# The Working Commodore 64

David Lawrence

A library of practical subroutines and programs

# The Working Commodore 64

A library of practical
subroutines and programs

**David Lawrence**

*Published by:*
Sunshine Books (An imprint of Scot Press Ltd.)
Hobhouse Court,
19 Whitcomb Street,
London WC2 7HF

First published February 1983
Reprinted July 1983
Reprinted September 1983
Reprinted October 1983
Reprinted December 1983

ISBN 0 946408 02 5

*Cover designed by*
Three's Company.
*Photograph of the Commodore 64*
kindly supplied by Commodore UK.
*Typeset and printed in England by*
Commercial Colour Press, London E7.

# CONTENTS

# Contents in detail

# PROGRAM NOTES

A number of functions on the Commodore 64, as with other Commodore machines, are dictated by 'control characters' which are contained in ordinary strings and take effect when the string is printed. Control characters can normally be recognised by the fact that they are inverse characters (the colours of the background and foreground are reversed in the character position). The functions under the control of such characters include cursor position, print colour, inverse (RVS) on and off, cursor home and clear screen.

The following table shows the control characters as they appear in the programs in this book:

| | |
|---|---|
| BLACK | ■ |
| WHITE | ■ |
| RED | ■ |
| CYAN | ■ |
| PURPLE | ■ |
| GREEN | ■ |
| BLUE | ■ |
| YELLOW | ■ |
| ORANGE | ■ |
| BROWN | ■ |
| LIGHT RED | ■ |
| GREY 1 | ■ |
| GREY 2 | ■ |

```
LIGHT GREEN ▌
LIGHT BLUE  ▣
GRAY 3      ▚
RVS ON      ▟
RVS OFF     ▆
UP          ▔
DOWN        ▣
RIGHT       ▐
LEFT        ▌
```

# Calling up Commodore

This book, and the series of which it forms a part, was undertaken to try and fill what seemed to be a yawning gap in the provision of books for home micro-owners. That gap was the absence of works aimed at fulfilling the dream that I think almost every owner has, that the new machine will not simply be a toy, not even an educational introduction to the silicon age, but a tool, taking over all kinds of tasks and opening up all kinds of possibilities. The majority of books consist either of trivia or assume too great a desire—perhaps even the capacity—to experiment.

I wanted to write a book based on a solid collection of programs that would be worth having—programs that would handle such areas as data storage, finance, graphics, music, household management and education. Discussion of programming techniques would arise out of the programs themselves rather than as part of a curriculum of 'things that should be learned'. I hope that you will find the book that has emerged from that desire a useful one, not only as a way of learning new programming techniques, but also as a collection of programs in itself, all of them tested by an independent assessor for errors and offering a wide range of applications that might only have been open to those prepared to buy expensive commercial software or already able to write substantial programs to fit their own needs.

In addition to the programs in this book you have the parts of the programs—not as silly as it sounds for the programs in the book are written in 'modular' form. That is to say they are made up of clearly identifiable functional units which, as you come to understand them, can be lifted out and employed for your own purposes. Each module is commented upon fully where it covers new ground and instructions are given for testing the programs as the modules are entered.

In using this book, though you will find that there are sections where general issues are discussed, it is not a book to be read but to be used. The relevance of the comments and advice will only be apparent when you have taken the plunge and and begun the task of entering what appear at first to be dauntingly long and complex programs. Here, the modular approach will help to prevent programs becoming unredeemable tangles of errors, so do test modules as suggested, especially in the early stages.

In the end however, the success or failure of the book must be judged on whether it helps you to enjoy your 64. It is very much a '64 book', for while the general structure of the book is based upon its two predecessors in this series, the programs were adapted and new programs added to take account of the 64's extraordinary abilities. While writing a book such as this is hard work, I have nevertheless enjoyed the polish that the 64 has given to programs that on less capable machines might have been far less exciting. In using these programs you won't have to work quite as hard—but the end product will be just as exciting.

Finally, no introduction to a book such as this could end without expressing profound thanks to Commodore UK for all the facilities they have made available and not least to Steve Beats at Commodore's UK headquarters, for his patience in answering the idiotic questions that opened up the 64 for me.

# CHAPTER 1
## Good things in small packages

The programs in this book are intended to be put to work on a variety of important applications. Because many of the applications are complex, so are many of the programs. That should not be taken to mean that useful programs cannot be compressed into a small space. As an introduction to the approach adopted, this chapter presents three relatively short programs that are anything but toys.

## 1.1 CLOCK

This program provides quite a pleasant introduction to some of the 64's abilities—it's easy to enter, fun to leave running on the family TV and it makes good use of the 64's flexible string and screen handling.

The program is exactly what it says, a clock, but you won't see a circle and hands appear when it is run. The 64 clock uses two lines sweeping across the screen, left to right for the minutes and downwards for the hours, dividing the screen into different colour areas. All of this is only possible because the 64 has a flexible time function which can be set and read in a straightforward way from within the program.

**Clock: Table of Variables**

| | |
|---|---|
| CS | Address of the start of colour memory |
| DT$ | Formatted adaptation of TI$ |
| H | Hour value adjusted into screen units |
| M | Minute value adjusted into screen units |
| M1$,M2$ | Two-colour strings displaying hour and minute values |
| SS | Address of the start of the screen |
| TI$ | A system variable containing the time by the internal clock |

MODULE 1.1.1

```
11000 REM******************************
11010 REM INITIALISE TIME AND DISPLAY
11020 REM******************************
11030 POKE 53280,0:POKE 53281,1
11040 INPUT "[illegible]PLEASE INPUT THE HOUR (0
1-12):";H$
```

```
11050 INPUT "▓▓PLEASE INPUT THE MINUTE (
00-59):";M$
11060 TI$=H$+M$+"00"
11070 PRINT "▒▒█     5 10 15 20 25 30 35
40 45 50 55 60  "
11080 SS=1024:CS=55296:FOR I=0 TO 24
11090 POKE CS+40*I,0:POKE SS+40*I,160
11100 POKE CS+40*I+1,0:POKE SS+40*I+1,16
0
11110 POKE CS+40*I+38,0:POKE SS+40*I+38,
160
11120 POKE CS+40*I+39,0:POKE SS+40*I+39,
160
11130 NEXT I
11140 PRINT "▓":FOR I=1 TO 11:PRINT "▓";
MID$(STR$(I),2):PRINT:NEXT
11150 PRINT "▓";MID$(STR$(I),2);
```

This module allows the user to input the time in hours and minutes (12 hour clock format), sets the timer and then displays the clock face.

*Commentary*

Line 11030: Two useful memory locations: 53280—redefines the colour of the border around the screen, 53281 redefines the screen background colour. Either of these can be reset instantaneously during the course of a program. In this case the border is set to black and the screen to white.

Lines 11040–11060: Hours and minutes are input in two digit form. They are then added together and 00 is added for the seconds. The system is told that this is TI$ and immediately resets the internal clock to count from that time.

Line 11070: The screen is cleared, the print colour set to black and reverse is set, then the figures are printed across the top of the screen.

Lines 11080–11130: The black borders of the clock area are now put onto the edge of the screen. When printing right to the edge of the screen it is often more convenient to POKE characters onto the screen, since this avoids the print position jumping from one line to the next. In order to POKE the screen successfully, two locations must be dealt with, one within the screen memory itself (addresses 1024–2023) and the other within the colour memory (55296–56295). All that this loop does is to POKE the first two characters and the last two characters of the 25 lines on the screen with a character code of 160 (an inverse space) and the corresponding location in the colour memory with zero, which turns that character position black.

Line 11140: The cursor is homed and the hours are printed down the left hand side of the screen. The last value is printed separately with a semi-colon following, so that the screen will not scroll upwards, since it is on the bottom line of the screen.

*Testing Module 1.1.1*

Insert a temporary line 11160 GOTO 11160 and run the module. You should be asked to input hours and minutes, then the borders of the clock face will be placed onto the screen. The temporary line ensures that the screen does not scroll upwards to print READY when the module is finished.

MODULE 1.1.2

```
12000 REM********************************
12010 REM CALCULATE AND DISPLAY TIME
12020 REM********************************
12030 M=INT((VAL(MID$(TI$,3,2))+0.8)*3/5
)
12040 H=2*VAL(MID$(TI$,1,2))
12050 IF M>=30 THEN LET H=H+1
12060 IF H>=24 THEN LET H=H-24
12070 IF M=0 THEN LET M=1
12080 M1$="[illegible]
          "
12090 LET M1$=LEFT$(M1$,M+4)+"[illegible]"+RIGHT$(
M1$,36-M)
12100 M2$="[illegible]
          "
12110 LET M2$=LEFT$(M2$,M+4)+"[illegible]"+RIGHT$(
M2$,36-M)
12120 PRINT "[illegible]";
12130 IF H>0 THEN FOR I=1 TO H:PRINT M1$
:NEXT
12140 IF H<23 THEN FOR I=H+1 TO 23:PRINT
 M2$:NEXT
12150 PRINT M2$;
12160 PRINT "[illegible]
[illegible]";
12170 DT$=LEFT$(TI$,2)+"  "+MID$(TI$,3,2
)+"  "+RIGHT$(TI$,2)
```

```
12180 FOR I=1 TO LEN(DT$):PRINT MID$(DT$
,I,1);"██";:NEXT
12190 GOTO 12030
```

This module derives the values necessary to create the display from the internal clock and displays the time in two forms.

*Commentary*

Line 12030: There are 36 available spaces across the screen once the borders have been drawn, so the number of minutes must be divided by 60/36 (5/3) to obtain the right units to move across the screen.

Line 12040: There are 24 screen lines available, so all that is necessary is to multiply the hours by 2.

Line 12070: The program is designed always to display minutes, so that on the hour the minute value increments to show one unit.

Line 12080: M1$ is set equal to two cursor moves to the right plus the purple control character and the reverse control character, followed by 36 spaces. If printed, this would show a purple line.

Line 12090: M1$ is now changed so that it becomes equal to the first four control characters plus M spaces, then a red control character, then the remaining spaces. This creates a new string which changes colour at a point defined by the value of M.

Lines 12100–12110: The same process is carried out for M2$, which will begin blue and end white.

Lines 12130–12150: M1$ is printed for as many lines as there are hours; M2$ is printed on the remainder of the lines. The two strings thus define a border between different colour areas dictated by the value of H.

Line 12160: The cursor is homed and the print position moved to about one third of the way down the penultimate column of the screen, with the print colour set to black.

Line 12170: DT$ is now defined as TI$ with two spaces inserted between the hour, minute and second values. Throughout the course of the program, the system has been updating TI$ so that it always contains the latest time.

Line 12180: DT$ is printed down the right hand side of the screen. The method used is to print one character at a time, then move the cursor down and back.

*Testing Module 1.1.2*

Your clock should now be ready to run, with four different rectangles of colour marking the lines for hours and minutes. The time is displayed digitally on the right-hand side of the screen.

Having said what it should do, almost inevitably there will be errors in what you have entered. If not here, then in later programs. From the kind of queries that come to me it seems that many micro-owners find it very difficult to know how to begin to deal with such errors and perhaps a few basic guidelines might be of help:

1) Make the most of the help available to you. If there is an error message, make sure you take account of it, noting the line where the error occurs and the type of error.
2) Don't run the program again to see if it will work a second time. If it does work then you are in a worse state than you started since you have lost the chance of running down the error for the present.
3) Use the direct mode (commands entered directly from the key board rather than program lines) to print out the values of all the variables in lines that appear to have an error. A ludicrous value will often give you the clue as to what is going wrong. An awful lot of almost indetectable errors result from the simple misspelling of a variable name, substituting 1 for I for instance.
4) Follow the program through in your head or on paper, using simple values so that you can see exactly what it should be doing at each point.
5) Don't be too hasty in making an alteration until you are sure that it is the only one you want to make. Once you enter a change to a line, all your data disappears and with it your chance to make further checks without running the program again.
6) Save your program regularly as you discover errors and/or add new lines. Many errors in final programs result from changes which were entered into a program but never finally recorded on tape. All my programs commence with the following 3 lines:

```
1 GOTO 3
2 SAVE 'XXXX':STOP
3 REM
```

These three lines allow programs to be saved with the command 'GOTO 2' (provided 'XXXX' is replaced with the program name. One incidental side-benefit is that I can always start my programs with GOTO 1 rather than having to remember the first line number of the main program.

Everybody makes mistakes in designing and entering programs, the difference is whether they learn to cope with their mistakes competently.

### *Summary*

Whether you like the clock is something only you can say. Personally I find it quite attractive. Regardless of the clock, however, the techniques of slicing up strings and of POKEing the screen and colour memory contained within the program will come in useful in a wide variety of programs, so it is worth entering the program and ensuring that you understand how it functions.

## 1.2 GRAPH

If you want to understand this program you can do no better than to look at the box in which your 64 arrived. There you will find a colourful three-dimensional bar chart—this program is an attempt to reproduce the prбgram that generated the chart. I say attempt, because on successfully reproducing the chart on the box I discovered that the data it was given to work on had been carefully chosen to hide the limitations of the tightly packed bars. Other data led to the graphics characters making up the bars knocking holes in neighbouring bars, making the whole thing a great deal less attractive than the box display.

This program, then, is a compromise, producing a less packed display but one which will work on any set of data and still look as good—so good, in fact, that when you have completed entering it, it is the kind of program to call the family in to impress them with your wizardry.

Colourful and practical, the displays produced will no doubt find many applications. In addition the program provides a simple introduction to the subject of saving data on tape and later reloading it.

### Graph: Table of Variables

| | |
|---|---|
| CO$ | Three-character string used to decide colour of different bars on the graph |
| F$ | Formatting string of right cursor characters |
| F1$,F3$ | Formatting strings of down cursor characters |
| F2$ | Temporary string derived from F$ |
| HH(2,6) | Array holding data for graph |
| NB | Number of banks in front of each other (1-3) |
| ND | Number of columns along the horizontal axis (1-6) |
| NH$ | Name for horizontal axis |
| NV$(2) | Names for each separate bank |
| TT$ | |

| | |
|---|---|
| | Temporary string used to format printing of vertical axis names |
| UV | Number to be represented by each unit on the vertical axis |

MODULE 1.2.1

```
11000 REM#*******************************
11010 REM ACCEPT DATA
11020 REM********************************
11030 POKE 53281,15:INPUT "♥▮▮DO YOU WIS
H TO LOAD FROM TAPE (Y/N):";Q$
11040 IF Q$="Y" THEN 12420
11050 PRINT "♥▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮GRAPH"
11060 PRINT "▮▮THERE ARE 19 UNITS ON THE
 VERTICALLY."
11070 PRINT "▮▮INPUT NUMBER TO BE REPRES
ENTED BY EACH"
11080 INPUT "UNIT:";UV
11090 INPUT "▮▮NAME FOR HORIZONTAL AXIS"
;NH$
11100 PRINT "♥▮▮YOU CAN HAVE ONE TO SIX
COLUMNS."
11110 INPUT "▮HOW MANY WOULD YOU LIKE:";
ND
11120 PRINT "▮▮▮YOU CAN HAVE ONE TO THRE
E BANKS."
11130 INPUT "▮HOW MANY WOULD YOU LIKE:";
NB
11140 FOR I=0 TO NB-1
11150 PRINT "▮▮▮NAME FOR VERTICAL AXIS";I
+1;:INPUT NV$(I):NEXT I
11160 DIM HH(2,6)
11170 PRINT "♥";:FOR I=0 TO NB-1
11180 FOR J=1 TO ND
11190 PRINT "▮INPUT BANK";I+1;" VALUE";J
;":";:INPUT T
11200 IF INT(T/UV)>19 THEN PRINT "▮VALUE
TOO HIGH.":GOTO 11190
11210 HH(I,J)=T:NEXT J,I
```

The purpose of this straightforward module is to allow the input of the data which will be used to build up and label the graph. Rather than ask the user to input maximum and minimum figures for the range of values and then calculate the units (which can result in extremely odd units), the program asks the user to specify how many units of the data input will be represented by each vertical unit on the graph. Names are given to the horizontal axis and to the banks of 3-D pillars, starting from the back. Finally data is requested in conformity with the structure chosen by the user.

*Testing Module 1.2.1*

Simply a matter of running the module to check the syntax is correct. Nothing can be drawn yet.

MODULE 1.2.2

```
12000 REM********************************
12020 REM DRAW GRAPH
12030 REM********************************
12040 POKE 53281,0:PRINT "■■■■■■■■■■■■■■■■■
■■■■■■■■■";
12050 F$="■■■■■■"
12060 FOR I=1 TO 4:PRINT F$;"■■
                        ■":F$=F$+"■":NEXT
12070 PRINT "■■■■■■■______________________________
____________":REM 30 CHAR LINE
12080 FOR I=1 TO 19:PRINT "■■■■■L";
12090 PRINT "
 _|":NEXT
12100 PRINT "■";:FOR I=1 TO 4:PRINT "■■■
■■■■■■■";
12110 PRINT "________________________________________
_":NEXT:REM 28 CHAR LINE
12120 F1$="■■■■■■■■■■■■■■■■■■■■■■■■■■":F$="■■■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■":CO$="■■■■"
12130 PRINT "■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■";
NH$;
12140 FOR H=0 TO NB-1:PRINT "■■■■■■■■■■";
MID$(F$,1,2*(H+1));MID$(CO$,H+1,1);
12150 TT$=NV$(H)+" *"+STR$(UV):FOR I=1 T
O LEN(TT$)
12160 PRINT MID$(TT$,I,1);"■■■";:NEXTI,H
12170 PRINT "■■■■■■■■15■■■■■■■10■■■■■■■5
■■■■■■■0"
12180 F3$=F1$:FOR H=0 TO NB-1:PRINT MID$
(CO$,H+1,1)
12190 F2$=LEFT$(F$,8+4*(ND-1)+H):PRINT F
1$;F2$;
12200 FOR I=ND TO 1 STEP-1:IF INT(HH(H,I
)/UV)=0 THEN 12270
12210 FOR J=1 TO INT(HH(H,I)/UV)+H
12220 IF INT(HH(H,I)/UV)=0 THEN 12270
12230 IF J=1 THEN PRINT "[illegible] [illegible]";MID$(CO$,
H+1,1);"|  [illegible]";
12240 IF J>1 THEN PRINT " | [illegible]  [illegible]";
12245 REM CHARS ARE " [illegible]"
[illegible]";
12250 NEXT J
12260 PRINT " [illegible] [illegible]"
12270 PRINT:F2$=LEFT$(F2$,LEN(F2$)-4):PR
INT F1$;F2$;:NEXT I
12280 F1$=F1$+"[illegible]"
12290 F2$=LEFT$(F$,9+4*(ND-1)):PRINT F3$
;F2$;
12300 NEXT H
12310 FOR I=1 TO ND:PRINT F3$;
12320 F2$=LEFT$(F$,9+4*(I-1)):PRINT F2$;
:FOR J=1 TO NB
12330 IF J>1 OR HH(2,I-1)=0 THEN PRINT "
[illegible]";
12340 PRINT "[illegible]";:NEXT J,I
12350 GET A$:IF A$="" THEN 12350
```

This is a fiddly module which is based, not on any clear set of methods but simply on the conditions that I found, in practice, had to be fulfilled to complete an attractive presentation of the graph.

