design, and my conclusion is that none are satisfactory. Corruption of discs is a common result of such attempts, in my experience, and this can be far more disastrous than the loss of data in memory.

The only real solution to trapping the BREAK key is a hardware modification enabling it to be switched off completely. There are several kits on the market, notably one manufactured by ATPL (See Appendix 2), allowing this facility to be fitted to the BBC but do not attempt the job yourself unless you are very confident of your ability.

Whatever you do to prevent the program from being stopped, remember not to leave it accessible once the user has ended it legitimately. If you insist that BREAK is pressed to leave the program and make this clear the memory, then the object will be achieved. A straightforward use of END, however, will leave the program in memory and perhaps listable, so some additional steps are required. The statement !PAGE=0 included just before the END will corrupt the first few bytes of the program and produce a Bad Program message on an attempt to list.

### Prevention of Listing

Assuming that someone has managed to load your program into memory it is possible to make it awkward to list by including control codes at various points which will switch off the screen and the printer. Do not forget the printer as it can work without the screen being active; VDU3 will turn off the printer and VDU21 switches off the screen. The codes must be sent in this order, by the way, or VDU21 will prevent the VDU3 being acted upon.

The trick is to include the sequence VDU3,21 in a REM statement at the beginning of the program so that when LIST is attempting the control codes in the REM will be executed and the screen and printer switched off so that you do not actually see anything. To confuse the issue further, you can include a sequence of deletes so that REM itself and its line number are not visible. The necessary control codes cannot be entered in the normal way; you will have to poke them into the program directly as illustrated in the following example:

Enter, as the first line of any program:

1REM\*\*\*\*\*\*\*\*

Notice the absence of a space after the line number; this is important. Now enter, in direct mode, the commands:

!(PAGE+5)=&7f7f7f7f

?(PAGE+9)=3

?(PAGE+10)=21

An attempt to list the program will now fail, whether the printer or screen is in use. The printer may carry out a lot of linefeeds but nothing will be printed out. What has actually happened is that 1REM was printed and then deleted by the four deletes, character &7F. The printer and screen are then shut off by VDU3,21. An attempt to list from a line other than 1 will still be successful or someone may simply delete the REM with the protection in it, but it you renumber the program to run from some unexpected line number in peculiar increments, say RENUMBER123,3 it will keep the casual snooper at bay.

If you are writing educational programs or some types of game you will frequently find that you have the answers to problems contained within the program in DATA statements and it is useful to be able to hide these specifically. This can be done in a similar manner to the method used to prevent listing, by including the printer-disable and screen-off characters in the data statements.

#### Listing 8.1

10 RESTORE

20 FOR I%=1 TO 10

30 REPEAT

40 READ A$

50 UNTIL A$<>"\*" AND A$<>CHR$(21) AND A$<>CHR$(3)

60 PRINT A$;

70 NEXT

80 DATA\*,\*,F,R,E,D

90 DATA\*,\*,A,L,A,N,R,O,W,L,E,Y

100 END

Listing 8.1 shows the way in which a program must be set up in order to make the required insertions. The DATA statements at lines 80 and 90 each have two stars as their first entries. The format of the beginning of these lines is important. Up to the comma after the second star they must be typed exactly as shown. Do not include any spaces. The READ in line 50 is written so that it will over the stars and either of the control codes with which we are going to replace them. This is necessary since the first two data items are not really data items at all; they are there for protection. If the data statement is to contain numerical information then you will have to arrange to read the numbers as strings initially and then convert them with VAL.

#### Listing 8.2

10 PRINT"CONCEAL-C or REVEAL-R ? ";

20 REPEAT

30 a$=GET$:UNTIL INSTR("CR",a$)

40 PRINT;a$;" \*PLEASE WAIT\*"

50 IF a$="C" THEN W%=21:V%=3 ELSE W%=ASC"\*":V%=W%

60 FOR I%=&1900 TO &5FFF

70 IF ?I%=&DC PROCcheck

80 NEXT

90 END

100

110 DEFPROCcheck

120 FOR J%=1 TO 10

130 C%=I%?J%:IF C%>31 AND C%<127 THEN PRINT CHR$(C%); ELSE PRINT"\*";

140 NEXT

150 IF a$="C" PRINT'"HIDE THIS ? "; ELSE PRINT'"REVEAL THIS ? ";

160 REPEAT \*FX15,1

170 b$=GET$:UNTIL INSTR("YN",b$):PRINT

180 IF b$="Y" ?(I%+1)=V%:?(I%+3)=W%

190 PRINT"\*PLEASE WAIT\*"

200 ENDPROC

Listing 8.2 is a program which will read through any BASIC code in memory from &1900 to &5FFF and detects the token for DATA, with is &DC. This is done at line 70. It then prints out the next ten characters, lines 120 to 140, so that you can identify the data statements to be protected. Control characters are not printed since this could lead to problems, one of them might be VDU21 for example. You can then exercise the option of replacing the starts at the start of the DATA with the two control codes VDU3 and VDU21. An attempt to list the program will then fail at the first protected DATA statement as will any attempt to list the DATA line in isolation since the control codes will switch off screen and printer. Listing 8.2 also allows you the option of restoring the program to full readability if necessary.