*Commentary*

Lines 12040–12060: The screen is set black and the brown base on which the graph stands is painted in.

Lines 12070–12110: The grid surrounding the graph area is drawn, units are marked down the sides and lines placed across to mark the five unit levels.

Lines 12140–12160: These three lines determine a print position at the top right hand corner of the screen and print the names of the three banks there, down the screen, in colours corresponding to the banks themselves. The screen position is determined using a chunk taken out of F$ and the colour by printing a different one of the three colour control characters for each execution of the loop.

Lines 12180–12300: Three loops are called upon in this section. The H loop determines how many banks will be drawn in front of one another, being also used to extract a colour control character from CO$ and to determine how many down cursor characters will be printed, thus moving the banks down consecutively. The I loop controls how many columns will be printed across the screen, the J loop controls how high any particular 3-D block will be.

Line 12190: The horizontal print position at the start of each bank (they are drawn from right to left) is calculated according to the bank—each bank moves across one, giving the 3-D appearance to the three banks.

Line 12200: A column is not printed if the relevant array element in the array HH is zero.

Lines 12230–12240: At the bottom of each column is printed the slanting bottom which makes the column look as if it is resting on the surface.

Line 12260: When the top of the column is reached the sloping top is added.

Line 12270: For the next column, four characters are subtracted from the cursor right string, defining the new print position.

Line 12280: The vertical print position is moved down before printing the next bank.

Lines 12310 – 12340: In printing the banks the bottoms of the columns have been corrupted. These are filled in.

Line 12350: The graph remains on the screen until a key is pressed.

*Testing Module 1.2.2*

Once again a straightforward matter of running the program and seeing that the resultant display does look right. If you run into problems, then the answer is to cut down the number of banks to one, and perhaps even the number of columns to one, to simplify your analysis of what is going awry. Look at the function of each of the loops separately to decide which of them appears to be producing the error. This is a frustrating module to debug so, if you think that a change to your lines would overcome the problem, even though you can't see where you have departed from the listing given here, make the change and see—nothing here or anywhere else in this book is sacrosanct.

MODULE 1.2.3

```
12360 INPUT "□■▨DO YOU WISH TO SAVE DATA
 (Y/N):";Q$:IF Q$="N" THEN END
12370 INPUT "▨POSITION TAPE CORRECTLY, T
HEN ▨RETURN■--";Q$:R$=CHR$(13)
12380 OPEN 1,1,1,"GRAPH"
12390 PRINT#1,NB;R$;ND;R$;NH$;R$;UV
12400 FOR I=0 TO NB-1:PRINT#1,NV$(I):FOR
 J=0 TO ND:PRINT#1,HH(I,J):NEXT J,I
12410 CLOSE1:END
12420 INPUT "□■▨POSITION TAPE CORRECTLY
THEN ▨RETURN■--";Q$:DIM NV$(2),HH(2,6)
12430 OPEN 1,1,0,"GRAPH"
12440 INPUT#1,NB,ND,NH$,UV
12450 FOR I=0 TO NB-1:INPUT#1,NV$(I):FOR
 J=0 TO ND:INPUT#1,HH(I,J):NEXT J,I
12460 CLOSE1:GOTO 12000
12470 GOTO 12470
```

Now that you have defined your graph, rather than lose the data and have to enter it again, you can store it on tape. This module will allow you to do that and to recall it subsequently. The module is designed to make tape storage as easy as possible in that it gives you time to position your tape to the correct place before it begins the process of loading or saving.

*Commentary*

Line 12380: This line opens a file, a place into which data is to be placed, and in this case the storage place is the cassette recorder. The three figures represent:

a) The number of the file—any instructions to store something in a file must mention the file number
b) The device number (the piece of equipment which is to receive the data) with 1 representing the cassette recorder
c) The type of file—1 means that it is a file into which data is to be placed rather than one from which data is to be taken.

NB,ND,NH$ and UV are placed into the file (onto tape). Note the use of the variable R$ here. When storing data on tape the 64 is a little finnicky about how each item is separated from the next—simply putting in commas can result in errors when the data is reloaded. R$ was defined in Line 12370 as CHR$(13), the code for RETURN, and placing it between items ensures that they are properly separated.

Line 12400: The program's arrays are printed one by one to the file.

Line 12410: When you have finished with a file for the time being it must be CLOSEd. Failure to do this will result in an error the next time you try to OPEN a file of the same number.

Line 12420: This is the part of the module which loads back data into the 64. The only difference between the specifications for the two types is that this one has a file type of 0, which means a file from which data will be taken.

Lines 12440 – 12450: Data which was printed into the file is now recalled. The safest way to build up your loading routine is to edit the line numbers of the SAVE routine and change the PRINT commands to INPUT. That way you know that the routine will pick up the data in exactly the same order as it was stored. If the data is picked up in the wrong order, not only will it make nonsense of your program but an error may result in the program stopping.

*Testing Module 1.2.3*

The simple test for this module is whether you can input data to the program, save it on tape and then reload it.

***Summary***

This program is a tribute to the quality of the 64's graphics set and screen handling. Once you have entered it you begin to see that it is not at all such a difficult thing to use loops and simple calculations to draw shapes and apparently solid objects at controlled places on the screen and that such displays are one of the most effective ways of getting the facts across that you will ever find.

## 1.3 TEXTED

The final program in this chapter is an attempt to provide some of the simpler functions of a word processor in a relatively short and uncomplicated program. The program is, of course, no match for a professionally written machine-code based word processor, nor would it be the tool I would choose for writing a book like this one, if only because Commodore UK were kind enough to provide me with a copy of their excellent Easyscript program for the purpose. Nevertheless the program works and I would and have chosen it in preference to a typewriter for many purposes because of the flexibility it provides in entering and editing text before it is finally output onto paper. Of course, if you don't possess a printer, then you will need to rush out and buy one to get the most out of the program.

**Texted: Table of Variables**

| | |
|---|---|
| A$ | Line of text being entered |
| CH | Code of character under flashing cursor |
| FNA(P) | Calculates position in memory of flashing cursor in text being entered |
| FNB(P2) | Calculates position in memory of edit cursor in main body of text |
| LL | Number of lines of text in main body of text |
| P | Position of flashing cursor in line being entered |
| P2 | Position of edit cursor down screen |
| PL | Line number of edit cursor in main body of text |
| SP | Number of spaces available at end of line when formatting |
| SS | Line number of first line of part of main text being displayed |
| T$ | Last character input when entering new line of text |
| T1$ | Character input as command in Move Edit Line module |
| TEXT$(500) | Main array for the storage of text |
| TT$ | Temporary storage of lines being entered into main body of text |
| X | Number of lines of text extracted from batch of text being inserted into main body |

MODULE 1.3.1

```
11000 REM#**********************************
11010 REM INITIALISE
12000 REM***********************************
12010 PRINT"❐";:DIM TEXT$(500):LL=1:PL=1
12020 TEXT$(0)="π▼▼▼▼▼▼▼▼▼▼▼▼
▼▼▼▼▼▼▼▼▼"
12030 TEXT$(1)="π◩▼▼▼▼▼▼▼▼▼▼▼
▼▼▼▼▼▼▼▼▼▼"
12040 DEF FNA(P)=1024+20*40+P
12050 DEF FNB(P2)=1024+40*P2
12060 GOSUB 14110
```

This initialises the main variables and places beginning and end of text markers into the main array.

MODULE 1.3.2

```
13000 REM#**********************************
13010 REM EDIT LINE
```

```
13020 REM*****************************
13030 A$=" "
13040 P=0:PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮"
;A$
13050 CH=PEEK(FNA(P)):POKE 54272+FNA(P),
14:POKE FNA(P),160
13060 FOR TT=1 TO 5:NEXT TT:POKE FNA(P),
CH
13070 GET T$:IF T$="" THEN GOTO 13050
13080 IF T$=CHR$(13) OR LEN(A$)=81 THEN
GOSUB 14000:GOTO 13050
13090 IF T$="↑" THEN GOSUB 15000:POKE FN
B(PL-SS),62:GOTO 13040
13100 IF T$="←" AND P<>0 THEN 13040
13110 IF T$="←" ANDP=0 THEN P=LEN(A$)-1:
GOTO 13150
13120 IF P>0 AND T$=CHR$(20) THEN A$=LEF
T$(A$,P-1)+MID$(A$,P+1):P=P-1:GOTO13150
13130 IF T$=CHR$(20) THEN 13050
13140 IF T$<>""ANDT$<>"▮" AND T$<>"▮" TH
EN A$=LEFT$(A$,P)+T$+MID$(A$,P+1):P=P+1
13150 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮";A$:I
F T$="▮" AND P>0 THEN P=P-1
13160 IF T$="▮" AND P<LEN(A$)-1 THEN P=P
+1
13170 GOTO 13050
```

The purpose of this module is allow the user to input and edit up to two screen lines of text at the bottom of the screen and to edit those lines in preparation for inserting them into the main body of the text with a subsequent module.

## *Commentary*

Lines 13050–13070: Our first encounter with a flashing cursor routine. On the basis of user-defined function A, these lines PEEK the screen memory at the point indicated by the variable P and obtain the code of the character located there. An inverse blue space is then POKEd into the same location, left there for the duration of a short timing loop and then replaced by the original character. If there has been no input from the keyboard, as indicated by the GET statement, the process is repeated.

Line 13080: Pressing RETURN inserts the line into the main body of text —a later module is needed. The line is also inserted automatically if the length exceeds two screen lines—this may strip the final characters from the line.

Line 13090: Pressing the '↑' symbol allows the user to move into a subsequent module which acts on the main body of text.

Lines 13100–13110: Pressing the left arrow symbol at the top left of the keyboard moves the cursor either to the beginning of the line or to the end, depending on its current position.

Lines 13120–13130: Provided that the cursor is not positioned at the beginning of the line, pressing DELETE removes the character before the cursor. Note that, using GET, the control keys such as DELETE have no effect unless they are PRINTed, so that if we do not print them we can redefine their function.

Line 13140: If the character entered is not a cursor move arrow, then it is assumed to be a character to be printed and it is added to the string in the position of the cursor. If the cursor is in the middle of the string, then the character is added in—it does not replace the character under the cursor.

Lines 13150–13160: The string is reprinted in its changed form. If the input was a cursor arrow then the cursor position is changed accordingly.

*Testing Module 1.3.2*

By entering a temporary RETURN at Line 14000, you should now be able to input text to the bottom of the screen and to edit that text.

MODULE 1.3.3

```
14000 REM#*******************************
14010 REM INSERT LINE
14020 REM********************************
14030 X=0
14040 IF LEN(A$)<41 THEN TT$(X)=LEFT$(A$
,LEN(A$)-1):A$="":GOTO 14070
14050 FOR I=41 TO 1 STEP-1:IF MID$(A$,I,
1)<>" " THEN NEXT I:I=41
14060 TT$(X)=LEFT$(A$,I-1):A$=MID$(A$,I+
1)
14070 X=X+1:IF A$<>"" AND A$<>" " THEN G
OTO 14040
14080 FOR I=LL+X TO PL+X STEP-1:TEXT$(I)
=TEXT$(I-X):NEXT I
14090 FOR I=0 TO X-1:TEXT$(PL+I)=TT$(I):
NEXT
14100 A$=" ":P=0:PRINT "□";:LL=LL+X:PL=P
L+X
```

```
14110 SS=PL-7:IF LL-PL<8 THEN SS=LL-15
14120 PRINT "▀▀████████████████████";A$;
"█";:IF SS<0 THEN SS=0
14130 FOR I=SS TO SS+15:PRINT "█";TEXT$(
I);:IF LEN(TEXT$(I))<40 THEN PRINT
14140 IF I=PL-1 THEN PRINT CHR$(62)
14150 NEXT I:PRINT "██
                              ":RETURN
```

The purpose of this module is to enter the current line into the main body of text.

*Commentary*

Line 14040: If the line being inserted is less than 41 characters long it is placed into the position indicated by the > edit cursor.

Lines 14050–14070: If the line is longer than 41 characters, these lines search back for the ending of the last word which will fit fully onto the line and make everything to the left of it the first line to be inserted and store it in TT$. A$ is now redefined as what is left and the process is repeated. The variable X records how many lines result.

Lines 14080–14090: The main body of text, from the edit cursor on, is shifted to make room for the X new lines and the new lines are inserted.

Lines 14100–14110: A$ is set to a single space, the flashing cursor position is set to zero and the edit cursor is moved down below the new lines. The start of main text display is redefined so that the edit cursor remains roughly in the middle.

*Testing Module 1.3.3*

You should now be able to insert lines of text into the main body of text by pressing RETURN.

MODULE 1.3.4

```
15000 REM********************************
15010 REM MOVE EDIT LINE
15020 REM********************************
15030 P2=PL-SS
15040 GET T1$:IF T1$<>"" THEN 15070
15050 POKE 54272+FNB(P2),8:POKE FNB(P2),
62:FORIT I=1 TO 20:NEXT
15060 POKE FNB(P2),32:GOTO 15040
```

```
15070 PL=PL+(T1$="⌝")+10*(T1$="U"):IF PL
<1 THEN PL=1
15080 PL=PL-(T1$="▩")-10*(T1$="D"):IF PL
>LL THEN PL=LL
15090 IF T1$=CHR$(13) THEN RETURN
15100 IF PL=>LL OR T1$<>CHR$(20) THEN 15
120
15110 LL=LL-1:FOR I=PL TO LL:TEXT$(I)=TE
XT$(I+1):NEXT:TEXT$(LL+1)=""
15120 IF PL<LL AND T1$="C" THEN A$=TEXT$
(PL)+" ":RETURN
15130 IF T1$="P" THEN GOSUB 17000
15140 IF T1$="S" THEN GOSUB 18000
15150 IF T1$="F" THEN GOSUB 16000
15160 GOSUB 14110:GOTO 15030
```

This module allows the main edit pointer to be moved about in the main body of text, thus allowing lines to be inserted at different points. From this module the user is also able to call up other modules which format the text, output it to a printer or save it to tape.

*Commentary*

Lines 15040–15060: Flashing cursor routine for the main edit cursor.

Lines 15070–15080: These two lines move the main edit cursor up and down. Single line moves are accomplished by the ordinary cursor move arrows. Pressing U or D will result in a 10 line jump. Note the use of logical conditions to accomplish these moves. The expression (T1$ = 'U') has a value of zero when the condition is false and of minus one when it is true and can thus be used to economically replace an IF statement such as IF T1$ = 'U' THEN etc.

Line 15090: Pressing RETURN will return to the text entry module.

Lines 15100–15110: Pressing DELETE removes the line beneath the cursor.

Line 15120: Pressing C copies the line beneath the cursor to the bottom of the screen for further editing.

*Testing Module 1.3.4*

You should now be able to move the main edit cursor, to delete lines and to copy them back to the bottom of the screen.

MODULE 1.3.5

```
16000 REM#******************************
16010 REM FORMAT LINE
16020 REM*******************************
16030 FOR I=1 TO LL-2:IF TEXT$(I)="" OR
TEXT$(I+1)="" THEN 16120
16040 SP=40-LEN(TEXT$(I)):FOR J=1 TO LEN
(TEXT$(I+1))
16050 IF MID$(TEXT$(I+1),J,1)<>" " THEN
NEXT J:J=J-1
16060 IF SP<J OR J=LEN(TEXT$(I+1)) THEN
16090
16070 TEXT$(I)=TEXT$(I)+" "+LEFT$(TEXT$(
I+1),J-1)
16080 TEXT$(I+1)=MID$(TEXT$(I+1),J+1):GO
TO 16040
16090 IF LEN(TEXT$(I+1))=>SP THEN 16120
16100 TEXT$(I)=TEXT$(I)+" "+TEXT$(I+1)
16110 FOR J=I+1 TO LL:TEXT$(J)=TEXT$(J+1
):NEXT J:LL=LL-1:PL=PL-1:GOTO 16040
16120 NEXTI:RETURN
```

This module formats the text, that is to say the text is rearranged so that empty spaces at the end of lines are, where possible, filled with words from the subsequent lines.

Line 16030: When an empty line is entered into the main body of text it will not be formatted. Empty lines can thus be used to separate paragraphs or other lines the user does not wish to have run together.

Lines 16040–16080: The space at the end of the line is calculated and an assessment is made of whether there is a word at the beginning of the next line which will fit the space—if so it is transferred.

Lines 16090–16110: If the whole of the next line will fit onto the end of the current line then it is added and the file collapsed to cover the resulting space.

*Testing Module 1.3.5*

If you input a series of single word lines to the main body of text you should now be able to enter the main edit mode, press F and see the words run together into continuous lines. You can also insert short lines into the middle of the main text and then reformat it.