In order to use Listing 8.2 set the BBC in mode 7 and then load the program to be protected at PAGE=&1900. Reset PAGE to &6000 and load Listing 8.2. Type RUN and when the program finishes change PAGE back to &1900. You can now save the program which will have its data statements protected.

As it stands Listing 8.2 and the instructions given above refer to a BBC Micro with DFS and set in mode 7. For use in a BBC with no DFS the &1900 in line 60 and in the instructions should be changed to &E00 and if running in the 6502 second processor to &800. For use in the Electron you should ensure that mode 6 is selected and line 60 should be changed to:

60 FOR I%=&E00 TO &4FFF

for the machine without discs and to:

60 FOR I%=&1D00 TO &4FFF

for the disc machine, to allow for the different PAGE values. In either case Listing 8.2 should be loaded at PAGE=&5000 on the Electron.

### Hiding a Disc Catalogue

As an additional contribution to confusion it is sometimes useful to prevent a disc from being catalogued. With the Acorn DFS this can be achieved in a similar manner to prevention of program listing by including control codes in the disc title. This is the first thing to be printed when the disc is \*CATed and so if the first two characters are 3 and 21 the list devices will both be turned off and no visible catalogue will appear. As with the REM statements in programs you cannot write control characters directly into the title. If you have a disc editor, such as *Disc Doctor* by Computer Concepts, then you can modify the title directly by changing the first two bytes of sector zero on track zero to 3 and 31. Alternatively use Listing 8.3 which loads the first sector of the disc into memory, makes the necessary modifications to the title, and then writes the modified sector back to the disc. Listing 8.3 also allows the damage to be reversed by replacing the control codes with spaces.

#### Listing 8.3

10 MODE 7

20 PROCinit

30 REPEAT

40 PRINTTAB(0,10)"Which Drive ?-PRESS 'E' TO EXIT ";:\*FX15,1

50 a$=GET$

60 UNTIL INSTR("0123E",a$)

70 IF a$="E" THEN CLS:END

80 D%=VALa$

90 PROCdisc("read")

100 REPEAT \*FX15,1

110 CLS:PRINTTAB(0,10)"REVEAL-R or CONCEAL-C? ";

120 a$=GET$

130 UNTIL INSTR("RC",a$)

140 IF a$="R" THEN !buff%=&20202020 ELSE !buff%=&1503

150 PROCdisc("write")

160 RUN

170

180 DEFPROCinit

190 DIM buff% 255,bl% 12

200 A%=&7F:X%=bl%MOD256:Y%=bl%DIV256

210 bl%!1=buff%:bl%?5=3:bl%?7=0:bl%?8=0:bl%?9=&21

220 ENDPROC

230

240 DEFPROCdisc(n$)

250 LOCALn%:IFn$="read" n%=&53 ELSE n%=&4B

260 bl%?0=D%

270 bl%?10=0

280 bl%?6=n%

290 CALL &FFF1

300 ENDPROC

If you cannot following Listing 8.3, do not worry. I offer it as a convenient utility rather than an example of programming. A word of warning, however, do be very careful about how you enter it and, *under no circumstances try it out for the first time on a disc with data that you cannot afford to lose.* A simple slip in the entry of this program could result in the complete corruption of a disc. If this happens the disc should recover alright if re-formatted but the data will be lost, of course.

Catalogues cannot be hidden in this way on the ADFS since control codes are filtered out of the title when it is output to screen or printer.

# Chapter 9.

# Documentation of Programs

In this chapter I want to deal with two types of program documentation, that which you will require for yourself and that which will need to be supplied to users of the program in the form of a manual. The complexity of the instructions required by users will depend on how the software has been written. Very friendly menu-driven software can be largely self-documenting: the user will be led through the various stages by the menus and prompts. A command-driven program, on the other hand, will probably need quite a large amount of documentation explaining the syntax and the various commands.

The necessity for some kind of user documentation will be recognized by most people but the requirement for documentation for the programmer is perhaps less obvious. The purpose of this second type of documentation is to provide information which can be referred to whenever the program needs to be modified or updated. It should explain the structure of the program, the purposes of the various routines and also what is held by the main variables. You might well feel that there is hardly any need to record this information formally. Over the days and weeks during which you are developing a piece of software the way it works and the uses to which the variables are put becomes a part of your existence and you may feel that there is no chance of your forgetting, but, if you have to return to the program even a few weeks later you will find that this is far from the truth. Additionally, if the program is being written for your business or school it is entirely possible that, in the future, you will depart to pastures new and someone else may want to modify the software. It is, I can assure you, an almost impossible task to modify and maintain undocumented software written by someone else. Their way of thought and programming style will almost certainly be different to your own and you are more than likely to find the resulting program most abstruse!

### Documentation for Users

However you construct the user guide to your program it will be criticized by someone, often quite vehemently. It is an interesting exercise to read the reviews of the same piece of software carried out by different magazines. Agreement about what is a good or bad program is fairly common, but the diversity of comments on the documentation is often quite astonishing. It follows that I cannot really give you a set of rules for writing a good user guide but rather offer a few pointers and topics for consideration.

The main reasons for the differing views on program documentation stem from the variety of abilities amongst users themselves. You will have to communicate with the total novice, the experienced operator and also the expert, who may well be a better software author than yourself. You will always need to err on the side of simplicity, an instruction that the program is to be started "by auto-booting the disc" is hardly helpful to someone who has just collected his first computer from the shop, whereas the old hand's first reaction to any disc-based program will be to shove the disc into drive zero, press SHIFT-BREAK and away. I like to include at the start of my manuals a section called 'Getting Started' which gives a simple set of instructions as to how to get the program up and running. If your software is well-written, much of it will be self-explanatory anyway so the best way for a user to get to grips with it will be to run it and "have a play".

I think that it is useful to provide a section in the manual which deals with the overall picture of what the software is intended to do and the various facilities provided. You might even consider including some information on how the software works. This will be of little significance to the novice user but can be very helpful to the experienced, particularly if they are programmers, and help them to use the software in a more constructive manner.

A detailed description of the commands or menu options will be required at some point since there is a limit to how much you can explain on the menus or in help facilities. As an example let us consider the fictitious database discussed in Chapter 1 where we offered a sort option on the entries. The menu can only label this but the user will need to know all kinds of details. For example, in an alphabetical sort are upper and lower letters distinguished? What will be the effect of numeric characters in the text on the sorting order? How will the space be regarded; is it ignored or treated as the lowest order character as in the ASCII list? This is all information that the user will be able to do without initially but he is bound to ask this kind of question eventually, and if the answers are not available he will have to resort to experiment to find them out. This should never be necessary but is frequently the only way out even with expensive software packages, I am very much afraid.

You must include something about the error-trapping facilities in your software. If you have done your job properly then the only errors that will normally arise are from the filing and operating systems. The user must be warned what, if anything, to do if an error message is received. Normally you will only have to point out the need to wait for things to be restored to a safe point in the program and to counsel against panic measures like pressing BREAK. There will also need to be full explanations of any error messages that are likely to occur in the normal course of events and the action needed to rectify the situation. For example, what if the user loads the wrong disc and the program fails to find a file that it needs? What message will the user receive?

As you have probably realized, the advice that I am giving in this section is much the same as the thread running throughout this this whole book. Be helpful, be informative and, above all try to anticipate difficulties that the user might encounter and provide the solutions in advance.

### Documentation for Programmers

The first thing in your own documentation for the program should be an explanation of the purpose of the software, its scope and limitations and a breakdown of the program structure in such a way as to give an understanding of how it works. Explain exactly what the main control structures are intended to achieve and how and under what circumstances the subroutines are called. If you are a devotee of flowcharts, and in my experience few people other than formally taught programmers are, then they should appear here also. It is also useful, at this point, to have an indication of the minimum hardware configuration required for the software and a note of its compatibility with other machines and systems.

BBC BASIC can, to an extent, be self-documenting. The freedom to use long variable names and to give subroutines names, i.e. procedures and functions, means that programs can be highly readable. If you try to take this too far, however, you will find that memory is eaten up at a great rate by the long names. It is generally useful to employ names for variables that represent reasonable abbreviations which act as an aid to program development for yourself, but to keep a separate list detailing, in full, the exact purpose of all the major variables including any arrays. Calling a variable debtors\_surname$ will certainly make for readability but is a bit extravagant in memory and dbsn$ may well be equally mnemonic when you are developing the software.

Similarly a list of procedure and function names together with a brief description of the jobs that they do will be useful. I also like to record details of any parameters that are passed to such subroutines and the purpose of each one.

REM statements can, of course, be included in the body of the program to explain the function of its various parts and, indeed, some programmers prefer to include all the documentation that we have detailed so far as a series of REMs at the start of the program. This latter practice is clearly only feasible with relatively small programs on micros but is a standard technique on machines where memory is not at such a premium and it ensures that the documentation and the program go together always. One option adopted by some people is to keep two versions of a program; a fully commented copy and a copy from which all comments are removed. The commented version may not actually be runnable, being too big to accommodate the code, variables, screen memory and any work space, and the stripped down version is what is actually used. To facilitate this kind of documentation there are several utilities on the market which will strip all REM statements from BASIC code and even any redundant spaces. Examples are the \*PACK facility in the Beebug Toolkit ROM and \*SQUASH in Computer Concepts' *Caretaker*.

Both of these will also list the variables, functions and procedures used in a program which can be a great help in preparing your documentation. The output from these utilities can even be spooled into a disc file which can then be loaded directly into a word processor such as View or Wordwise and serve as the nucleus for the documentation preparation.

If your software uses data files on disc or tape it is particularly important that you record the format of these for future reference. It is often very difficult, particularly in the case or random access files, to deduce the file format from the program and a clear explanation in the documentation is of inestimable value here.

As a conclusion to this book we have included a complete software package. It consists of a listing of a program designed to produce a neat function key strip for the BBC micro and the user and programmer's documentation. You will find this in Appendix 1. We hope that you will find the program useful and offer it as an example of how to prepare and present a program.