MODULE 1.3.6

```
17000 REM#********************************
17010 REM OUTPUT TO PRINTER
17020 REM*********************************
17030 OPEN 1,4:X=1
17040 IF X=LL THEN 17100
17050 IF TEXT$(X)="" THEN PRINT#1,"":X=X
+1:GOTO 17040
17060 PRINT#1,TEXT$(X);" ";
17070 IF X+1=LL THEN 17100
17080 PRINT#1,TEXT$(X+1):IF TEXT$(X+1)="
" THEN PRINT#1,""
17090 X=X+2:GOTO 17040
17100 PRINT#1,"":CLOSE1:RETURN
```

This simple module opens communication with the printer (device 4) and prints out the main body of text. Text is printed in 80 column format (ie two screen lines make one printed line), and clear lines are printed wherever there is a clear line in the main text. Note that though the program itself will happily deal with characters in lower case mode (press SHIFT and COMMODORE key together), most printers require a special command to actually output lower case characters. This is not provided since it differs from printer to printer. Your printer manual will provide the necessary information.

MODULE 1.3.7

```
18000 REM#********************************
18010 REM DATA FILES
18020 REM*********************************
18030 PRINT "▯█▚POSITION TAPE CORRECTLY,
 THEN ▚RETURN█←–"
18040 INPUT "MOTOR WILL STOP AUTOMATICAL
LY:";Q$:POKE 192,7:POKE 1,39
18050 PRINT "COMMANDS AVAILABLE:":PRINT
"██1)SAVE DATA":PRINT "█2)LOAD DATA"
18060 INPUT "██WHICH DO YOU REQUIRE:";Q:
ON Q GOTO 18080,18120
18070 RETURN
18080 POKE 1,7:FOR I=1 TO 2000:NEXT
18090 OPEN 1,1,2,"TEXTED":PRINT#1,PL:PRI
```

```
NT#1,LL
18100 FOR I=0 TO LL:FF$=TEXT$(I)+"@":PRI
NT#1,FF$:NEXT I
18110 CLOSE1:RETURN
18120 OPEN 1,1,0,"TEXTED":INPUT#1,PL,LL
18130 FOR I=0 TO LL:INPUT#1,TEXT$(I):NEX
T:CLOSE1
18140 FOR I=0 TO LL
18150 IF TEXT$(I)<>"@" THEN TEXT$(I)=LEF
T$(TEXT$(I),LEN(TEXT$(I))-1)
18160 IF TEXT$(I)="@" THEN TEXT$(I)=""
18170 NEXT I:RETURN
```

A standard data-file module.

### *Summary*

The techniques used in this program of altering something while you are looking at it on the screen bear some study since this is by far the easiest way (for the user) of altering strings and can be written into a variety of programs where string data, having been input has to be changed—including, if you wanted to, most of the programs in this book.

**Texted: Summary of one-key instructions**

Text entry mode:

Text characters may be entered at position of flashing cursor.
Left and right arrows move cursor over string.
'←' moves cursor to beginning or end of line. '↑' calls up main edit module.
RETURN places current string into the main body of text.

Main edit mode:

| | |
|---|---|
| RETURN | returns to previous mode |
| U,D | and up and down cursor arrows move main edit cursor |
| DELETE | removes line beneath main edit cursor |
| C | copies line beneath main edit cursor |
| P | sends text to printer |
| S | saves text on tape |
| F | formats text |

# CHAPTER 2
## Programming Tools

Having been introduced to some of the 64's capabilities we now depart from the normal format of the book for a brief space to present three tightly packed programs which will provide you with essential tools enabling you to merge separate programs together,renumber them and delete whole sections with ease. The programs are densely packed for the simple reason that, using the Merge routine, they are intended to be strung together and then added to the end of existing programs without taking up too much memory space. When you have finished merging in extra sections to your program and renumbering it, the Delete routine will happily delete itself and its two companions!

## 2.1 MERGE

This program, together with the other two presented in this short chapter, is a must for those who intend to take modular programming seriously. Using this tiny program hours of work can be saved by keeping useful modules on tape and simply stringing them together with the press of a tape recorder button. In presenting the program I am indebted to Steve Beats of Commodore UK, who suggested to me the basic idea from which it was developed.

There are no modules in the program—at eight lines it would hardly be worth it and yet a program such as this will make modular programming come alive for you.

What the program does is to pick up another program, or section of a program from tape and to enter it into the 64 without danger of losing what is already there—unless the line numbers coincide, in which case the first program will be overwritten.

MODULE 2.1.1

```
63990 PRINT "⊐";
63991 OPEN 1,1,0,"TEST"
63992 POKE 184,1:POKE 185,96:POKE 186,1:
POKE 152,1:PRINT "⊐▣"
63994 GET#1,A$:PRINT A$;:IF ST THEN 6399
9
```

```
63995 IF A$<>CHR$(13) THEN 63994
63996 PRINT "GOTO 63992■";:POKE 631,13:P
OKE 632,13:POKE 633,13
63997 POKE 198,3:END
63999 CLOSE1
```

*Commentary*

Line 63991: The file which this program will read is a listing of any other program which was stored on tape using the following command: OPEN 1,1,2 'TEST':CMD1:LIST. The CMD command means that anything that would normally be output to the screen is actually sent to the file number specified—in this case a file opened to the cassette recorder. The only distinctive thing about the file is the secondary address, which is 2, meaning that this is an output file which will have a special end of file marker printed at its conclusion. The LISTing of the program is sent, not to the screen, therefore, but to the cassette recorder, not in the same form that a program is normally stored but in ASCII format or character for character what you would see on the screen if the program were listed out. When the cassette recorder stops, without switching off the cassette recorder you must finish the file off by entering: PRINT#1: CLOSE1 which ensures that the final characters of the program are printed and the file properly closed.

Lines 63994–63995: Skipping over line 63992 for a moment, the merge program now begins to pick up the characters of the program listed onto tape, until it reaches a RETURN code, signifying the end of a line.

Line 63996: This strange line is actually the key to the program. Having picked up a line and printed it onto the screen, the program now prints, just underneath the line, the command GOTO 63992 and the cursor home control character. Following this, three RETURN codes are POKEd into the keyboard buffer, the area of memory which stores any keys which have been pressed but not yet acted upon, and the number three is POKEd into location 198, which records how many keys have yet to be acted upon. Having done this the program now ENDs. Or at least it would do, except that the 64 now believes that RETURN has been pressed three times and proceeds to react accordingly. The effect of this is to move the cursor down over the line picked up from tape and printed on the screen and over the GOTO 63992. The result is that the line is entered into the memory just as if you had placed the cursor over it and pressed RETURN. The program then begins again at 63992.

Line 63992: This line of mysterious POKE commands is there to overcome a basic problem—whenever a new line is entered to a program, all existing files are CLOSEd. When the first line of the new program is entered, the file to the cassette recorder is closed and any instructions to GET from it will result in an error. The file cannot be re-opened with a Basic command because we are already past the file header, which would tell the 64 that a file has been found on the tape. What the POKEs do is cheat a little and tell the 64 that file number one, with a secondary address of zero, is open to the cassette recorder. The second line can now be picked up, and so on ad infinitum.

Eventually the program encounters the end of file marker which means that the listing is complete and then the second part of line 63994 (IF ST THEN 63999) detects this and jumps to the last line. Normally the program will stop with the error message OUT OF DATA. This means the merge has been a success.

In actual use this program is by no means fast. It alternates between the blank screen of tape-loading and flashes of lines at the top of the screen. It needs a good tape, since a tape with dropouts may well crash the whole thing. But given a little care this is a program you will come back to time and time again—try it and see.

## 2.2 DELETE

When developing programs which use similar modules to programs you have previously entered, a useful ability is to be able to load the original program and delete only those parts that are not needed for the new application. This 12 line routine will allow you to do just that.

The routine is based upon the extremely clear and simple way in which program listings are set out in the memory of Commodore computers. Each program line in the memory begins with two link bytes of memory which specify the start address in memory of the next line. This is followed by two bytes which record the actual line number. What this routine does is to scan along the line numbers between a start number specified by the user and a finish number also user specified. When the address of the final line to be deleted is found, the program simply sets the next line address in the first line to be deleted to point to the line after the last line to be deleted. The effect of this is to produce a single line stretching from the beginning of the first line to be deleted to the end of the last. Deletion can now be accomplished by merely deleting the first line—all the others go with it.

MODULE 2.2.1

```
63700 INPUT "FIRSTLINE TO BE DELETED:";D
1
63710 INPUT "LAST LINE TO BE DELETED:";D
2
```

```
63715 DEF FNDH(X)=PEEK(X)+256*PEEK(X+1)
63716 DEF FNH1(X)=X AND 255
63717 DEF FNH2(X)=INT(X/256)
63720 LA=2049
63730 LN=FNDH(LA+2):IF LN<D1 THEN LA=FND
H(LA):GOTO 63730
63740 DSTART=LA
63750 LN=FNDH(LA+2):IF FNDH(LA)=0 THEN 6
3760
63755 IF LN<=D2 THEN LA=FNDH(LA):GOTO 63
750
63760 POKE DSTART,FNH1(LA):POKE DSTART+1
,FNH2(LA)
63770 POKE DSTART+4,143:FOR I=5 TO 10:PO
KE DSTART+I,33:NEXT
```

*Commentary*

Line 63715: This function, which can be useful in a variety of contexts, converts a two byte number of the kind that most computers work with, into a normal decimal number in the range 0-65535. The two byte number effectively has a base of 256, that is to say that it is composed of up to 255 units and a second digit of up to 255*256, in the same way that 99 is 9 units and 9 times 10. Just to confuse you, however, the digits are stored back to front, with the higher value byte coming second.

Lines 63716–63717: These two user-defined functions do the opposite job of transforming a decimal number into a two byte form.

Line 63720: LA is set equal to the start address of the first line of the program.

Line 63730: LN is set equal to the value of the third and fourth bytes of the line—the line number—and if this is less than the value of the first line to be deleted then LA uses the two link bytes to jump to the start address of the next line. The process is repeated until a line number is found which is equal to or greater than the first line number to be deleted.

Line 63740: The start address of the first line to be deleted is stored in the variable DSTART.

Line 63750: Using the FNDH function, the variable LA shoots up the memory from line start to line start, and with each jump the variable LN is set equal to the line number found there. If FNDH finds a memory location with zero in it, where there should be a pair of link bytes, it has reached the end of the program.

Line 63755: Each time a new line number is found, it is compared with the number of the last line to be deleted. If the last line has not been reached, the next jump is made.

Line 63760: If the program has reached this point, it has found the last line to be deleted and into the two link bytes at DSTART it POKEs the address of the line after the last line to be deleted.

Line 63770: The first character of the new single line block to be deleted is made into a REM statment and a series of exclamation marks are POKEd in after the REM to mark the line to delete.

The routine has now finished its job and all that remains is to enter the number of the line marked and press RETURN—the whole block, from a few lines to a complete program, will disappear.

In practice, this routine is best used with the merge routine in the previous section of this chapter (load the merge routine and either add this one or merge it), since this will allow the routine to be added to existing programs from which you wish to extract some lines while discarding others. It takes only moments to run and can save a lot of key pounding!

## 2.3 RENUMBER

One thing that everyone wants to do is to have neatly numbered programs —somehow it makes all the difference between something that looks professional and something which appears downright sloppy. Using the relatively short program presented here, you can renumber to your heart's content, though it isn't what you'd call fast and it does impose a simple limitation on the range of program lines.

The program will renumber any program, including GOTOs, GOSUBs, ON...GOTOs and GOSUBs and line numbers following IF...THEN. What it will not do is relocate the program in the memory, so it cannot add digits to a GOTO (etc.) or subtract digits, for to do so means moving the whole of the program that follows the altered address. It is not that that is impossible, or even particularly difficult, it is simply that to be at all practical it must be done in machine code, which is outside the scope of this book.

It is because of that limitation that all the programs in this book, which were renumbered using this routine (where line numbers are irregular it is because changes were made late in the process), begin at 11000, thus ensuring that all line numbers have five digits.

The way in which the renumbered program is structured can be controlled by the use of formatting lines within the program to be renumbered. Look closely at the programs in this book and you will see that the modules almost invariably start with a REM statement, and that the first character after the REM is a '#' symbol. The renumber program is so designed that it starts renumbering at 11000, and continues in steps of 10 until it comes across such a line, then it increments the line number to the next 1000 up.

Not only does this make for easily readable programs, it means that you can control the structure of the program to be renumbered. Say you have an existing program from which you want to use three or four modules, but the present line numbers do not conform to the structure you want for the new program—not enough space between the modules, say, to merge in something else you have on tape. By inserting two REM statements, whose first character is a '#' and then renumbering, you automatically open a gap of 2000 where the REM statements are located—hardly complex!

MODULE 2.3.1

```
63000 CLR:DIM ZZ(500,1):LA=2049:PP=LA
63010 DEF FNDH(X)=PEEK(X)+256*PEEK(X+1)
63013 DEF FNH1(X)=X AND 255
63016 DEF FNH2(X)=INT(X/256)
63050 IF PP<>FNDH(LA) THEN 63060
63053 LA=FNDH(LA)::NL=FNDH(LA+2):IF NL=6
3000 THEN GOTO 63500
63058 IF PEEK(LA+5)=143 THEN PP=FNDH(LA)
:GOTO 63053
63059 PP=PP+4
63060 IF PEEK (PP)<>167 THEN 63070
63062 S=0:IF PEEK(PP+1)=32 THEN S=1
63064 IF PEEK (PP+1+S)<48 OR PEEK (PP+1+
S)>57 THEN 63200
63065 GOTO 63076
63070 IF PEEK (PP)<>137 AND PEEK(PP)<>14
1 THEN 63200
63076 LET S=0:IF PEEK(PP+1)=32 THEN S=1
63080 GG$="":FOR I=1+S TO 5+S:IF PEEK(PP
+I)<48 OR PEEK(PP+I)>57 THEN GOTO 63140
63085 LET GG$=GG$+CHR$(PEEK(PP+I)):NEXT
63090 GG=VAL(GG$):L1=2049:L2=FNDH(2051):
LL=11000
63093 IF L2=63000 THEN PRINT "UNDEFINED
LINE AT LINE";NL:STOP
63095 IF L2=GG THEN 63100
```

```
63097 L1=FNDH(L1):L2=FNDH(L1+2):LL=LL+10
63098 IF PEEK(L1+4)=143 AND PEEK(L1+5)=3
5 THEN LL=1000*INT((LL+1000)/1000)
63099 GOTO 63093
63100 LET ZZ(ZI,0)=LL:LET ZZ(ZI,1)=PP+S:
ZI=ZI+1
63110 IF PEEK(PP+S+6)=44 THEN PP=PP+6+S:
GOTO 63076
63135 GOTO 63200
63140 PRINT "NON-STANDARD COMMAND AT LIN
E ";NL:STOP
63200 PP=PP+1:GOTO 63050
63500 LA=2049:LL=10000
63503 IF PEEK(LA+4)=143 AND PEEK(LA+5)=3
5 THEN LL=1000*INT((LL+1000)/1000)
63505 IF FNDH(LA+2)=63000 THEN 63600
63510 POKE LA+2,FNH1(LL):POKE LA+3,FNH2(
LL)
63520 LET LL=LL+10:LET LA=FNDH(LA):GOTO
63503
63600 IF ZI=0 THEN STOP
63605 FOR I=0 TO ZI-1
63610 FOR J=1 TO 5:POKE ZZ(I,1)+J,ASC(MI
D$(STR$(ZZ(I,0)),J+1)):NEXT J
63620 NEXT I
63650 STOP
```

*Commentary*

Line 63000: The array ZZ will be used to record the addresses of GOTOs etc. that need to renumbered and the new number they are to be given.

Lines 63010–63016: The same functions as in the block delete routine. If you merge the two together you will need to delete this set.

Lines 63050–63059: PP is a pointer that scans the memory for GOTOs etc. It starts at the beginning of the program area at 2049. Every time it reaches the start of a new line the line address variable (LA) is incremented. The line number of the current line is stored in LN and the program stops working at line 63000—the start of this routine. PP now jumps to the first character of the line.

Lines 63060–63065: These lines check when THEN is found in the memory, to see if it is followed, either immediately, or after a space, by a number. The variable S simply records whether a space is present. If THEN is not followed by a number, PP moves on.

Lines 63070–63085: If the code for GOSUB or GOTO is found the program checks to see whether a space follows or not. GG$ is constructed out of the digits of the line destination. Less than five digits produces an error message at line 63140.

Lines 63090–63099: GG is set equal to the GOTO or GOSUB destination. The routine now scans up the line numbers in the program from the start, looking for the destination. For each line that is examined, the variable LL is incremented by 10, starting at 11000, and thus records what the line number will be once the program is renumbered—the line cannot be renumbered at this point since there may be another GOTO pointing to it. When REM # is encountered LL increments to the next 1000 upwards.

Line 63100: At this point the correct line number has been found, so the address of the GOTO is stored in the array ZZ along with its future line number (LL).

Line 63110: If the 6th character after the GOTO or GOSUB is a comma, it is assumed that this is an ON...GOTO—GOSUB and PP is moved on and the new line destination picked up by an earlier part of the routine.

Line 63200: The process continues, with PP moving up the memory until Line 63000 is encountered.

Lines 63500–63520: These lines start at the beginning of the program and renumber the lines only (remembering REM #).

Lines 63600–63650: All that remains is to take the addresses of all the GOTO—GOSUB destinations out of ZZ and to POKE the new destinations into the five bytes following each address—they have already been calculated.

Though this program does not compare in speed or flexibility with a good machine code utility, it does do the job, as the programs in this book illustrate. If you do not own a machine code renumber routine then I predict that you will come back to this routine more often than almost any you possess.

When merged with the two previous utilities, with all three isolated from each other by STOP statements, you will have built for yourself a powerful three function tool which will make your programming more pleasant and your programs more presentable.

# CHAPTER 3
# The Colourful 64

The Commodore 64 provides an almost bewildering array of graphics capabilities. The shapes and colours that it can display are enough to cover almost any imaginable need and certainly enough to keep the amateur artist occupied for a lifetime. In this chapter you will find four graphics programs which will allow you to explore the world of the graphics character set, user-defined characters, sprites and bit-mapped graphics. A mere four programs are by no means the last word in what the 64 can achieve so the programs are designed as tools, whose object is to allow you to feed into your later programs all the colourful features that will lift them out of the ordinary run.