We hope that you will have found something in this book to inform you and something to make you think, but above all we would hope that you will be stimulated to look critically at your own programming and be moved to try for the highest possible standards. Good software is expensive to buy but costs only time and effort to write. The microcomputer will, no doubt, change out of all recognition as time goes on but there will always be the need for good software written by careful and creative individuals. Good luck with your efforts but always remember the first law:

THERE IS ALWAYS ONE MORE BUG

and you will never be complacent. The second law is much more comforting if you want to be kept in employment writing software and it states:

ALL PROGRAMS CAN BE IMPROVED.

# Appendix 1.

# Function Key Strip Maker

The main object of this section is to provide you with an example of a complete and fully documented program although I would also like to think that you will find the software useful. The documentation is presented in two parts, that which would be supplied to the user and that which would be required by the programmer as a full record of the software for maintenance purposes. I have assumed, in the user instructions, that the software will be mounted on a disc which has been designed to auto-boot.

### Instructions and User Notes

This program will enable you to prepare neat, professional-looking function key strips for the BBC Microcomputer. Each strip can have up to three lines of text, each line having up to twelve characters. The lines will be centred automatically if less than 12 characters in length. Strips can be saved to disc and retrieved at a later date for printing or editing.

A BBC Microcomputer with at least one disc drive is required and a dot matrix printer which is capable of producing condensed text at a spacing of 8 lines to the inch. As supplied the program is configured for the Epson FX80 and RX80 printers but should run on any compatible printer which selects condensed mode with CHR$(15) an sets a line spacing of 1/8 inch by ESC"O". The printer is assumed to be connected to the parallel printer port and to be set to carry out an automatic line feed for each carriage return.

#### Getting Started

If you want to save strips to disc you will need a formatted disc at hand. To start the program press CTRL-BREAK to reset your machine and then load the disc in to drive 0. Press BREAK and SHIFT together and then release the BREAK key whilst holding down SHIFT. Release SHIFT. The program will load and run automatically and you will be asked whether you want to load a strip from disc, option A, or to start a new strip, option B. Select the second option and you are ready to enter the text to appear over each key. A maximum of twelve characters is allowed on each line and there are three lines per key. To leave a line blank just press RETURN without making an entry. You will not be allowed to enter a line which is too long, an attempt to do so will receive a warning bleep and the keyboard will not respond. The delete key can be used to remove any wrongly typed characters. When you have finished typing each line press RETURN, and the next line, or next key, as appropriate will be prompted for. Do not worry about trying to centre lines relative to one another. This is done automatically when the strip is printed.

The program will jump, automatically, to the main menu when you press RETURN after entering the third line on the label for key 9. If you want to stop before this, press ESCAPE to return to the menu. Any keys not labelled will be supplied with blanks automatically.

#### Main Menu Options

The main menu is displayed in a window down the right hand side of the screen and the currently held key strip is seen down the left. To select a menu option you must press the indicated key and then RETURN. If you make a wrong selection press DELETE rather than RETURN and make the correction.

#### E – EDIT STRIP

On selecting this option you will be prompted for the key to be edited, 0 to 9, and then for the line, 1 to 3, within that key. Only valid numbers will be accepted as input. You can make corrections by pressing DELETE and each input is terminated by pressing RETURN.

The new version of the line should be typed when the prompt appears and RETURN pressed, when the key strip representation on the left of the screen will be updated and you will see the effect of the change. Remember that a maximum of 12 characters per line is permitted.

#### P – PRINT STRIP

This option will produce the finished hard copy of the strip at the printer. If you select it and there is a problem with the printer, such as it not being switched on, you will be notified. If this happens you will need to press the ESCAPE key and put things right before trying again.

#### S – SAVE TO FILE

On selecting this option you will be asked to load the disc which is to be used to store the strip and to enter the drive number. If you only have one disc drive then you need make no input here other than pressing the RETURN key.

The catalogue of the disc which you are using will be displayed and you will be asked to enter a name for the file which is to contain the data for the strip. The name can be a maximum of 7 characters in length and each character must be a capital letter. If either of these rules is violated the filename will be rejected and you will have to make another input.

Once you have entered your filename a check will be made to see if a file of that name is already present on the disc. If this is the case then you will be given the chance to change your mind and think of an alternative name. If your new file is merely intended as a replacement for an existing one of the same name, perhaps you have been editing a strip to correct a mistake, then you will presumably be quite happy to proceed and overwrite the old file with the new. On the other hand the coincidence of filenames may simply be fortuitous and so you will presumably not want to go on. In this case you will be asked for a re-input of the filename.

#### R – RE-RUN PROGRAM

This option simply starts the program from the beginning again. Since any data in memory will be destroyed on the re-run you are asked to confirm this selection before it is acted upon. Any strips not saved to disc will be lost if this option is used.

The main function of this re-run option is to allow you to start a new strip or to load a strip from the discs.

#### F – END PROGRAM

Again, since this option is destructive of data, you are asked to confirm. Make sure that you have saved any strips that you wish to keep.

The main menu can generally be restored by pressing the ESCAPE key so if you get yourself lost or make a wrong selection from the menu this will be your best route to safety.

#### Loading Strips for Editing or Printing

The option to do this is provided at the start of the program or if the re-run option is selected from the menu. You will be asked to load the disc with the data and to enter a drive number. If you have only one drive then press RETURN without making an entry.

The disc catalogue will be displayed and you will be asked to enter the name of the file to be loaded. The filename must be no more than seven characters in length and can have only capital letters, names breaking either rule will be rejected. The filename which you enter must correspond to a file actually present on the disc. If the program cannot find the file that you asked for, it will tell you so and you must try again. The most likely reason for this is that you mistyped the filename or have the wrong disc.

Once a valid file has been loaded then the main menu will appear and you will be able to set about your editing or printing as required.

#### Errors

The only errors that should occur are those arising from the disc filing system and these will only happen if you have a fault on the disc itself or in the drives or the disc becomes full. If an error does happen then it will be reported and, after a brief pause, execution of the program will begin again from a safe point. Provided you wait and do not take panic measures like pressing BREAK your data should be intact. The most likely error that you will see is Catalogue Full, which indicates that you have tried to write more than 31 files to a disc. Change to a new disc with space left on it and try the save again.

One problem that could arise is if you enter a disc number which does not correspond to a drive in your hardware or if you no disc in the drive you select. If you make this mistake, the system will seize up. Try pressing ESCAPE to retrieve the situation; if it works any data should be preserved. If pressing ESCAPE fails then press BREAK. This will, unfortunately, destroy all the data in memory but the program will be re-activated.

### Program Documentation for Maintenance

Key strip printed for disc-based BBC Micro and Epson FX80 and RX80 printers. Should work on MX80, CP80 (Shinwa) and any other Epson compatible but not actually tried on these. Printer control codes used are CHR$(15) – Condensed text and ESC "O" – Line spacing 1/8th inch. Default is Centronics port and no line feeds from host.

Strip has three lines per key, 12 characters per line. Auto-centred.

Compatible BASIC 1 and 2 with 6502 second processor running ordinary BASIC. Not HI-BASIC or Z80 compatible due to direct call into BASIC in FNinline.

#### Initialisation System

Program BREAK, KEY 10 to re-run.

Initial error trapping to re-run and pick up attempt to load a non-existent file.

Set mode 4 and make logical colour 0 (Black) actual 4 (Blue).

Check BASIC 1 or 2 by peek to &8015.

Assemble code for line input. Calls BASIC line input routine; simply entry into OSWORD 0.

#### Initialisation Variables

m$ - Message

key$(9,2) – Holds strings for key labels. First index is key number and second is line. Pad out to maximum of 12 characters at initialization.

Code% 20 – Machine code location for line input.

Buffer% - Buffer space for line input routine and PROCoscli.

#### Main Loop

Screen in two windows. Vertical split. Defined in PROCwindow. Menu right screen; strip display on left.

Error trap standard REPORT unless ESCAPE then return to menu top. Selection of option by letter then CR, upper/lower case acceptable (See FNget). End on F and re-run on R.

#### Procedures and Functions

DEF PROCdisplay

No parameters.

Local variables I%, J% - loop counters.

Prints strip held in global array key$ in screen window.

DEF FNcentre(a$)

Parameter is string to be centred. Must be maximum of 12 characters or the code will not cope.

Local variables: i% - loop counter; pad$ - Holds leading and trailing spaces to be added; temp$ - Holds string being built up.

Centres the parameter string on 12 character line by adding leading and trailing spaces. Returns correctly padded string. Odd numbers of characters are left adjusted.

DEF FNget(allowed$)

Parameter string of characters to be acceptable key presses. Must all be high case if letters.

Local variable in$ - holds keys pressed.

Reads keyboard buffer and returns key pressed as string if the character is contained in allowed$. Lower case key presses are recognized and converted to upper case by AND operation with &DF.

DEF FNprinteron

No parameters.

No local variables.

Sends three delete codes to printer and examines output buffer with ADVAL(4). If buffer empty, i.e. printer is valid, returns TRUE otherwise returns FALSE.

DEFPROCwindow(n%)

Parameter is 1 or 2 for window number. 1 is left half of screen for data display and 2 is right half for menu.

Select window.

DEF FNsure

No parameters.

Local variable A$ - holds key press. Used to give second chance on menu selections with serious consequences. Issue prompt and read keyboard buffer, return TRUE if Y pressed else return FALSE.

DEF FNinline(x%,y%)

Parameters X and Y TABs of screen position for input.

Local variable in$ - temporary store for input.

Set cursor at required input position on screen then call input routine in BASIC ROM. Extract input from buffer and return it as string.

DEF FNstrip(a$)

Parameter is string to be processed.

No local variables.

Strip leading and trailing spaces from string and return resulting processed string.

DEFPROCnewstrip

No parameters.

Local variables i%,j% - loop counters.

Accept key label input. Exit on ESCAPE key error trap or on completion of line 3 of key 9.

DEFPROCedit

No parameters.

No local variables.

Accept input of label for key key% and line line% and read into appropriate elements of array key$.

DEFPROCload

No parameters.

Local variables i%, j% - loop counters; fnm$ - user entered filename.

Select drive.

Accept user filename and attempt load. Error &DE traps non-existent file.

DEFPROCdrive

No parameters.

Local variable d$ - user drive number.

Accept user input of drive number and validate. Validation condition, single character and value 0 to 3 inclusive. If valid pass to OSCLI and change drive. Non-existent drives accepted.