## 3.1 ARTIST

Few home micros have a graphics character set as useful as that of Commodore machines. Using the combinations of characters available from the keyboard it is very difficult to think of anything that could not be drawn in some shape or fashion. This is extremely useful when livening up the output of the most mundane programs, especially when combined with the 64's excellent colour capabilities.

One limitation to all of this is in the creative process of actually developing graphics displays. Of course, this can be done purely with print statements in a program, but getting the print statements exactly right, with a variety of colour commands, reverse commands and so forth, with each line having to be defined separately can be an extremely tedious process. What is really needed is a way of using the screen rather like an easel, painting on graphics characters in a variety of colours, erasing, changing at will and then, for the sake of posterity or at least for the sake of other programs which could use the design created, saving the design onto tape. All of this the current program sets out to do.

Along the way you will pick up a fair amount of information about how to manipulate the screen and colour memory, together with useful memory locations for controlling such characteristics as print colour.

**Artist: Table of Variables**

| | |
|---|---|
| CC | The current cursor position |
| CO(3) | Co-ordinates of two corners of design to be saved |

| | |
|---|---|
| CT | Temporary storage of cursor position |
| CU | Current cursor colour |
| D1$ | Values of characters in design to be saved |
| D2$ | Colour values of characters to be saved |
| D3$,D4$ | Temporary copies of D1$ and D2$ |
| MODE | Defines which of a variety of colour characteristics is being addressed |
| PC | The colour of the character in position PP |
| PP | The original contents of the current cursor location |
| PT | Location in memory of the two corners held in array CO |

MODULE 3.1.1

```
11000 REM#******************************
11010 REM VARIABLES
11020 REM*******************************
11030 R$=CHR$(13)
```

Hardly fair to call this a module, but its presence does mean that if you decide to develop the program further, there is a proper area set aside for the necessary variables. The string actually defined is a standard-data file separator.

MODULE 3.1.2

```
12000 REM#******************************
12010 REM CURSOR,MOVE,PRINT
12020 REM*******************************
12030 PRINT "⊐";
12040 POKE 650,255:GET A$
12050 CC=PEEK(211)+PEEK(210)*256+PEEK(20
9):PP=PEEK(CC):PC=PEEK(CC+54272)
12060 POKE CC,42:POKE CC+54272,CU:FOR I=
1TO15:NEXT:POKE CC,PP:POKE CC+54272,PC
12070 IF A$="" THEN 12040
12080 IF CC>1983 AND A$="Q" THEN 12040
12090 IF CC=2023 AND A$="▌" THEN 12040
12100 IF CC=1024 AND A$="█" THEN 12040
12110 IF CC<1064 AND A$="⌉" THEN 12040
12120 IF A$="⌉" OR A$="Q" OR A$="▌" OR A
$="█" THEN PRINT A$;:GOTO 12040
12130 IF A$=CHR$(133) THEN MODE=1
12140 IF A$=CHR$(137) THEN MODE=2
12150 IF A$=CHR$(134) THEN MODE=3
12155 IF A$=CHR$(138) THEN 12030
```

```
12160 IF (MODE=1 OR MODE=2 OR MODE=3) AN
D A$="←" THEN MODE=MODE+.5:GOTO 12040
12170 IF A$=CHR$(135) THEN INV=(INV=0)
12180 IF A$=CHR$(139) THEN MODE=6
12190 IF A$=CHR$(136) THEN MODE=7
12200 IF A$=CHR$(140) THEN MODE=8
12210 IF MODE=1 AND A$>="1" AND A$<="8"
THEN POKE CC+54272,VAL(A$)-1:GOTO 12040
12220 IFMODE=1.5ANDA$>="1" AND A$<="8" T
HEN POKE CC+54272,8+VAL(A$)-1:GOTO 12040
12230 IF MODE=2 AND A$>="1" AND A$<="8"
THEN POKE 53281,VAL(A$)-1:GOTO 12040
12240 IF MODE=2.5 AND A$>="1" AND A$<="8
" THEN POKE 53281,8+VAL(A$)-1:GOTO 12040
12250 IF MODE=3 AND A$>="1" AND A$<="8"
THEN POKE 646,VAL(A$)-1:GOTO 12040
12260 IF MODE=3.5 AND A$>="1" AND A$<="8
" THEN POKE 646,8+VAL(A$)-1:GOTO 12040
12270 IF MODE<>6 OR (A$<>"R" AND A$<>"D"
 AND A$<>"S") THEN 12320
12280 IF A$="R" THEN GOSUB 13000
12290 IF A$="D" THEN GOSUB 13060
12300 IF A$="S" THEN GOSUB 14000
12310 CC=CT:GOTO 12040
12320 IF MODE=8 AND A$>="1" AND A$<="8"
THEN CU=VAL(A$)-1:GOTO 12040
12330 IF INV=-1 THEN PRINT "▨";
12340 PRINT A$;"▓";
12350 GOTO 12040
```

This module is really all that is needed to turn your screen into a graphics easel. Its purpose is to allow you to move a flashing cursor around the screen, printing characters, changing them, erasing them, changing colours for foreground and background of characters......

*Commentary*

Lines 12040–12070: This routine provides a flashing cursor under user control.

Line 12040: This POKE sets the repeat characteristic so that a key, once held down, will continue printing the same character. The second part of the line receives any single character input from the keyboard.

Line 12050: CC is set equal to the address in the memory of the current print position. PEEK(211) gives the position along the line (0-39) while PEEK(210)*256 + PEEK(209) gives the memory address of the beginning of the line. PP is set equal to the screen code of whatever is currently occupying the position where the cursor is about to flash. PC is the colour of the character in that position.

Line 12060: An asterisk, screen code 42, is now POKEd into the position where the cursor is meant to flash and the current cursor colour CU, is POKEd into colour memory at the corresponding position. A short timing loop keeps the asterisk on the screen for a moment, then the orginal character (code PP) and the original colour (PC) are rePOKEd into the memory.

Line 12070: If no key has been depressed then the cycle is repeated.

Lines 12080–12110: These lines check that if cursor move arrows are input, the cursor does not attempt to move off the screen.

Line 12120: If a cursor control is input and passes through the tests in the four lines above it is immediately printed and the program returns to the flashing cursor routine.

Lines 12130–12200: Using the function keys on the right of the keyboard as inputs, these lines allow the user to specify different modes which permit different colour characteristics to be set.

Line 12130: Pressing key f1 puts the program into MODE 1. In this mode, pressing any of the keys 1 to 8 will redefine the colour of any character over which the cursor is currently placed. The colour will be that indicated on the front of the key.

Line 12140: Pressing f2 allows the same procedure to redefine the screen background colour.

Line 12150: Pressing f3 allows the resetting of the print colour by the same procedure.

Line 12160: Since there are in fact 16 colours available, input of the left arrow at the top left hand corner of the keyboard while in MODEs 1,2 or 3, redefines the mode so that entry of keys 1-8 will provide the colour that would normally be obtained by pressing that key together with Commodore logo key. The characteristic redefined will be the same as that referred to by the main mode number.

Line 12170: On pressing f5 the inverse characteristic is set or reset—thus allowing inverse characters to be printed.

Line 12180: On pressing f6 the program will allow the saving of the design created. The correct procedure will be explained in detail later.

Line 12190: Pressing f7 does nothing at all except to redefine into a non-effective mode. This allows the user to print the numbers 1-8 on the screen rather than redefine a colour characteristic.

Line 12200: Pressing f8 allows the user to change the cursor colour to any of the first eight colours. This is useful if the screen colour has been re-defined in such a way that the cursor is no longer clearly visible.

Lines 12210–12220: If MODE is 1 or 1.5 then the colour input is POKEd into the colour memory for the current square.

Lines 12230–12240: If MODE is 2 or 2.5 the new colour code is POKEd into location 53281, which sets the screen background colour.

Lines 12250–12260: If MODE is 3 or 3.5 then the new colour code is POKEd into location 646, which dictates the current print colour.

Lines 12270–12310: When in MODE 6 this routine relates to the saving of either small or large designs. Input of R allows the definition of a rectangle of screen to be saved. D saves a small-scale design. S saves the whole screen to tape.

Line 12310: CT is used to save the current cursor position, which may be altered during the SAVE routine.

Line 12320: If MODE is 8 then the new colour code is stored in the variable CU.

Line 12330: If the inverse characteristic is set (INV=-1) the RVS ON control character is printed, thus inverting the next character to be printed.

Line 12340: If the program has reached this point then whatever character was input is printed on the screen and the reverse off control character is printed following it.

## *Testing Module 3.1.2*

After entering this module you should be able to create designs on the screen at will, using the whole character set available from the keyboard. All the colour redefinition commands should be available but you will not yet be able to save any design you create.

## MODULE 3.1.3

```
13000 REM********************************
13010 REM SAVE DESIGN
13020 REM********************************
13030 GET T$:IF T$="" THEN 13030
13040 IF T$<>"1" AND T$<>"2"THEN RETURN
13050 CO(VAL(T$)*2-2)=PEEK(211):CO(VAL(T
$)*2-1)=INT((CC-1024)/40):RETURN
13060 REM********************************
13070 CT=CC:POKE 646,CU
13080 IF CO(0)<=CO(2) AND CO(1)<=CO(3) T
HEN 13110
13090 PRINT "■RECTANGLE IMPROPERLY DEFIN
ED.";:FOR I=1 TO 1000:NEXT
13100 PRINT "■
          ";:RETURN
13110 IF (CO(2)-CO(0)-1)*(CO(3)-CO(1)-1)
<251 THEN 13140.
13120 PRINT "■DESIGN TOO LARGE.";:FOR I=
1 TO 1000:NEXT
13130 PRINT "■
          ";
13140 FOR I=1 TO 2:PT=1024+40*CO(I*2-1)+
CO(I*2-2):TC(I)=PT:TC(I+2)=PEEK(PT)
13150 POKE PT,42:POKE PT+54272,CU:NEXT
13160 INPUT "■THESE POINTS O.K. (Y/N):";
Q$:IF Q$="Y" THEN 13170
13162 PRINT "■
     ":FOR I=1 TO 2:POKE TC(I),TC(I+2):NEXT
13164 RETURN
13170 D1$="":D2$="":FOR I=CO(1)+1 TO CO(
3)-1:FOR J=CO(0)+1 TO CO(2)-1
13180 D1$=D1$+CHR$(PEEK(1024+40*I+J)):PO
KE 1024+40*I+J,42
13190 D2$=D2$+CHR$(PEEK(55296+40*I+J)):P
OKE 55296+40*I+J,0:NEXT J,I:PRINT "▐";
13200 D3$=D1$:D4$=D2$:FOR I=CO(1)+1 TO C
O(3)-1:FOR J=CO(0)+1 TO CO(2)-1
```

```
13210 POKE 1024+40*I+J,ASC(LEFT$(D3$,1))
:D3$=RIGHT$(D3$,LEN(D3$)-1)
13220 POKE 55296+40*I+J,ASC(LEFT$(D4$,1)
):D4$=RIGHT$(D4$,LEN(D4$)-1):NEXT J,I
13230 PRINT "▒THIS IS WHAT IS BEING SAVE
D.":FOR I=1 TO 1000:NEXT
13240 INPUT "▒POSITION TAPE CORRECTLY, T
HEN RETURN:";Q$
13250 PRINT "▒
             ▒";
13260 OPEN 1,1,1,"ARTIST":FOR I=0 TO 3:P
RINT#1,CO(I):NEXT:PRINT#1,LEN(D1$)
13270 FOR I=1 TO LEN(D1$):PRINT#1,ASC(MI
D$(D1$,I,1)) R$ ASC(MID$(D2$,I,1)):NEXT
13280 CLOSE1:RETURN
```

The purpose of this module is to allow a small design to be defined on the screen and then saved economically.

*Commentary*

Lines 13030–13050: If, in the previous module, MODE 6 is set and then R pressed these three lines accept the input of a further character which must be a 1 or a 2. If 1 is pressed, the current cursor square is defined as the top left-hand corner of a rectangle to be saved, 2 defines the bottom right-hand corner. These two squares are actually outside the design to be saved, they define an outside border to what is to be saved. The positions in memory of the two design corners are stored in the array CO. Note that this array has not been declared since it has less than 10 elements—simply inputting a value to it will set it up satisfactorily. CO(0) or CO(2) is set to the position of the cursor in the row, as indicated by PEEK(211). CO(1) or CO(3) is set to the current screen line number +(actual memory position-screen start)/40.

Lines 13060–13280: These lines allow the saving of the rectangle previously defined.

Line 13070: The print colour is set temporarily to the colour of the cursor.

Line 13080: A check is made that a valid rectangle has been defined ie that it has length and width). If not, an error message is printed.

Line 13110: Data for the design will be temporarily stored in a string so a check is made that the string will not be too long. An error message is printed if the string is likely to be too long.

Lines 13140–13160: Using the co-ordinates contained in the array CO, the corners of the rectangle are rePOKEd onto the screen and the user is asked to confirm the correctness of the rectangle to be saved.

Line 13170: The two strings which will be used are initialised. The two loops combine to mean that J characters (the width of the design) will be read from the screen for I lines (the height of the design).

Lines 13180–13190: On the basis of the addresses provided by the loops, the screen and colour memory are PEEKed, and the values added to the storage string in the form of characters of that code value. The two strings thus formed would make no sense printed out, they are merely a simple way of temporarily storing a series of values without having to set complicated pointers to a position in an array. After this is done, an asterisk is POKEd into the screen location and its colour characteristic set to black, making the processing of the design visible.

Line 13200: A copy of D1$ and D2$ is taken, then two more loops are used to POKE back onto the screen the characters which have been stored, together with their colour characteristics. Each time a character is POKEd back onto the screen the two strings are stripped of their left-hand character, so that it is always the first character of the string which is used. It is this stripping process that necessitates the creation of a temporary copy of the two original strings. The sole purpose of these two loops is really to re-assure the user that the design is going to be saved correctly.

Lines 13240–13280: The design is saved onto tape.

Line 13260: The values in the array CO are saved, together with the length of D1$ (which is also the length of D2$).

Line 13270: A loop equal to the length of D1$, saves the values of the characters of both strings (ie the values taken from screen and colour memory). Unfortunately the two strings themselves cannot be saved onto tape since they may contain non-printing characters which the 64 is not capable of saving in string form.

*Testing Module 3.1.3*

You should now be able to save a design onto tape. If the redisplay of the design is satisfactory, it is likely that the saving is being done correctly, but this can only be fully tested if you subsequently enter at least the relevant module of the program Words, which is intended to make use of the designs so created.

MODULE 3.1.4

```
14000 REM#******************************
14010 REM SAVE SCREEN
14020 REM*******************************
14030 PRINT "▒POSITION TAPE CORRECTLY, T
HEN RETURN:";Q$
14040 PRINT "▒
                  ";:OPEN 1,1,1,"SCREEN"
14050 FORI=0TO999:PRINT#1,PEEK(1024+I):P
RINT#1,PEEK(55296+I):NEXT:CLOSE1:RETURN
```

If, during the execution of Module 2, MODE 6 is set and S then pressed, this module will ensure that the whole of the contents of the screen are saved to tape. This is done by the uncomplicated method of PEEKing the contents of the memory from 1024 to 2023 (the screen memory) plus the equivalent colour memory locations and saving the values. A later program can read the values from tape and POKE them back into the same locations.

*Summary*

This program is capable of providing a great deal of fun but its greatest contribution is the capacity that it gives you to design complex graphics with ease, editing them at will and simply calling them up for use in subsequent programs. You should also, based on the techniques employed here, have no difficulty with subsequent programs of your own which need to POKE the screen and colour memory.

***Going Further***

1) No provision is made to save the screen background colour—it would be a simple matter to add this.
2) Why not add a display on the bottom line of the screen to show which mode is currently set.

**Artist: Table of one-key commands**

| | |
|---|---|
| f1 | Allows redefinition of colour of character under cursor |
| f2 | Allows redefinition of screen colour |
| f3 | Allows change of print colour |

| | |
|---|---|
| f4 | Erases current design |
| f5 | Sets or resets RVS |
| f6 | SAVE mode |
| f7 | Dummy mode |
| f8 | Allows change of cursor colour |
| → | Allows entry of second colour set in modes 1-3 |

SAVE mode:

| | |
|---|---|
| R | then 1 or 2 defines corner of rectangle to be saved |
| D | saves small scale design |
| S | saves whole screen |

## 3.2 CHARACTERS

No matter how good the character set provided by a home micro, there is bound to come a time when the character you want is not available. It may be that you want to print in another language and use characters with accents, or in abstruse mathematical symbols, or it may be that you need something rather special to put the finishing touches to your latest game. Whatever it is that you need, the 64 is waiting to meet that need with its user-defined character capability.

When the 64 is started up, all its potential characters are stored in its Read Only Memory, in a section beginning at address 54248. Each character takes the form of eight bytes of memory and the 8*8 grid of dots making up each character is represented by the individual bits of the eight bytes set aside for each character. For instance, if the eight bytes of memory for a particular character were 128, 64, 32, 16, 8, 4, 2 and 1 then, in binary notation they would be 10000000, 01000000, 00100000, 00010000, 00001000, 00000100, 00000010 and 00000001. Now place those values in a grid:

```
10000000
01000000
00100000
00010000
00001000
00000100
00000010
00000001
```

The bits which are set (or on) define a character (in this case a diagonal line) with each set bit being translated into one pixel on the screen.

That's all very well but since the character data is stored in Read Only Memory, it cannot be changed—it is permanently set when the ROM chip is manufactured. Fortunately the 64 provides a neat way around the problem but to understand it we first have to examine the method by which the video display is generated.

All the tasks relating to the video screen on the 64 are handled by a separate chip, the 6567 Video Interface Chip (or VIC II chip). This workhorse handles both the screen itself and the characters which are to be placed onto the screen, defining an area of memory in which the screen information will be stored and another it will draw upon for character data.