Display disc catalogue.

DEFPROCsave

No parameters.

Local variables i%, j% - loop counters; fnm$ - user entered filename; a$ - key press.

Select drive and then prompt for user entered filename and validate. If valid check for existence of file. If file exists offer escape chance or continue with overwrite and carry out save.

DEFPROCerr

Main error trap. Close all files. If ESCAPE pressed go to main menu with no output otherwise REPORT and wait for 10 seconds or keypress.

DEFPROCtype

No parameters.

Local variables i%, j% - loop counters.

Check printer on etc. and issue error message if problem. Return on key press. Reset error trap to reactivate screen before returning to menu.

Set output to printer only.

Set condensed mode and 1/8th inch line spacing. Epson codes as default.

Output strip to printer and then restore output to screen only.

DEFPROCoscli(m$)

Parameter string to be passed to command line interpreter.

No local variables.

Load string plus CR into buffer and call command line interpreter with X and Y registers pointing to the command, Y% MSB.

This routine is required for compatibility with BASIC 1.

DEF FNexists(f$)

Parameter f$ - filename to be checked.

No local variables.

Checks for existence of file by opening channel. Returns FALSE if file does not exist, i.e. channel zero set, otherwise returns TRUE.

DEF FNok(f$)

Parameter filename to be checked.

Local variables: i% - loop counter; char$ - temporary storage for characters to be examined.

Validate filename and return TRUE or FALSE.

Validation conditions are length must be 1 to 7 characters inclusive and all characters upper case letters.

#### Machine Code

This is used to set up the line input for the labels. Calls the routine in the BASIC ROM which accepts buffered input with OSWORD 0. Locations &37 and &38 point to buffer, &38 MSB, so these are stamped on. BASIC routine is entered with maximum line length in the accumulator, 12 (decimal ) in this case.

#### File Format

Direct save of array key$ with first index priority, i.e. save order key$(0,0), key$(0,1), key$(0,2), key$(1,0)… etc.

10 REM\*KEY STRIP MAKER\*

20 \*KEY10 OLD|MRUN|M

30 ON ERROR CLOSE #0:IF ERR=17 THEN RUN ELSE IF ERR=&DE PRINT'

" No such file on this disc-Press a Key":E$=GET$:RUN ELSE CLS:REP

ORT:E$=INKEY$(1000):RUN

40 MODE 4

50 VDU 19,0,4,0,0,0

60 DIM key$(9,2),code% 20,buffer% 20

70 m$="FILE NAMES MUST BE NO LONGER THAN SEVEN CHARACTERS AND

ALL MUST BE CAPITAL LETTS"

80 FOR I%=0 TO 9:FOR J%=0 TO 2

90 key$(I%,J%)=FNcentre(key$(I%,J%))

100 NEXT:NEXT

110 IF ?&8015=&32 input%=&BC0F ELSE input%=&BC2A:REM\*BASIC 1 OR

2 ?\*

120 P%=code%:REM\*CODE FOR LINE INPUT\*

130 [

140 OPT 2

150 LDA #buffer% MOD 256:STA &37:LDA #buffer% DIV 256:STA &38:L

DA #12:JSR input%:RTS

160 ]

170 CLS

180 PRINTTAB(3)"FUNCTION KEY STRIP MAKER FOR BBC"''TAB(5)"by Al

an & Sue Rowley (C) 1985"

190 PRINTTAB(0,10)"Do you want to:-"''TAB(5)"A. Load a strip fr

om a file"''"or"''TAB(5)"B. Start a new strip ? ";

200 IF FNget("AB")="A" PROCload ELSE PROCnewstrip

210 REPEAT PROCerr

220 CLS:PROCdisplay

230 PRINT''TAB(2):CLS

240 PRINT'TAB(2)"E-EDIT STRIP."

250 PRINT'TAB(2)"P-PRINT STRIP."

260 PRINT'TAB(2)"S-SAVE TO FILE."

270 PRINT'TAB(2)"R-RE-RUN PROGRAM."

280 PRINT'TAB(2)"F-END PROGRAM."

290 REPEAT

300 PRINTTAB(2,14)"Please select ";+CHR$(8);

310 A$=FNget("ERFSP")

320 PRINT A$;

330 e$=FNget(CHR$(13)+CHR$(127))

340 UNTIL e$=CHR$(13)

350 IF A$="R" OR A$="F" THEN IF NOT FNsure A$=""

360 IF A$="E" PROCedit

370 IF A$="S" VDU26:PROCsave

380 IF A$="R" THEN RUN

390 IF A$="P" PROCtype

400 UNTIL A$="F"

410 MODE 7

420 END

430 REM\*DISPLAY CURRENT STRIP\*

440 DEF PROCdisplay

450 LOCAL I%,J%

460 PROCwindow(1)

470 CLS:PRINT

480 FOR I%=0 TO 9

490 PRINT"KEY "+STR$(I%);

500 FOR J%=0 TO 2

510 PRINTTAB(7)key$(I%,J%)

520 NEXT

530 NEXT

540 ENDPROC

550 REM\*CENTRE ON 12 CHARACTER LINE\*

560 DEF FNcentre(a$)

570 IF LEN(a$)=12 THEN =a$

580 LOCAL i%,temp$,pad$

590 pad$=STRING$((12-LEN(a$)) DIV 2," ")

600 temp$=pad$+a$+pad$

610 IF LEN(temp$)=12 THEN =temp$ ELSE =temp$+" "

620 REM\*SINGLE KEY INPUT\*

630 DEF FNget(allowed$)

640 LOCAL in$

650 REPEAT \*FX15,1

660 in$=GET$:IF in$>="a" AND in$<="z" in$=CHR$(ASC(in$) AND &DF

)

670 UNTIL INSTR(allowed$,in$)

680 =in$

690 REM\*CHECK FOR PRINTER OK\*

700 DEF FNprinteron

710 \*FX3,10

720 VDU127,127,127

730 \*FX3,0

740 IF ADVAL(-4)<63 THEN =FALSE:REM\*NO PRINTER\*

750 =TRUE:REM\*PRINTER OK\*

760 REM\*SWAP BETWEEN WINDOWS\*

770 DEFPROCwindow(n%)

780 IF n%=1 VDU 28,0,31,19,0:COLOUR 128:COLOUR 3 ELSE VDU 28,20

,31,39,0:COLOUR 131:COLOUR 0

790 ENDPROC

800 REM\*SECOND CHANCE AT SINGLE KEY INPUT\*

810 DEF FNsure

820 LOCAL A$

830 PRINTTAB(0,16)"Are You Sure(Y/N) ?"+CHR$(7);

840 A$=FNget("YN")

850 IF A$="Y" THEN =TRUE

860 =FALSE

870 REM\*TAKE LINE FROM KEYBOARD-12 CHARACTERS MAX\*

880 DEF FNinline(x%,y%)

890 LOCAL in$

900 PRINTTAB(x%,y%);

910 CALL code%

920 in$=$buffer%

930 in$=FNstrip(in$)

940 =FNcentre(in$)

950 REM\*STRIP LEADING AND TRAILING SPACES\*

960 DEF FNstrip(a$)

970 IF LEFT$(a$,1)<>" " THEN 1010

980 REPEAT

990 a$=RIGHT$(a$,LEN(a$)-1)

1000 UNTIL LEFT$(a$,1)<>" "

1010 IF RIGHT$(a$,1)<>" " THEN =a$

1020 REPEAT

1030 a$=LEFT$(a$,LEN(a$)-1)

1040 UNTIL RIGHT$(a$,1)<>" "

1050 =a$

1060 REM\*START KEY STRIP FROM SCRATCH\*

1070 DEFPROCnewstrip

1080 LOCAL i%,j%

1090 PROCerr

1100 FOR i%=0 TO 9:CLS

1110 PRINT'TAB(4)"Maximum line length 12 characters"

1120 PRINT'''"KEY "+STR$(i%)

1130 PRINTTAB(2,30)"\*PRESS 'ESCAPE' TO END YOUR INPUT\*"

1140 FOR j%=0 TO 2

1150 PRINTTAB(0,10+j%);"Line "+STR$(j%+1)

1160 key$(i%,j%)=FNinline(8,10+j%)

1170 NEXT

1180 NEXT

1190 ENDPROC

1200 REM\*EDIT KEY STRIP\*

1210 DEFPROCedit

1220 PRINT''" Which Key ? ";

1230 key%=VAL(FNget("0123456789"))

1240 PRINT;key%

1250 PRINT'"Which Line ? ";

1260 line%=VAL(FNget("123"))-1

1270 PRINT;line%+1

1280 PRINT'"Type the new version"

1290 key$(key%,line%)=FNinline(4,25)

1300 ENDPROC

1310 REM\*LOAD FROM DISC\*

1320 DEFPROCload

1330 LOCAL i%,j%,fnm$

1340 PROCdrive

1350 REPEAT

1360 PRINTTAB(0,25);SPC(40)

1370 PRINTTAB(0,28)m$;

1380 INPUTLINE TAB(2,25)"Name of file to be loaded "fnm$

1390 UNTIL FNok(FNstrip(fnm$))

1400 CLS:C%=OPENUP(fnm$)

1410 FOR i%=0 TO 9:FOR j%=0 TO 2

1420 INPUT #C%,key$(i%,j%)

1430 NEXT:NEXT

1440 CLOSE #C%

1450 ENDPROC

1460 REM\*SET DRIVE\*

1470 DEF PROCdrive

1480 LOCAL d$

1490 REPEAT REPEAT

1500 CLS:PRINTTAB(0,10)"Please load the disc and then enter the

"'