Contrary to what you might expect, the VIC II does not draw its character data from the ROM at address 53248. The reason for this is that the VIC II is only capable of perceiving 16K of memory at one time, so with the screen memory in its normal position at 1024–2023 in the memory, character data must be drawn from somewhere between 0 and 16383 in the memory. To achieve this, the operating system cheats a little and makes the VIC II believe that there is copy of the character set data located at 4096–6143. Whenever the VIC II looks at that memory area it detects the character set data, despite the fact that in actual fact that area of memory will probably be filled with a Basic program.

This may seem a little abstruse but it is of vital importance since it means that, rather than looking for its character data in the ROM, which cannot be altered, the VIC II looks for the data in Random Access Memory (RAM), that is to say memory that the user can get at and alter. Of course it's not quite that simple. We have already noted that when the VIC II looks at the memory area from 4096 onwards, it is not actually those addresses that it sees but an image of the character data in the ROM. Fortunately, this is a feature of only two blocks of memory within the 16K block, 4096–6143 and 6144–8193. If the VIC II is instructed to seek its character data from any of the other 2K blocks within the total 16K, then it will not see the ROM image but will take the data which is actually in memory and treat it as if it were the character set.

The question now becomes, which block shall we specify? The first one available is 2048–4095 but this has the slight drawback that it is where the Basic program starts and POKEing it with new character data will crash the program. We could use the blocks at 8192, 10240,12288 or 14336, but unfortunately this would mean that we would have to limit the area available to the Basic program quite drastically since otherwise there would be a danger that a large program would overwrite the area used for characters. The solution adopted here is to move the whole area that the VIC II chip addresses further up into the memory.

You will remember that the VIC II is capable of seeing a 16K chunk of memory at one time, however it is not fussy which 16K block it sees. There are four such blocks, beginning at 0,16384,32768 and 49152. Moving to the block at 49152, while providing the maximum memory area for Basic, presents the problem that that is where the ROM is, so we shall ask the VIC II to address the 16K block starting at address 32768. Having done that, all that remains is to specify where in that block the character data will be taken from and where the screen data will be located. There will now be 30K of memory available for a Basic program (2048–32767) and the potential for a user-defined character set in the RAM above 32768.

No doubt all this seems inordinately complicated. In fact it is simply a matter, given the 64's flexible memory structure, of a few POKEs and the job is done. On the basis of the changes wrought by those POKEs, the program which follows will allow you to completely redefine all or part of the 64's character set and to store your new character set so that it can picked up and used by other programs.

**Characters: Table of Variables**

| | |
|---|---|
| A$ | Single key command obtained by use of GET |
| C1 | Original colour of screen at location CC |
| CC | Current position of flashing cursor in the screen memory |
| CH | Number of current character in character set |
| CP | Pointer to location in memory of character CP |
| MM | Value input to change CP |
| P1 | Row of cursor position on screen |
| P2 | Column of cursor position on screen |
| PP | Original contents of screen at location CC |
| TT((7,7) | Used to allow the manipulation of data for current character |

MODULE 3.2.1

```
11000 REM#******************************
11010 REM RE-ARRANGE MEMORY
11020 REM*******************************
11030 POKE 53281,6:PRINT CHR$(142)
11040 POKE 52,128:POKE56,128
11050 POKE 56334,PEEK(56334)AND 254
11060 POKE 1,PEEK(1) AND 251
11070 FOR I=0 TO 2047:POKE 32768+I,PEEK(
53248+I):NEXT
11080 POKE 1,PEEK(1)OR4
11090 POKE 56334,PEEK(56334)OR1
11100 POKE56578,PEEK(56578)OR3
11110 POKE 56576,(PEEK(56576)AND252)OR1
11120 POKE 648,136
11130 POKE 53272,32
```

The purpose of this module is to accomplish all the changes to memory structure specified above and to copy an initial character set into the RAM area specified.

*Commentary*

Line 11030: This sets the machine in capital characters mode, since only the first of the two available character sets on the 64 is going to be usable once the VIC II stops looking at the ROM image.

Line 11040: These two POKEs set the top of the area available for the Basic program in such a way that any program entered will not be capable of interfering with the area of memory set aside for characters. 30K of memory is available under this setting.

Lines 11050–11060: These two POKEs switch off the keyboard scan so that no interrupts can disturb the next section of the program and then make the ROM character set visible to the program by switching off the normal process of input and output. During the loop that follows, the only way to stop the program will be to switch off the machine.

Line 11070: This copies the character set from ROM to the memory area beginning at 32768—this involves the transfer of 2K bytes.

Lines 11080–11090: These switch the normal input—output regime back on and restore the normal interrupts.

Lines 11100–11110: These two POKEs first ready the VIC II chip for a change of memory block and then specify Block 1 (32768–49151).

Line 11120: This location is outside the VIC II chip and is the operating system's guide to where screen memory is to be located—in this case starting at 256*136 = 34816.

Line 11130: The location at which the VIC II expects to find both screen and character data within its 16K block is dictated by the contents of address 54272—the upper four bits for the screen, the lower four for the character set. This POKE sets the upper four bits to 0010, which signifies the 1K block starting at 32768 + 2048 for the screen, and the lower four bits to 0000, specifying that character data will be taken from 32768 + 0. To arrive at other possible locations in the 16K block, the formula to dictate the POKE would be ((SCREEN START-BLOCK START)/1024)*16 + (CHARACTERS-BLOCK START)/2048). The screen can only start at a 1K boundary within the block and characters at only a 2K boundary. Note that we could have left the screen at 1024 and the character memory at 4096, except that in this 16K block of memory as in the block starting at address zero, the VIC II sees a ROM image at 4096 onwards.

*Testing Module 3.2.1*

The test for this module is quite simple. Run it and the machine will lock up for a while—there is nothing you can do to interrupt it. When the READY comes up on the screen, nothing should have changed—which shows that the module has worked! If the module has not worked, then the screen will be filled with garbage.

MODULE 3.2.2

```
12000 REM********************************
12010 REM PRINT GRID
12020 REM********************************
12030 CH=0:DIM TT%(7,7)
12040 PRINT "▒▒";:FOR I=1 TO 8:PRINT "▒█
█████████ ":NEXT
12050 PRINT "▒              "
12060 CP=32768+CH*8
12070 PRINT "▒▒";:FOR I=CP TO CP+7:FOR J
=7 TO 0 STEP -1
12080 IF (PEEK(I) AND 2↑J)=2↑J THEN PRIN
T "▒ ";
12090 IF (PEEK(I) AND 2↑J)=0 THEN PRINT
"█ ";
12100 NEXT J:PRINT:NEXT I
12110 PRINT "▒▒▒CHARACTER NUMBER:";CH
12120 INPUT "▒NUMBER TO MOVE POINTER:(0=
REDEF.):";MM:CH=CH+MM
12130 IF CH<0 THEN CH=0
12140 IF CH>255 THEN CH=255
12150 IF MM=0 THEN 13000
12160 GOTO 12040
```

There are many ways in which new characters can be entered into the character memory. You can, if you wish, draw them on an 8*8 grid, translate the lines of dots into binary and then into decimal, enter the figures as data statements and then POKE them into the memory. Fortunately, it is much easier to get the 64 to do the work by drawing the current character grid on the screen and then allowing it to be easily manipulated. This module draws the character grid, the next one allows manipulation.

*Commentary*

Lines 12040–12050: An 8*8 box is outlined in the top-left hand corner of the screen.

Line 12060: The position of the current character's data is calculated.

Lines 12070–12100: An enlarged version of the current character is printed in the box outlined. Note the use of AND here to get at the contents of individual bits within the eight bytes of memory for the character. All that AND does in this context is to compare two binary numbers and produce a third which has only those bits set which were also set in both the two numbers originally compared. Thus if 193 is ANDed with 129 (Binary:11000001 AND 10000001) the result is 129 since bit 6 in the first number is not set in the second as well. ANDing the value of a byte with 2↑(J) where J is from 0 to 7, will show whether bit J is set or not—remember that bits are numbered from 0-7 right to left.

Lines 12100–12160: The number of the character displayed is given and the user has the option to move the character pointer within the 255 characters. If zero is input, the program moves on to the next module.

*Testing Module 3.2.2*

Once again, the test is quite simple. If the module has been correctly entered, running the program will result (after a pause) in the printing of an enlarged version of '@' on the screen. You should also be able to page through the other characters.

MODULE 3.2.3

```
13000 REM#*****************************
13010 REM REDEFINE CHARACTER
13020 REM******************************
13030 PRINT "⌐
                   ⌐TT⌐":REM 40 SPACES
13040 PRINT "▆'I' TO INVERT",,,"▆'M' TO
MIRROR",,,"▆'R' TO RETURN"
13050 PRINT "▆'1' TO INK IN SQUARE":PRIN
T "▆'0' TO BLANK SQUARE"
13060 PRINT "▆'T' TO TURN",,,"▆'P' TO PL
ACE IN MEMORY"
13070 PRINT "▆'D' TO SAVE IN A DATA FILE
",,"▆'C' TO PICK UP SET FROM TAPE"
13080 PRINT "▆'N' TO NORMALISE MEMORY AN
D END"
```

```
13100 PRINT "█ CURSOR ARROWS TO MOVE"
13110 PRINT "█";
13120 GET A$
13130 CC=PEEK(211)+PEEK(210)*256+PEEK(20
9):PP=PEEK(CC):C1=55296+CC-34816
13140 C2=PEEK(C1)
13150 POKE CC,42:POKE C1,1
13160 FOR I=1 TO 15:NEXT:POKE CC,PP:POKE
 C1,C2:IF A$="" THEN 13120
13170 P1=INT((CC-34816)/40):P2=CC-(34816
+40*P1)
13180 IF (P1>0 AND A$="█") OR (P1<7 AND
A$="█") THEN PRINT A$;:GOTO 13120
13190 IF (P2>0 AND A$="█") OR (P2<7 AND
A$="█") THEN PRINT A$;:GOTO 13120
13200 IF A$="1" THEN PRINT "██ ";
13210 IF A$="0" THEN PRINT "██ █";
13220 IF A$<>"I" THEN 13280
13230 FOR I=0 TO 7:FOR J=0 TO 7
13240 IF PEEK(34816+I*40+J)=32 THEN 1326
0
13250 POKE 34816+40*I+J,32:POKE 55296+40
*I+J,182:GOTO 13270
13260 POKE 34816+I*40+J,160:POKE 55296+4
0*I+J,181
13270 NEXT J,I
13280 IF A$="R" THEN GOTO 12040
13290 IF A$<>"M" THEN 13370
13300 FOR I=0 TO 7:FOR J=0 TO7:TT%(I,J)=
0:NEXT J,I
13310 FOR I=0 TO 7:FOR J=0 TO 7
13320 IF PEEK(34816+40*I+J)=160 THEN TT%
(I,J)=1
13330 NEXT J,I
13340 PRINT"█";:FORI=0TO7:FOR J=7 TO 0ST
EP-1:IF TT%(I,J)=1 THEN PRINT "██ █";
13350 IF TT%(I,J)=0 THEN PRINT "█ ";
13360 NEXT J:PRINT:NEXT I
13370 IF A$<>"T" THEN 13450
13380 FOR I=0 TO 7:FOR J=0 TO 7:TT%(I,J)
=0:NEXT J,I
13390 FOR I=0 TO 7:FOR J=0 TO 7
13400 IF PEEK(34816+40*I+J)=160 THEN TT%
(7-J,7-I)=1
```

```
13410 NEXT J,I
13420 PRINT"▓";:FORI=0TO7:FOR J=7 TO 0ST
EP-1:IF TT%(I,J)=1 THEN PRINT "▓▓ █";
13430 IF TT%(I,J)=0 THEN PRINT "▓ ";
13440 NEXT J:PRINT:NEXT I
13450 IF A$<>"P" THEN 13510
13460 :FOR I=0 TO 7:TT%(0,I)=0:NEXT
13470 FOR I=0 TO 7:FOR J=0 TO 7
13480 IF PEEK(34816+40*I+J)=160 THEN TT%
(0,I)=TT%(0,I) OR 2↑(7-J)
13490 NEXT J,I
13500 FOR I=0 TO 7:POKE CP+I,TT%(0,I):NE
XT:GOTO 12040
13510 IF A$<>"D" THEN 13550
13520 OPEN 1,1,1,"CHARACTERS"
13530 FOR I=0 TO 2047:T%=PEEK(32768+I):P
RINT#1,T%:NEXT
13540 CLOSE 1
13550 IF A$<>"C" THEN 13590
13560 OPEN 1,1,0,"CHARACTERS"
13570 FOR I=0 TO 2047:INPUT#1,T:POKE 327
68+I,T:NEXT
13580 CLOSE1
13590 IF A$<>"N" THEN 13660
13600 POKE 52,160:POKE56,160:CLR
13610 POKE56578,PEEK(56578)OR3
13620 POKE 56576,(PEEK(56576)AND252)OR3
13630 POKE 53272,21
13640 POKE 648,4
13650 END
13660 GOTO 13120
```

This module performs a variety of functions to do with the manipulation of the character on the screen, allowing it to be redefined, placed back into memory and SAVEd to tape among other things.

*Commentary*

Lines 13030–13100: Brief instructions for the use of the module are printed on the screen.

Lines 13120–13160: This module holds no surprises. It is simply the cursor flash routine from the Artist program.

Line 13170: Cursor position:P1 is the row down from the top of the screen, P2 is the column across from the left.

Lines 13180–13190: Limits of cursor movement with the 8*8 square.

Lines 13200–13210: Pressing 1 inks in a green square, pressing 0 blots out a square.

Lines 13230–13270: These two loops scan across the square reversing the inked-in or blank elements, thus producing an inverse character.

Line 13280: Input of R returns to the previous module.

Lines 13290–13360: This routine produces a mirror image of whatever is the grid—ie the character is apparently seen from behind.

Line 13300: The array TT% is cleared.

Lines 13310–13330: The contents of the screen are transferred to the array. The screen cannot be manipulated directly since this might result in a square being transferred from the left to right and then read twice, producing nonsense.

Lines 13340–13360: Having transferred the contents of the grid to the array the information is now read back onto the screen but the horizontal element is reversed so that position 7 is placed into position zero.

Lines 13370–13440: The contents of the grid are turned 90 degrees anti-clockwise.

Lines 13420–13440: The contents of the array are put back onto the screen anti-clockwise—thus position 0,7 becomes position 0,0 and position 0,0 becomes position 7,0.

Lines 13450–13500: The redefined character is placed back into the character memory. It now becomes a permanent part of the user-defined set.

Line 13460: Since only eight bytes are required for each character, only eight bytes of the array, line zero, 0-7, need be cleared.

Lines 13470–13490: Each line of the array is scanned and when an inked-in square is detected, it is translated into a single bit in one of the eight bytes used to define the character. Having used AND to read individual bits, note the use of OR to manipulate individual bits. When two binary numbers are ORed, all the bits which are set in either (or both) are set in the resulting number. Thus to OR a number with 2↑(J), where J is from 0 to 7 means that bit J will be turned on, regardless of whether it was on or off before.

Line 13500: The eight bytes of the array are placed into the memory at the position previously occupied by the character which has been redefined.

Lines 13510–13540: The area of memory starting at 32768 is stored onto tape in the form of integer numbers.

Lines 13550–13560: A previously stored character set can be picked up from tape for further manipulation. **NB** This is also an example of how your new character set can be picked up by another program for subsequent use.

Lines 13590–13650: If the program is terminated, the memory must be reset to its original condition—unless you wish to go on using your new character set with another program you are going to load. Failure to reset the memory would mean that subsequent programs will be deprived of 8K of memory and forced to use the redefined character set.

Line 13600: Basic is reset to its full potential size.

Lines 13610–13620: The bank of memory addressed by the VIC II is reset to 3 (0-16383).

Lines 13630–13640: The screen is reset to start at 1024 (its normal position) and the character memory reset to 4096 onwards.

### *Testing Module 3.2.3*

Since this is a long module with a variety of functions, it is suggested that you test each function as it is entered. Note that if a particular function is faulty and you have entered changes to a line, there is no need to RUN the program from the start. Simply GOTO 12000 since the character set, which is above the Basic area, and the memory structure are undisturbed by the entry of new lines. If all is well, the functions described in the commentary will be available.

## ***Summary***

This is, as you will discover, an extremely enjoyable program to use, purely for its own sake but its real power comes in what it can do in livening up the output of your other programs. Because it does not actually relocate Basic, only limits the space available, new programs can be loaded into the machine to make use of the redefined character set. If the machine has been switched off since the character set was redefined, or the memory normalised, all that needs to be done is to add the first module to the front of subsequent programs (minus lines 11050–11090) and then to load the redefined character set from tape using the routine at 13560–13580. But do remember that if you redefine the letter A as a space invader character then every A output by the program, even in the program listing, will be redefined. For the sake of legibility it's usually better to stick to redefining the graphics characters!

Quite apart from the general usefulness of the program, however, you have also been introduced to some of the possibilities opened up by the 64's flexible memory structure and the techniques necessary to make the most of what is available. If you want to look further into memory manipulation you will need to get hold of a copy of the *Programmers Reference Manual* —with this program under your belt you should have no difficulty understanding and applying what you find there.

### *Going Further*

1) One simple addition to the program would be a routine to allow the position of two characters to be swapped, or for a redefined character to be placed at another location in the character set.
2) Making up a whole new character set with this program would be extremely time consuming. Why not try adding some block manipulation commands which would allow you to invert, turn, mirror etc. a whole set of characters between specified limits. Program listings look extremely interesting with all the letters upside-down!

## 3.3 SPRITES

With the Characters program entered we have prepared the way for an examination of one of the features of the 64 that other micro-owners can only dream of—sprites. With the advent of the 64, gone are the days when only machine code programmers could make high-resolution designs move smoothly and easily around the screen with an eerie realism. In the field of games especially, sprites represent a revolution in affordable micros.

In essence, a sprite is very little different from the user-defined characters we have been experimenting with. A great deal of technical imagination and competence has gone into the creation of the sprite facility, but when it comes to the user's part, a sprite is just a larger character which can be more flexibly moved around the screen.

Like the characters of the normal character set, sprites are defined by a series of bytes stored in RAM. Instead of an 8*8 grid, however, sprites use a grid which is 24 dots across by 21 down. Clearly this cannot be defined by the 64 bits present in eight bytes. In fact, each row of a sprite is defined by three bytes (24 bits) and, since there are 21 rows, it takes a total of 63 bytes to define a sprite. Sprite data can be stored at any secure place within the 16K block of memory addressed by the VIC II. Within this block, up to eight sprites can be defined at any one time but many more sprite designs can be held in reserve, if necessary, for instant activation

The main locations in memory which control the use of sprites are as follows:

a) 2040–2047: These eight locations are the sprite pointers. Their function is to indicate where in the 16K block the data for any particular sprite is to taken from. Since sprites are stored in blocks of 64 bytes (though they only use 63), the 256 values that can be POKEd into each pointer allow them to cover the whole of the 16K block. Thus, the data for sprite 2 will be taken from the memory at 64*PEEK(2042).

b) 53269: The sprite enable register. A sprite is only visible when the corresponding bit in this register is set.

c) 53248–53264: The sprite position registers. These work in pairs from 53248 to 53263, defining the X and Y co-ordinates of the top left-hand corner of the sprite grid on the screen. However, since the screen is actually wider (320 pixels) than the maximum value storable in a single byte (255), one bit at location 53264 is used to remember whether the position of each sprite on the X axis is more than 255. This gives a total of 512 possible positions on the X axis and 256 on the Y axis.

d) 53287–53294:The sprite colour registers. Each sprite can take on any of the 64's 16 colours, simply by POKEing the correct value into the appropriate register. There are in fact more locations than this which are relevant but these will do to be going on with.

The final issue to be decided is where to put the sprite data. If you only want three sprites, then a practical place is the Cassette Input—Output buffer which is located from 828 to 1019 (obviously you can't load or save data while the sprites are located there). If you want more sprites than that then you must set aside an area of memory for them, exactly the same situation as with user-defined characters. For the sake of variety, for our sprite-defining program we shall adopt a different solution to that offered for the Characters program. What we shall do is shift the start of the Basic program from 2048 to 4096, thus leaving ourselves 2K of memory in which to store up to 32 separate sets of sprite data. This is convenient in that it involves absolutely no shifting around of the video memory structure—what it will involve, however, is a resetting of the Basic start address before the program is loaded.

Having done that, the program, like the character generator, will allow the definition and manipulation of the sprite grids and the option of saving them to tape for use by later programs. The simplest way to enter this program is to first load Characters and adapt that program.

## Loader Program for Sprites

The following lines are NOT part of the main program, they are intended to be entered into the 64 and saved onto tape before the main program is entered and saved. The function of the program is to reset the beginning of Basic and then to load the main program into the reconfigured memory:

```
100 REM********************************
110 REM LOADER
120 REM********************************
130 POKE43,1:POKE 44,16:POKE 4096,0:CLR
140 LOAD 'SPRITES'
```

*Commentary*

Locations 43 and 44 are the pointers used by the system to the beginning of the Basic program, normally containing the values 1 and 8 (location 1 + 256*8 = 2049). All that the main line does is to alter this value to 4097. The first program byte must always be a zero, so this is POKEd in then the memory is cleared, completing the reconfiguration. When this has been done the main program is loaded automatically. At the risk of boring you, remember that this is NOT part of the main program—to include it at the beginning of the main program would chop off the first 2K of the program when it was run.

## Sprites: Table of Variables

(where different from Characters)

| | |
|---|---|
| SP | Address of current sprite pointer |
| SC | The address of the sprite colour register |
| SS | Start of sprite data |
| FNS(SN) | Start of block of data for sprite SN |
| SN | Sprite number |
| TT((20,23) | Array for temporary manipulation of sprite data |
| MM | Value to move SN |

MODULE 3.3.1

```
12000 REM#******************************
12002 REM SET UP SPRITE POINTERS
12005 REM*******************************
12010 SP=2040:SE=53269:SS=2048
12020 DEF FNS(SN)=SS+64*(SN)
12030 SN=0:DIM TT%(20,23):POKE 53281,6
```

The variables declared here are explained in the table of variables.

MODULE 3.3.2

```
13000 REM#*******************************
13010 REM PRINT GRID
13020 REM********************************
13030 POKE53269,1:POKE 53287,1:POKE SP,F
NS(SN)/64
13040 POKE 53248,0:POKE 53264,1:POKE 532
49,80
13050 PRINT "▯";:FOR I=1 TO 11:PRINT "▯▮
▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮ "
13060 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▒"
:NEXT
13070 PRINT "▯▮▯
 "
13080 PRINT "▯▒▒▮▒▮▒▮▒▮▒▮▒▮▒▮▒▮▒▮▒▮▒▮▒▮▒
"
13090 PRINT "▮▮";:FOR I=FNS(SN) TO FNS(S
N)+62 STEP 3:FOR J=0 TO 2
13100 FOR K=7 TO 0 STEP -1
13110 IF (PEEK(I+J) AND 2↑K)=2↑K THEN PR
INT "●";
13120 IF (PEEK(I+J) AND 2↑K)=0 THEN PRIN
T " ";
13130 NEXT K,J:PRINT:NEXT I
13140 PRINT "▮▮DESIGN NUMBER:";SN
13150 INPUT "NUMBER TO MOVE POINTER:(0=R
EDEF.):";MM:SN=SN+MM
13160 IF SN<0 THEN SN=0
13170 IF SN>31 THEN SN=31
13180 IF MM=0 THEN POKE 53248,0:POKE 532
64,1:POKE 53249,200:GOTO 14000
13190 GOTO 13000
```

Almost exactly the same as the grid drawing module in Characters.

*Commentary*

Line 13030: 53269 is the sprite enable register—this POKE sets bit zero and turns on sprite 0. 53287 is the colour register for sprite zero and the POKE sets the colour to white. The sprite zero pointer is set to point to the first 64 byte block in the reserved memory area.

Line 13040: Sprite zero is set at 256 on the X axis and 80 on the Y axis.

Lines 13050–13080: The outline of the grid is printed.

Lines 13090–13130: The I loop looks at the sprite data in groups of three, the J loop looks at each byte, the K loop looks at each bit. A circle is printed for each set bit.

Lines 13140–13190: The user can move the sprite pointer. Note that it is the area of memory pointed to by the sprite pointer, not the actual sprite pointer that is changed. In this program we shall always be using sprite zero.

*Testing Module 3.3.2.*

On running this module (remember that the loader program must first have been run) a garbage sprite will appear to the right of the grid and the individual dots will be filled in.on the grid. It is sometimes difficult to see the correspondence between the two because of the automatic shadowing placed into sprites. Two set bits on the same line with a space between them will actually appear as a block of black.

MODULE 3.3.3

```
14000 REM#********************************
14010 REM REDEFINE SPRITE
14020 REM********************************
14030 PRINT "■
                    ■":REM 39 SPACES
14040 F$="■■■■■■■■■■■■■■■■■■■■■■■■■■■■"
14050 PRINT F$;"■'I' INVERT"
14060 PRINT F$;"■'M' MIRROR"
14070 PRINT F$;"■'R' RETURN"
14080 PRINT F$;"■'1' INK IN"
14090 PRINT F$;"■'0' BLANK"
14100 PRINT F$;"■'T' TURN"
14110 PRINT F$;"■'P' MEMORY"
14120 PRINT F$;"■'D' SAVE"
14130 PRINT F$;"■'L' LOAD"
14140 PRINT F$;"■'E' END"
14150 PRINT F$;"■ARROWS MOVE"
14160 PRINT "■";
14170 GET A$
14180 CC=PEEK(211)+PEEK(210)*256+PEEK(20
9):PP=PEEK(CC):C1=55296+CC-1024
14190 C2=PEEK(C1)
14200 POKE CC,42:POKE C1,1
```

```
14210 FOR I=1 TO 15:NEXT:POKE CC,PP:POKE
 C1,C2:IF A$="" THEN 14170
14220 P1=INT((CC-1024)/40):P2=CC-(1024+4
0*P1)
14230 IF (P1>0 AND A$="▔") OR (P1<20 AND
 A$="▒") THEN PRINT A$;:GOTO 14170
14240 IF (P2>0 AND A$="▌") OR (P2<23 AND
 A$="▐") THEN PRINT A$;:GOTO 14170
14250 IF A$="1" THEN PRINT "▓▓▌";:GOTO 1
4170
14260 IF A$="0" THEN PRINT "▓ ▌▒";:GOTO
 14170
14270 IF A$<>"I" THEN 14330
14280 FOR I=0 TO 20:FOR J=0 TO 23
14290 IF PEEK(1024+I*40+J)=32 THEN 14310
14300 POKE 1024+40*I+J,32:POKE 55296+40*
I+J,182:GOTO 14320
14310 POKE 1024+I*40+J,81:POKE 55296+40*
I+J,181
14320 NEXT J,I:GOTO 14170
14330 IF A$="R" THEN GOTO 13030
14340 IF A$<>"M" THEN 14420
14350 FOR I=0 TO 20:FOR J=0 TO 23:TT%(I,
J)=0:NEXT J,I
14360 FOR I=0 TO 20:FOR J=0 TO 23
14370 IF PEEK(1024+40*I+J)=160 THEN TT%(
I,J)=1
14380 NEXT J,I
14390 PRINT"▓";:FORI=0TO20:FOR J=23TO0 S
TEP-1:IF TT%(I,J)=1 THEN PRINT "▓▓ ▓▒";
14400 IF TT%(I,J)=0 THEN PRINT "▓ ";
14410 NEXT J:PRINT:NEXT I:GOTO 14170
14420 IF A$<>"T" THEN 14500
14430 FOR I=0 TO 20:FOR J=0 TO 23:TT%(I,
J)=0:NEXT J,I
14440 FOR I=0 TO 20:FOR J=0 TO 20
14450 IF PEEK(1024+40*I+J)=81 THEN TT%(2
0-J,20-I)=1
14460 NEXT J,I
14470 PRINT"▓";:FORI=0TO20:FOR J=23TO0 S
TEP-1:IF TT%(I,J)=1 THEN PRINT "▓▓▒";
14480 IF TT%(I,J)=0 THEN PRINT "▓ ▒";
14490 NEXT J:PRINT:NEXT I:GOTO 14170
14500 IF A$<>"P" THEN 14560
```

```
14510 FOR I=0 TO 20:FOR J=0 TO 2:TT%(I,J
)=0:NEXT J,I
14520 FOR I=0 TO 20:FOR J=0 TO 2:FOR K=0
 TO 7
14530 IF PEEK(1024+40*I+J*8+K)=81 THEN T
T%(I,J)=TT%(I,J) OR 2↑(7-K)
14540 NEXT K,J,I
14550 FORI=0TO20:FOR J=0 TO 2:POKE FNS(S
N)+I*3+J,TT%(I,J):NEXT J,I:GOTO 13030
14560 IF A$<>"S" THEN 14620
14570 PRINT "[illegible]H
OW MANY SPRITES TO BE SAVED:";NN
14580 IF NN<1 OR NN>32 THEN 14570
14590 OPEN 1,1,1,"SPRITES":PRINT#1,NN
14600 FOR I=0 TO NN*64-1:T%=PEEK(SS+I):P
RINT#1,T%:NEXT
14610 CLOSE 1
14620 IF A$<>"L" THEN 14660
14630 OPEN 1,1,0,"SPRITES":INPUT#1,NN
14640 FOR I=0 TO NN*64-1:INPUT#1,T:POKE
SP+I,T:NEXT
14650 CLOSE1
14660 IF A$<>"E" THEN 14680
14670 POKE 53269,0:POKE 43,1:POKE 44,8:P
OKE 2048,0:CLR:END
14680 IF A$<>"X" THEN 14740
14690 INPUT "[illegible]NU
MBER TO EXCHANGE WITH:";S2
14700 IF S2<0 OR S2>31 THEN 14690
14710 FOR I=FNS(SN) TO FNS(SN)+62:LET T1
=PEEK(I):POKE I,PEEK(I+FNS(SN)-FNS(S2))
14720 POKE I+FNS(SN)-FNS(S2),T1:NEXT
14730 GOTO 13000
14740 IF A$<>"C" THEN 14840
14750 IF PEEK(53276)AND 1=1 THEN POKE 53
276,0:GOTO 14170
14760 INPUT "[illegible]INP
UT COLOUR FOR 01 (0-15):";C1
14770 IFC1<0ORC1>15THENPRINT"[illegible]
                                ":GOTO 14760
14780 INPUT "[illegible]INP
UT COLOUR FOR 10 (0-15):";C2
```

```
14790 IFC2<0ORC2>15THENPRINT"]
                                 ":GOTO 14780
14800 INPUT "[illegible]INP
UT COLOUR FOR 11 (0-15):";C3
14810 IFC3<0ORC3>15THENPRINT"]
                                 ":GOTO 14800
14820 POKE 53285,(PEEK(53285)AND240)ORC1
:POKE 53287,(PEEK(53287)AND240)ORC3
14830 POKE 53286,(PEEK(53286) AND 240) O
R C2:POKE 53276,1:GOTO 14170
14840 GOTO 14170
```

This module serves the same purpose as the character redefine module in the last program.

*Commentary*

Lines 14050–14150: Instructions for the use of the module.

Lines 14170–14210: Standard cursor move module.

Line 14220: Row and column of the cursor on the grid.

Line 14330: R returns to previous module.
Lines 14340–14490: Clockwise turn.

Lines 14500–14550: Redefined sprite is placed back into memory. The I loop scans each row of the grid, the J loop scans in groups of three bytes, the K loop scans each bit.

Lines 14560–14610: The sprite data is saved onto tape. The user has the option of declaring how many sprites are to be saved. This makes it possible to save three sprites, which can be stored in the tape input buffer of a subsequent program.
If any sprite is called up from tape for further manipulation, sprites currently in the memory are lost.

Lines 14660–14670: This routine switches off the sprite, normalises the memory and ends the program.

Lines 14680–14730: X allows the current sprite data to be exchanged with data at another position—particularly useful when making up a set of three.

Lines 14740–14830: This routine enables the user to enter or leave sprite multi-colour mode.

Line 14750: If multi-colour mode is set (ie the corresponding bit in the sprite multi-colour register at 53276 is set) and this function is called, then multi-colour mode is reset (turned off) for sprite zero.

Lines 14760–14800: The enigmatic 01, 10 and 11 in these prompts refer to bit combinations on the sprite grid. When in multi-colour mode, the sprite is regarded as having only 12 dots across (though they are twice as long). Bits are read in pairs from the left and, naturally, form pairs of 00,01,10 or 11. Each of these three combinations will produce a different colour in multi-colour mode,with 00 being the screen background coloured by the sprite multi-colour register at 53285. The 10 colour is dictated by the ordinary sprite colour register. 11 colour comes from the sprite multi-colour register at 53286.

*Testing Module 3.3.3*

All the functions described in the commentary should be available once this module has been entered. As with the equivalent module in Characters, it is better to test each function as it is entered.

***Summary***

A little thought about this program will demonstrate just how easy sprites are to use once the functions of a few memory locations are understood. The program itself will provide an endless series of sprites which can be stored for future use on a separate tape. The techniques contained in the program will make it a simple matter to make the best use of such sprites in your own creations.

***Going Further***

1) The program makes no provision for one other sprite function, and that is the expand capability, which doubles the height or width of the sprite (same number of bytes—just made longer). This would be a simple matter to add since all that is involved is setting the corresponding bit in the register at 53277 for the horizontal expansion and 53271 for the vertical. For the purposes of this program the correct bit is zero.
2) It would be useful to be able to pick up only part of a set of sprites from tape, say one at a time, to decide whether you wanted to compile it into the current set. A slight change to the load routine would enable you to do this.

## 3.4 HI-RES

Though the possibilities provided by user-defined characters and sprites are almost limitless, the 64 does provide yet another major graphics mode, bit-mapped graphics. What this means is that rather than being able to address a minimum of one of the 1000 character squares on the normal screen, the user is able to set any individual pixel (short for picture element) or dot on the screen. In this mode line drawings and curves can be drawn on the screen, though to make the fullest use of it you will need to get hold of the graphics extension cartridge for the 64, which will provide you with a variety of flexible graphics commands.

To understand the program given here it is necessary to know a little about the way the bit-mapped screen is set up. The screen itself contains 320*200 separate positions, a total of 64000. In order to store each of these separately, 8000 bytes of memory are needed, providing 64000 individual bits. Each of the standard character positions requires eight bytes (the 8*8 grid that we used for user-defined graphics). Starting from the top left-hand corner of the screen, the first 8 (0-7) bytes of the screen memory are used to create what would be on the normal screen the first character position. The second eight bytes form the second 8*8 grid and so on along the line. Since there are 40 character positions in a line, each line takes 320 bytes. In actual fact, because the bit-mapped mode enables individual pixels to be addressed, this line of 8*8 grids is capable of holding eight single pixel thickness lines (though if you drew them all it would look like a solid bar).

The 8K of memory necessary to hold the bit-mapped screen is obviously not storable in the normal 1K screen memory nor, in fact, can it even use that area as a part of its area, since 1024 to 2023 is used to store colour information for the bit- mapped screen. The solution adopted in the program that follows is to locate the screen beginning at 8192, leaving 6K of memory for the Basic program, with the option of relocating Basic if the program is developed and lengthened. Using the program given here you will be able to use the bit-mapped screen as a sketch-pad, using either the cursor move arrows or a simple line-drawing algorithm to create a design on the screen.