1510 INPUTLINE"disc drive number "d$

1520 UNTIL LEN(d$)<2:UNTIL VAL(d$)>=0 AND VAL(d$)<4

1530 CLS

1540 PROCoscli("DR. "+STR$(VAL(d$)))

1550 \*CAT

1560 ENDPROC

1570 REM\*SAVE ROUTINE\*

1580 DEFPROCsave

1590 LOCAL i%,j%,fnm$,a$

1600 PROCdrive

1610 REPEAT

1620 PRINTTAB(0,25);SPC(200)

1630 PRINTTAB(0,28)m$;

1640 INPUTLINE TAB(2,25)"File name for saving under "fnm$

1650 UNTIL FNok(FNstrip(fnm$))

1660 IF FNexists(fnm$) PRINTTAB(2,27);SPC(160);TAB(2,27);"This f

ile already exists, do you want to overwrite it with the new on

e ? ";:a$=FNget("YN"):PRINTa$:IF a$="N" THEN 1610

1670 CLS:C%=OPENOUT(fnm$)

1680 FOR i%=0 TO 9:FOR j%=0 TO 2

1690 PRINT #C%,key%(i%,j%)

1700 NEXT:NEXT

1710 CLOSE #0

1720 ENDPROC

1730 REM\*NORMAL ERROR TRAP\*

1740 DEFPROCerr

1750 ON ERROR CLOSE #0:IF ERR=17 THEN 210 ELSE CLS:REPORT:E$=INK

EY$(1000):GOTO 210

1760 ENDPROC

1770 REM\*OUTPUT TO PRINTER\*

1780 DEFPROCtype

1790 REM\*FOR FX80, RX80\*

1800 LOCAL i%,j%

1810 IF NOT FNprinteron PRINT''"No Printer, printer"'"not on lin

e or not"'"powered."''" \*PRESS ESCAPE\*":REPEAT UNTIL FALSE

1820 ON ERROR PROCoscli("FX3 0"):IF ERR=17 THEN 210 ELSE CLS:REP

ORT:E$=INKEY$(1000):GOTO 210

1830 \*FX3,10

1840 PRINTCHR$(15):REM\*CONDENSED MODE

1850 PRINTCHR$(27);"O":REM\*1/8 INCH SPACING\*

1860 PRINTSTRING$(10,"| ");"|"

1870 FOR j%=0 TO 2

1880 FOR i%=0 TO 9

1890 PRINT"|";key$(i%,j%);

1900 NEXT:PRINT"|"

1910 PRINTSTRING$(10, "| ");"|"

1920 NEXT

1930 \*FX3,0

1940 ENDPROC

1950 REM\*STRING TO CLI-BASIC 1 OR 2\*

1960 DEFPROCoscli(m$)

1970 $buffer%=m$+CHR$(13)

1980 X%=buffer% MOD 256:Y%=buffer% DIV 256

1990 CALL &FFF7

2000 ENDPROC

2010 REM\*CHECK FOR EXISTENCE OF FILENAME\*

2020 DEF FNexists(f$)

2030 C%=OPENUP(f$)

2040 CLOSE #C%

2050 IF C%=0 THEN =FALSE THEN =TRUE

2060 REM\*CHECK FILENAME\*

2070 DEF FNok(f$)

2080 IF LENf$<1 OR LENf$>7 THEN =FALSE

2090 LOCAL i%,char$

2100 FOR i%=1 TO LENf$

2110 char$=MID(f$,i%,1)

2120 IF char$<"A" OR char$>"Z" i%=1000

2130 NEXT

2140 IF i%>500 THEN =FALSE ELSE =TRUE

# Appendix 2.

# Some Useful Addresses

I have referred to a number of pieces of software, publications and other products in this book, and you may find it useful to have the information as to where to obtain them.

The user notes on the ROM filing system are produced by Acorn but are available through:

Vector Marketing

London Road

Wellingborough

Northants

The publications of interest are *BBC Microcomputer, ROM Filing System*, which contains the ROM image software, and *BBC Microcomputer, Sideways ROMs*, which deals with language and utility ROMs.

The utility ROMs which I have mentioned in the text are *Toolkit* by Beebugsoft and *Disc Doctor* and *Caretaker* by Computer Concepts. All of these are readily available through retail outlets. The addresses of the companies are:

Computer Concepts Ltd.

Gaddesden Place

Hemel Hempstead

Herts

and

Beebugsoft

P.O. Box 50

St. Albans

Herts

The BREAK key lock which I referred to is from Advanced Technology Products Ltd. It is marketed under the name of *BREAKLOCK* from the following address:

ATPL

Station Road

Clowne

Chesterfield

The wordprocessors referred to are *Wordwise* from Computer Concepts and *View*, which is by Acornsoft and available through Vector Marketing. Alternatively both of these are widely stocked by retailers.

Polish Up Your Programs...

Inside this book, you'll find out how to turn your programs into software to be proud of. You'll learn the techniques used by professionals to make their programs "user friendly" and robust enough to keep working all the year round, even in the hands of the most inexperienced users.

Numerous examples are presented to illustrate the techniques – including input/output, printer techniques, error trapping, and filing methods. There are also some valuable sections on data security (including the prevention of copying) and program documentation.

All listings are in the widely-used BBC BASIC but users of other dialects will find that conversion is straightforward.

...and profit from the results!

After you've mastered these techniques, you'll be able to exploit them to the full. There is a ready market for professional software in the business, educational and home environments. You'll be able to offer your programs to software distributors, or even to set up your own software house!

***About the authors***

Alan and Sue Rowley run their own software business, and are well known for their many articles in the popular computing press.

***About Us***

Sigma have published wll over a hundred books – and we'd like to hear from new authors in computing, science, electronics and engineering.

ISBN 1-85058-033-2

£7.95