**Hi-Res: Table of Variables**

| | |
|---|---|
| DX | Distance between ends of line along X axis. |
| DY | Distance between ends of line along Y axis. |
| FN PE | The value that must be POKEd into PP to erase pixel X,Y. |
| FN PP | The location of the byte in which pixel X,Y falls. |
| FN PV | The value that must be POKEd into PP to set pixel X,Y. |
| MO | The current mode of the program. |
| | SC Start of screen. |

| | |
|---|---|
| SL | The slope of the line to be drawn. |
| X1,X2 | X co-ordinates of ends of line to be drawn. |
| Y1,Y2 | Y co-ordinates of ends of line to be drawn. |

MODULE 3.4.1

```
10000 REM#******************************
10010 REM INITIALISE HI-RES SCREEN
10020 REM******************************
10022 CL$="":INPUT "♥■CLEAR SCREEN (Y/N)
:";CL$
10025 REM POKE 44,64:POKE 43,1:POKE 1638
4,0:CLR
10027 DEF FNPP(X)=SC+320*INT(Y/8)+8*INT(
X/8)+(Y AND 7)
10028 DEF FNPV(X)=PEEK(FNPP(X)) OR (2↑(7
-(X AND 7)))
10029 DEF FNPE(X)=PEEK(FNPP(X)) AND (255
-2↑(7-(X AND 7)))
10030 POKE 53272,(PEEK(53272))OR 8:POKE
53265,PEEK(53265) OR32:SC=8192
10035 IF CL$="N" THEN 10050
10040 FOR I=SC TO SC+7999:POKE I,0:NEXT
10050 FOR I=1024 TO 2023:POKE I,6*16+12:
NEXT
10060 MO%(0)=2:MO%(1)=5:MO%(2)=10
```

This module configures the screen memory for the bit-mapped mode, defines some useful functions and clears the high resolution screen.

*Commentary*

Line 10025: The POKEs in this REM statement are not necessary for the running of this program. They are included in order that if you wish to expand the program in such a way that it may overrun the screen at 8192 and onwards, you will have the necessary information to relocate Basic. As with the Sprites program, the POKEs should be included in a loader program which is run BEFORE the main program. The program as given here works happily within the 6K of memory up to 8192—there is no necessity even to set a limit to the top of Basic.

Lines 10027–10029: The use of these functions is given in the table of variables.

Line 10030: 53272 is the register normally used to control where the VIC II looks for character data, in this case it will dictate the beginning of the bit-mapped screen. POKEing 8 in here sets the screen start to 8192. POKEing 53265 with 32 sets the bit-mapped mode.

Lines 10035–10040: In Line 10022, the user was given the option of clearing the screen. During the development of the program, when the program is stopped and RUN—RESTORE pressed, alterations can be made to the program without affecting the contents of the screen at all. On running the program again it saves time not to have to clear the 8000 bytes.

Line 10050: This line clears the normal screen memory area, which is now employed to hold the colour data for each of the 1000 normal character positions.

*Testing Module 3.4.1*

On first running the program, the screen should immediately fill with garbage. Gradually this will clear, leaving a screen which may still be covered with coloured squares corresponding to the position of characters on the normal mode screen. These too should then begin to clear and the screen be set to white. When the module is finished, press RUN and RESTORE to return to normal mode.

MODULE 3.4.2

```
11000 REM*******************************
11010 REM DRAW ON SCREEN
11020 REM*******************************
11030 X=160:Y=96:MO=1:POKE 1024,(PEEK(10
24)AND240) OR (MO*2)
11040 TT=PEEK(FNPP(X))
11042 GET A$:IF A$<>"" THEN 11050
11044 POKE FNPP(X),FNPV(X):POKE FNPP(X),
FNPE(X):GOTO 11042
11050 POKE FNPP(X),TT
11060 IF MO<3 THEN X=X-(A$="▐" AND X<319
)+(A$="▌" AND X>0)
11062 IF MO=3 THEN X=X-10*(A$="▐" AND X<
310)+10*(A$="▌" AND X>10)
11070 IF MO<3 THEN Y=Y-(A$="▣" AND Y<191
)+(A$="┐" AND Y>0)
11072 IF MO=3 THEN Y=Y-10*(A$="▣" AND Y<
182)+10*(A$="┐" AND Y>10)
11075 IFA$="*"THEN MO=MO+1:MO=MO+4*(MO>3
):POKE1024,(PEEK(1024)AND240)OR(MO*2)
```

```
11080 IF MO=1 THEN POKE FNPP(X),FNPV(X)
11090 IF MO=0 THEN POKE FNPP(X),FNPE(X)
11100 IF A$="1" THEN X1=X:Y1=Y
11110 IF A$="2" THEN X2=X:Y2=Y
11120 IF A$="L" THEN GOSUB 12000
11200 GOTO 11040
11499 GOTO 11499
```

This module allows a flashing pixel to be moved around the screen, inking in and erasing individual pixels.

***Commentary***

Line 11030: X and Y are the co-ordinates of the pixel on the 300*200 screen. The flashing pixel cursor is set to the middle of the screen. The first position in normal screen memory is POKEd with a value which produces a colour indicator of the current mode (black = 0,red = 1, purple = 2,blue = 3). Effects of modes will be explained later.

Line 11040: The state of the screen at the position at which the cursor is to be flashed is obtained.

Lines 11042–11050: The cursor is flashed on and off until a key is pressed.

Lines 11060–11072: In mode 3, pressing the cursor arrow results in the flashing pixel moving 10 positions in the required direction (within screen limits). In modes 0,1 and 2 the cursor moves only one space at a time.

Line 11075: The unshifted function keys, from top to bottom, are used to set the modes. If the mode is changed the colour indicator is changed.

Lines 11080–11090: If the mode is zero (black) then the pixel at the cursor position is blanked. If the mode is 1 (red) then the pixel is inked in. The remaining two modes allow the cursor to be moved around, slow or fast, without affecting what is on the screen.

Lines 11100–11120: These inputs relate to the next module.

*Testing Module 3.4.2*

You should now be able to move the tiny cursor around the screen, drawing or erasing.

MODULE 3.4.3

```
12000 REM******************************
12010 REM LINE DRAWING
12020 REM******************************
12025 X=X1:Y=Y1
12030 DX=X2-X1+SGN(X2-X1):DY=Y2-Y1+SGN(Y
2-Y1)
12032 IF ABS(DY)>ABS(DX) THEN 12200
12035 SL=ABS(DY/DX)-0.5
12040 FOR I=1 TO ABS(DX)
12050 IF MODE=1 THEN POKE FNPP(X),FNPV(X
)
12055 IF MODE=0 THEN POKE FNPP(X),FNPE(X
)
12060 IF SL>0 THEN Y=Y+SGN(DY):SL=SL-1:G
OTO 12060
12070 SL=SL+ABS(DY/DX)
12100 X=X+SGN(DX):NEXT I
12120 RETURN
12200 SL=ABS(DX/DY)-0.5
12210 FOR I=1 TO ABS(DY)
12220 IF MODE=1 THEN POKE FNPP(X),FNPV(X
)
12225 IF MODE=0 THEN POKE FNPP(X),FNPE(X
)
12230 IF SL>0 THEN X=X+SGN(DX):SL=SL-1:G
OTO 12230
12240 SL=SL+ABS(DX/DY)
12250 Y=Y+SGN(DY):NEXT I
12300 RETURN
```

This module provides for the drawing of straight lines between points defined by the user. It is an adaptation of a method known as Bresenham's algorithm and a version of it is often used in those Basics which have line drawing commands.

*Commentary*

Line 12025: The values X1 and Y1 were defined when the user input 1—at that point they were set equal to the X and Y positions of the cursor. X2 and Y2 were set on input of 2. The line will be drawn from X1,Y1.

Line 12030: DX and DY are set equal to the distance between X1 and X2, and Y1 and Y2, plus one. The SGN function means that it does not make any difference if the distance is positive or negative (if it is negative then minus one will be added rather than 1).

Line 12032: The line-drawing algorithm uses the greater of the two differences as the basis of its calculations so it is faster to have two separate routines.

Line 12035: SL is the slope, or ratio between DX and DY minus 0.5.

Lines 12040: The loop is as long as the difference along the X co-ordinate.

Lines 12050–12055: Depending on whether the mode is 0 or 1, a single dot on the line is erased or drawn. Note that nothing will happen in modes 2 or 3.

Line 12060: According to the ratio between DX and DY, SL may now indicate that the next dot should move up or down the Y axis. If so the Y position is changed and SL is reduced by one.

Line 12070: The slope value is added to SL each time a dot has been printed.

Line 12100: The X position is incremented for each iteration of the loop. Once again the SGN function takes care of lines which move backwards along the axis.

Lines 12200–12250: Exactly the same routine for those cases where DY is greater than DX.

*Testing Module 3.4.3*

You should now be able to specify a start and end point for a line (1 and 2)then to draw it or erase an existing line, depending upon whether mode 1 or 0 is set.

***Summary***

This program is intended as no more than an appetiser for the possibilities raised by the bit-mapped mode. Full use of bit-mapped graphics requires some careful thought as to what you wish to achieve and some often complex mathematics to achieve it. Should you decide to go further, the techniques given here, and the functions used to locate individual pixels, will make the task that much easier.

### *Going further*

1) Why not add a facility allowing the saving of a screen of graphics onto tape—you'll need a fairly long tape but the routine would be simple enough.
2) Computer graphics books provide a number of algorithms which allow the drawing of circles and arcs. Why not add a module to the end of the program to achieve this—the main drawback will be lack of speed.

# CHAPTER 4
# The 64 as Secretary

Sooner or later, most micro-owners realise that their new digital friend really comes into its own when it is storing information, processing it and presenting it in a variety of ways that would be laborious in the extreme if done manually. They then begin the task of writing simple programs which will store their friends' names and addresses or catalogue their record collection. They may end up with half a dozen programs, each limited to a single use, and yet each program employing much the same methods.

In this chapter we begin a section of more substantial programs by examining how a single program can be written to satisfy a wide variety of filing needs without the constant need for rewriting every time a new application comes along.

## 4.1 UNIFILE

The first program is called Unifile and, in the form presented here, it is capable of storing up to 500 entries, as well as allowing the user to search through them for named items, to amend entries and to delete them. Quite apart from the wide applications of such a program, I hope that the simple act of entering it and understanding the methods used will provide you with a host of ideas for further applications.

**Unifile: Table of Variables**

| | |
|---|---|
| IN | Flag indicator used to show whether the program has been initialised. |
| A%(499,X-1) | Records the length of individual items in each entry. |
| A$(499) | Main file array. |
| B$(X-1) | Holds names of item types for each entry. |
| FF | Flag indicator used to determine whether a user search has been successful. |
| IT | Number of entries in file so far. |
| PO | Used in binary search to indicate number of search samples necessary. |
| PP | Pointer to start position of current item to be printed from an entry. |
| R$ | Separator for use in saving data on tape. |

| | |
|---|---|
| S1 | Temporary search pointer for user search module. |
| SS | Main search pointer in binary search module. |
| T1$ | Temporary storage string used to build up new entry. |
| TI%(20) | Temporary storage for length of items being built up into new entry. |
| X | Holds number of items specified for each file. |
| Z | Indicator for number of program function to be called up from main menu. |

MODULE 4.1.1

```
11000 REM*******************************
11010 REM MENU
11020 REM*******************************
11030 POKE 53281,7:PRINT "
UNIFILE"
11040 PRINT "COMMANDS AVAILABLE:"
11050 PRINT "   1)ENTER INFORMATION"
11060 PRINT "   2)SEARCH/DISPLAY/CHANGE
"
11070 PRINT "   3)DATA FILES"
11080 PRINT "   4)SET UP NEW FILE"
11090 PRINT "   5)STOP"
11100 INPUT "WHICH DO YOU REQUIRE:";Z:
PRINT "";
11110 IF Z>3 OR IN=1 THEN 11140
11120 PRINT "NOT INIT
IALISED YET.":FOR I=1 TO 1000:NEXT
11130 GOTO 11000
11140 ON Z GOSUB 13000,17000,18000,12000
,11150:GOTO 11000
11150 PRINT "FILING
SYSTEM CLOSED":END
```

The purpose of the module is to present all the functions which the program makes available and to allow the user to make a choice between them. As a rule of thumb, any complex program which does not begin with a clear-cut menu of what the program does, is a bad program. And if you don't agree with that statement now, you certainly will at some time in the future when you have to return to a complex program which has not been used for a few weeks and find youself spending half an hour going through the listing trying to remind yourself what it does and how.

*Commentary*

Line 11030: A typical use of Commodore's flexible cursor control commands. The string clears the screen, moves the cursor down one space, across to the middle of the line, sets the RVS ON characteristic and prints in green.

Lines 11110–11130: No program can be successfully run unless the arrays it uses have been set up. In this program the variable IN is set to 1 when that happens. If IN is not equal to 1 then the only functions available from the menu are initialisation (setting up the arrays) and stop.

Line 11140: For those to whom this command is new, ON...GOSUB and ON...GOTO are simply ways of cutting down on lists of messy IF...THEN...GOSUB (GOTO) statements. The command will choose the destination in the list which is designated by Z.

At this stage, all that can be tested is that the module presents a neatly ordered menu page and accepts an input. The only input that will not produce an error report is 5—program stop.

MODULE 4.1.2

```
12000 REM#*****************************
12010 REM STRUCTURE OF FILE
12020 REM******************************
12030 CLR:DIM A$(499):PRINT "▮▮▮▮▮▮▮▮▮▮▮
▮▮▮▮▮FILE STRUCTURE":IN=1:R$=CHR$(13)
12035 INPUT "▮▮ARE YOU LOADING FROM TAPE
 (Y/N):";Q$:IF Q$="Y" THEN 11000
12040 INPUT "▮▮▮HOW MANY ITEMS IN EACH E
NTRY:";X:DIM B$(X-1),A%(499,X-1)
12050 PRINT "▮";:FOR I=0 TO X-1:PRINT "▮
NAME OF ITEM";STR$(I+1);":";:INPUT Q$
12060 B$(I)=Q$:NEXT I:GOTO 11000
```

This module performs the essential function of setting up the arrays which will be used to store the program data—until it has been called up, the program cannot be used. Once data has been entered, calling up the module again will result in the loss of all the data—the memory is wiped clean ready for a new set of data. The use of the main variables is explained in the table of variables and during the subsequent commentary on the program.

*Commentary*

Line 12030: Note that before any array is dimensioned, the memory must be cleared. Failure to do this results in the REDIMMED ARRAY error message.

Line 12040: Unifile does not dictate to the user how many items the typical entry can contain, it is up to the user to specify. Once this is done the program configures the pointer array A% and the item title array B$ accordingly.

Lines 12050–12060: Having specified the number of items per entry, the items are named eg name, address, telephone number. Note that because the memory has been cleared, the module cannot RETURN to the menu, it has to be given the specific line to GOTO.

*Testing Module 4.1.2*

On calling up the module you should be asked to specify the number of items per file and to give names to the items.

MODULE 4.1.3

```
13000 REM#**********************************
13010 REM ENTRY OF NEW ITEMS
13020 REM***********************************
13030 T1$="":PRINT "█████████████████████NE
W ITEMS"
13040 PRINT "███████████████████████████";I
T;" ITEMS SO FAR"
13050 PRINT "██COMMANDS AVAILABLE:"
13060 PRINT "██>█ENTER ITEM SPECIFIED"
13070 PRINT "█>█INPUT 'ZZZ' TO RETURN TO
 MENU██"
13080 FOR I=0 TO X-1:PRINT B$(I);":";:IN
PUT Q$:IF Q$="ZZZ" THEN RETURN
13090 IFLEN(T1$)+LEN(Q$)<=255THEN 13110
13100 PRINT "█ENTRY TOO LONG.":FOR J=1 T
O 3000:NEXT J:RETURN
13110 T1$=T1$+Q$:TI%(I)=LEN(T1$):NEXT I:
PRINT "██WAIT"
13120 GOSUB14000:GOSUB 15000:GOTO 13000
```

The purpose of this module is to accept the input of the items specified by the user and to compile them into an entry ready for the main file.

*Commentary*

Line 13080: Using the variable X to determine the number of repetitions, the program prompts the user to input each of the named items.

Lines 13090–13100: Individual entries can be a maximum of 255 characters long—the maximum length of a single string on the 64. These lines check that the limit is not being exceeded.

Line 13110: The item input is added to the temporary storage string T1$ and the length of the entry so far is recorded in TI%. Note that TI% was not declared in the initialisation module. Simply mentioning it in the course of the program, automatically dimensions it with 10 elements (0-9). If you want to have entries with more than 10 items then you must declare a bigger TI% in the initialisation module.

*Testing Module 4.1.3*

At this stage, by entering temporary RETURNs at lines 14000 and 15000, you should be able to call up this module and be prompted to input items under the names you have specified. Note that there is not yet any provision to enter these into the file.

MODULE 4.1.4

```
14000 REM#***************************
14010 REM BINARY SEARCH
14020 REM****************************
14030 IF IT=0 THEN SS=0:RETURN
14040 PO=INT(LOG(IT)/LOG(2)):SS=2↑PO-1
14050 FOR I=PO TO 0 STEP-1
14060 IF A$(SS)<T1$ THEN SS=SS+2↑I
14070 IF A$(SS)>T1$ THEN SS=SS-2↑I
14080 IF SS<0 THEN SS=0
14090 IF SS>IT-1 THEN SS=IT-1
14100 NEXT I:IF A$(SS)<T1$ THEN SS=SS+1
14110 RETURN
```

Of all the modules in this program, this one is most likely to look like double dutch on first sight. In reality it is very simple, but first you need to understand the basic principles that lie behind a method of searching for something called the binary search, which dramatically reduces the amount of work needed to find the right place for a new item in an ordered list of data.

Consider the following example:

We have established a file containing 2,000 names in alphabetical order and there is a new name to be inserted, whose rightful place will actually be at position 1731, though this has yet to be determined. The search routine therefore begins by examining the first name in the file, decides that the new name will come after it and moves on to the second name. Eventually, after examining 1732 names, the search routine finds a name which the new name should come before and it knows that it has found the right place to insert the new name. This is a straightforward procedure and one that is easy to program but compare it with this:

The search procedure begins by examining the name in position 1024 of the file, because 1024 is the greatest power of 2 that can be fitted into the total number of names in the file. The name at 1024 is found to be alphabetically less than the new name, so the search routine adds 1024/2 to the original 1024 and moves on to name number 1536. That name is still less than the new name, so 1024/4 is added to 1536, making 1792. Now something different happens—name number 1792 is alphabetically greater than the new name—the solution is to subtract 1204/8, giving 1664. The search routine goes on adding or subtracting decreasing powers of 2 to build a search pattern that looks like this:

1644 (then add 64)
1728 (then add 32)
1760 (then subtract 16)
1744 (then subtract 8)
1736 (then subtract 4)
1732 (then subtract 2)
1730 (then add 1)

The number of comparisons needed to find the correct place in the file has been reduced from 1732 to 10. The power of the binary search should be apparent.

Line 14030: If there are no items in the file yet, then the right position does not have to be calculated.

Line 14040: The LOG function is used to find the maximum power of 2 that will fit the current number of items.

Lines 14050–14100: The binary jump is performed, with checks to see that the search is not leaving the ends of the file. One final comparison is made when the loop is finished and the correct position has been determined and stored in the variable SS.

### *Testing Module 4.1.4*

Full testing of the module will have to wait until the next module has been entered but a check that the syntax is correct can be made by simply calling up the insert module, from which this module is called.

MODULE 4.1.5

```
15000 REM#********************************
15010 REM INSERT
15020 REM*********************************
15030 IF IT=0 THEN GOTO 15060
15040 FOR I=IT TO SS+1 STEP -1:A$(I)=A$(
I-1)
15050 FOR J=0 TO X-1:A%(I,J)=A%(I-1,J):N
EXT J,I
15060 A$(SS)=T1$:FOR I=0 TO X-1:A%(SS,I)
=TI%(I):NEXT:IT=IT+1:RETURN
```

The correct position having been determined, this module moves all the entries from that position onwards, one space up the file, together with their associated pointers in A%. The new entry is placed into position SS of the file, and the pointers which show the length of the individual items are placed into the same position in A%.

### *Testing Modules 4.1.4 and 4.1.5*

You should now be able to input entries to the file which will be placed into alphabetical order. To check this you must stop the program and print out, in direct mode, the contents of A$(0), A$(1) etc. You should also check that the pointers stored in the same line of A% do in fact point to the last character of each item in the entry.

MODULE 4.1.6

```
18000 REM#********************************
18010 REM DATA FILES
18020 REM*********************************
18030 PRINT "■■POSITION TAPE CORRECTLY,
THEN ■ENTER■←"
18040 INPUT "MOTOR WILL STOP AUTOMATICAL
LY:";O$:POKE 192,7:POKE 1,39
18050 PRINT "■■COMMANDS AVAILABLE:":PRIN
T "■■ 1)SAVE DATA":PRINT " 2)LOAD DATA"
18060 INPUT "■■WHICH DO YOU REQUIRE:";Q:
ON Q GOTO 18070,18120:RETURN
```

```
18070 POKE 1,7:FOR I=1 TO 2000:NEXT
18080 OPEN 1,1,1,"UNIFILE":PRINT#1,IT,R$
,X
18090 FOR I=0 TO IT-1:PRINT#1,A$(I):FOR
J=0 TO X-1:PRINT#1,A%(I,J):NEXT J,I
18100 FOR I=0 TO X-1:PRINT#1,B$(I):NEXT
18110 CLOSE1:RETURN
18120 OPEN 1,1,0,"UNIFILE":INPUT#1,IT,X:
DIM B$(X-1),A%(499,X-1)
18130 FOR I=0 TO IT-1
18132 GET#1,T$:IF T$<>CHR$(13) THEN A$(I
)=A$(I)+T$:GOTO 18132
18134 FOR J=0 TO X-1:INPUT#1,A%(I,J):NEX
T J,I
18140 FOR I=0 TO X-1:INPUT#1,B$(I):NEXT
18150 CLOSE1:RETURN
```

Now that you can input some data to the file, the first thing to do is to store some data on tape then, as you enter new modules or change lines to correct errors, you will not have to go through the chore of re-entering all the data every time.

*Commentary*

Line 18040: Having positioned your tape, you may wish to first place it into RECORD or PLAY mode and run up to precisely the point indicated by the tape counter. When the precise point is reached, pressing RETURN switches off the cassette recorder motor by the use of these two POKEs.

Line 18070: The recorder motor is switched back on before any data is recorded and a header printed in addition to that added automatically by the operating system—this helps to ensure that you do not record on the non-magnetic leader of the cassette if you are starting from the beginnning.

Lines 18120–18140: Data which was printed into the file is now recalled. Note that because the strings in the main file may be more than 80 characters long, we cannot use the INPUT command. Instead, each character of the strings in the main file is picked up separately using GET, and each entry is considered complete when a carriage return character is picked up from tape.

*Testing Module 4.1.6*

The simple test for this module is whether you can input data to the program, save it on tape and then reload it.

MODULE 4.1.7

```
17000 REM*******************************
17010 REM SEARCH
17020 REM*******************************
17030 S1=0:FF=0:PRINT "▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
▒▒SEARCH"
17040 PRINT "▒▒COMMANDS AVAILABLE:"
17050 PRINT "▒ ▒>▒INPUT ITEM FOR NORMAL
SEARCH"
17060 PRINT " ▒>▒PRECEDE WITH 'III' FOR
INITIAL SEARCH"
17070 PRINT " ▒>▒PRECEDE WITH 'SSS' FOR
SPECIAL SEARCH"
17080 PRINT " ▒>▒▒ENTER▒ FOR FIRST ITEM
ON FILE"
17090 PRINT "▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒";
17100 T1$="":INPUT"▒▒▒INPUT SEARCH COMMA
ND:";T1$
17110 IF LEFT$(T1$,3)<>"III" THEN 17140
17120 T1$=RIGHT$(T1$,LEN(T1$)-3):GOSUB 1
4000:S1=SS:IF S1>IT-1 THEN RETURN
17130 GOTO 17240
17140 IF LEFT$(T1$,3)<>"SSS" THEN 17190
17150 FF=0:T1$=RIGHT$(T1$,LEN(T1$)-3):FO
R I=S1 TO IT-1:FOR J=1 TO LEN(A$(I))
17160 IF MID$(A$(I),J,LEN(T1$))=T1$ THEN
 FF=1:S1=I:J=LEN(A$(I)):I=IT-1
17170 NEXT J,I:IF FF=1 THEN T1$="SSS"+T1
$:GOTO 17240
17180 RETURN
17190 IF T1$="" THEN 17240
17200 FF=0:FOR I=S1 TO IT-1:PP=0:FOR J=0
 TO X-1
17210 IF MID$(A$(I),PP+1,A%(I,J)-PP)=T1$
THEN FF=1:S1=I:J=X-1:I=IT-1
17220 PP=A%(I,J):NEXT J:NEXT I:IF FF=1 T
HEN 17240
17230 RETURN
17240 IF S1>IT-1 THEN S1=IT-1
```

```
17250 IF IT=0 THEN RETURN
17260 IF S1<0 THEN S1=0
17270 PRINT "■■■ENTRY ";S1+1;":-■":PP=0
17280 FOR I=0 TO X-1:PRINT"■";B$(I);":■"
;MID$(A$(S1),PP+1,A%(S1,I)-PP)
17290 PP=A%(S1,I):NEXT I:S1=S1+1:PRINT "
■■■■■■■■■■■■■■■■■■■■■■■■■■■"
17300 PRINT "■■SEARCH■ ■COMMANDS AVAILAB
LE:"
17310 PRINT "■ ■>■■ENTER■ FOR NEXT ITEM"
17320 PRINT " ■>■'AAA' TO AMEND'
17330 PRINT " ■>■'CCC' TO CONTINUE SEARC
H"
17340 PRINT " ■>■'#' FOLLOWED BY NO. TO
MOVE POINTER"
17350 PRINT " ■>■'ZZZ' TO QUIT FUNCTION'
17360 P$="":INPUT "■■WHICH DO YOU REQUIR
E:";P$
17370 IF P$="CCC" THEN 17110
17380 IF P$="" THEN 17240
17390 IF P$="AAA" THEN GOSUB 16000:GOTO
17240
17400 IF P$="ZZZ" THEN RETURN
17410 IF LEFT$(P$,1)="#" THEN S1=S1+VAL(
MID$(P$,2))-1:GOTO 17240
17420 S1=S1-1:GOTO 17240
```

Having placed your data into the 64 it would be nice to think that you could get it back again. The purpose of this module is to enable you to do just that, retrieving the information stored in a variety of ways that make the filing system more useful.

## *Commentary*

Lines 17110–17130: If the item to be searched for is preceded by the letters III, then the binary search module is called up to find an entry which begins with the letters specified or the one nearest to the right position if there is no correct entry. Note that this will not necessarily be the first item in the file to satisfy the condition, so if you are using the initial search function to find the first entry beginning with L, for instance, you will get an entry beginning with L and can page backwards to see if it is the first. IIILA would get you closer, while IIILAAAA should do the trick unless you are storing some very unusual names.

Lines 17140–17180: Preceding the item to be searched for with SSS will result in the whole file being scanned for that combination of characters—it doesn't have to be a whole item. SSSLO would pick up any entries containing London, loganberries or hello. This search is necessarily slower than any of the others.

Line 17190: If you have pressed RETURN, without an input, the first item on the file will be displayed.

Lines 17200–17230: Any other input will be understood as a full item to be searched for and only those entries which have an item in exactly that form will be returned. Note, in this routine, how the pointer array A(, which gives the position of the last character of each item in an entry, is used to extract items from the entry even though there is no visible marker for the items if an entry is printed out in direct mode.

Lines 17270–17290: The entry which the search module has discovered is printed onto the screen.

Lines 17300–17420: Having displayed an entry, the program now gives you the option of viewing the next entry, amending the entry, continuing the search specified, moving to another entry by entering NN, where NN is a positive or negative number to move along the file, or returning to the main menu. If a recognisable input is not made, the same entry is displayed again.

*Testing Module 4.1.7*

You should now be able to display any data that you have stored and to search through it using the search methods described. You cannot yet amend entries.

MODULE 4.1.8

```
16000 REM#******************************
16010 REM CHANGE ENTRY
16020 REM*******************************
16030 S1=S1-1:T1$=""
16040 PP=0:FOR I=0 TO X-1:PRINT "■■■ENTR
Y ";S1+1;":-■"
16050 PRINT"■";B$(I);":■";MID$(A$(S1),PP
+1,A%(S1,I)-PP)
16060 PRINT "■■■■■■■■■■■■■■■■■■■■■■■■■■■
■■AMEND"
16070 PRINT "■■COMMANDS AVAILABLE:"
16080 PRINT "■ ■>■■ENTER■ LEAVES ITEM UN
```

```
CHANGED"
16090 PRINT " ■>▮INPUT NEW ITEM TO REPLA
CE ONE SHOWN"
16100 PRINT " ■>▮'DDD' DELETES WHOLE ENT
RY"
16110 PRINT " ■>▮'ZZZ' LEAVES ENTRY UNCH
ANGED"
16120 Q$="":INPUT "■■WHICH DO YOU REQUIR
E:";Q$
16130 IF Q$="ZZZ" THEN RETURN
16140 IF Q$="" THEN Q$=MID$(A$(S1),PP+1,
A%(S1,I)-PP)
16150 IFQ$="DDD"THEN GOSUB 16180:RETURN
16160 PP=A%(S1,I):T1$=T1$+Q$:TI%(I)=LEN(
T1$):NEXT I:GOSUB 16180:GOSUB 14000
16170 S1=SS:GOSUB 15000:RETURN
16180 FOR J=S1 TO IT-1:A$(J)=A$(J+1):FOR
 K=0 TO X-1:A%(J,K)=A%(J+1,K):NEXT K,J
16190 IT=IT-1:RETURN
```

The purpose of this module is to allow you to make changes to items in entries which have already been stored, without having to make the whole entry over again, as well as to delete items or whole entries if desired.

### *Commentary*

Line 16140: The module's method of working is similar to that of the main input module, except that if RETURN is pressed, the item being input is defined as being the current item on display.

Line 16180: This routine moves all the following entries in the file down one place, thus erasing the current entry.

### *Testing Module 4.1.8*

You should now be able to amend items in an entry arrived at in the search module or to delete the whole entry. If this module is working correctly, then the program is ready for use.

### *Summary*

You have now completed the entry of a substantial and complex program which I hope you will find useful in a variety of applications. Along with that process you have also learned a number of techniques which will stand you in good stead whenever you decide on ambitious programs of your own to store and process non-numeric data.

More importantly, however, if you have taken the trouble to understand what you have been entering, tracing through the functions of the individual lines, as well as the overall functions of the modules, you will have gained confidence that substantial and complex programs are not always as awesome as they are made out to be. Using a modular approach, which breaks down the program into a series of manageable tasks, applications like this one can be developed by anyone who is prepared to invest a little time (and a little hair).

**Going Further**

1) If you have a printer then you will want to add some provision for outputting entries or groups of entries onto paper. The easiest way to do this would be to add another command to the second part of the Search Module.

2) One interesting challenge would be to see whether you could give the program the ability to deal with numeric data as well as non-numeric. This would involve setting up a numeric array with 500 elements, with provision to input values to it and perhaps some search commands along the lines of 'find any entries which hold a value of greater than X'. There are quite a large range of applications where the ability to store one or more numeric items would be an advantage.

## 4.2 UNIFILE II-DATABASE

After entering Unifile and debugging it, the last thing that you may want to face is a variation on the same theme. If so, feel free to skip this program for the present and move on to greener pastures. At some stage, however, you will want to come back to this program to solve at least some of the problems that Unifile is not designed to cope with. Unifile is fine for files which have a regular structure, and many do. Equally, there are a large number of applications where you simply do not know in advance how many items there are going to be in a particular entry. You may, for instance, want to catalogue your books. You could set up the original Unifile program to request author and title, but with probably many more than one book by most authors, tagging the author's name onto every individual title is going to be a considerable waste of space.

Unifile II is designed to cope with such less structured files. It is more flexible than Unifile in that you can go on adding items to an entry as long as you like within the overall limit of 255 characters and can specify a more complex form of search which will seek out any entries which contain up to 10 separate search targets. This flexibility has a price, however, in that the program is more complicated to use—there are none of the easy prompts to dictate which item to input next. In addition, if you want to label items within an entry with a title, you will have to specify what those titles are and attach them in a coded form—the program has no idea what is coming next so you have to.

Because the program is similar in structure to Unifile, the easiest way to enter it is to first load Unifile itself. As you enter Unifile II you will find that many of the program lines are identical, or nearly so, even if the numbering differs. Renumbering those lines before going on to deal with the differences will save you an enormous amount of time.

**Unifile II: Table of variables**

| | |
|---|---|
| B$(49) | Contains the optional item titles specified. |
| EX | Temporary indicator to show that an extra item has been added to an entry during the Amend module. |
| FNA(S1) | Function which extracts from the value of the last character in an entry the number of items within that entry. |
| FNB(S1) | Function which obtains the position of the last character of an item within an entry. This function must be used within a loop with a loop variable I specifying the number of the item. |
| NN | Temporary variable registering the number of items within an entry being input. |
| SS$ | Item extracted from entry on the basis of FNA and FNB. |
| S2 | Temporary pointer used during searches. |
| S3 | Temporary record of value of S1 during multiple search |
| TI((49) | Temporarily used to store the position of items within an entry that is being input. |
| TN | The type number of an item if one is specified. |

MODULE 4.2.1

```
11000 REM******************************
11010 REM MENU
11020 REM******************************
11030 POKE 53281,7:PRINT "[illegible]
[illegible]UNIFILE"
11040 PRINT "[illegible]COMMANDS AVAILABLE:"
11050 PRINT "[illegible]    1)ENTER INFORMATION"
11060 PRINT "[illegible]   2)SEARCH/DISPLAY/CHANGE
"
11070 PRINT "[illegible]   3)NEW TYPE NAMES"
11080 PRINT "[illegible]   4)DATA FILES"
```

```
11090 PRINT "▮   5)SET UP NEW FILE"
11100 PRINT "▮   6)STOP"
11110 INPUT "▮▮WHICH DO YOU REQUIRE:";Z:
PRINT "▯";
11120 IF Z>4 OR IN=1 THEN 11150
11130 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮NOT INIT
IALISED YET.":FOR I=1 TO 1000:NEXT
11140 GOTO 11000
11150 ON Z GOSUB 13000,17000,19000,20000
,12000,11160:GOTO 11000
11160 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮FILING
SYSTEM CLOSED":END
```

A standard menu module.

MODULE 4.2.2

```
12000 REM#******************************
12010 REM INITIALISE FILE
12020 REM*******************************
12030 CLR:DIM A$(499),B$(49),TI%(49):IN=
1
12040 DEF FNA(S1)=ASC(RIGHT$(A$(S1),1))+
1
12050 DEF FNB(S1)=ASC(RIGHT$(A$(S1),FNA(
S1)-I+1))
12060 GOTO 11000
```

The module initialises the arrays and returns immediately to the menu.

MODULE 4.2.3

```
13000 REM#******************************
13010 REM ENTRY OF NEW ITEMS
13020 REM*******************************
13030 T1$="":NN=-1:TN=0:PRINT "▮▮▮▮▮▮▮▮▮
▮▮▮▮▮▮▮▮NEW ITEMS"
13040 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮";I
T;" ITEMS SO FAR"
13050 PRINT "▮▮COMMANDS AVAILABLE:"
13060 PRINT "▮▮>▮ENTER ITEM TO BE INPUT"
13070 PRINT "▮>▮ENTER '*' TO TERMINATE T
HIS RECORD"
```

```
13080 PRINT "■>▒ENTER '↑NN' FOR ITEM TYP
E"
13090 PRINT "■>▒INPUT 'ZZZ' TO RETURN TO
 MENU■■"
13100 INPUT Q$:IF Q$="ZZZ" THEN RETURN
13110 IF Q$="ZZZ" THEN RETURN
13120 IF LEFT$(Q$,1)<>"↑" THEN 13150
13130 TN=VAL(MID$(Q$,2)):TN=TN+10
13140 PRINT"■";B$(TN-11);":";:GOTO13100
13150 IF TN<>0 THEN Q$=Q$+"↑"+MID$(STR$(
TN),2):TN=0
13160 IF LEN(T1$)+LEN(Q$)+NN+2<=255 THEN
 13180
13170 PRINT "■ENTRY TOO LONG.":FOR J=1 T
O 3000:NEXT J:RETURN
13180 IF Q$="*" THEN GOTO 13200
13190 T1$=T1$+Q$:TI%(NN+1)=LEN(T1$):NN=N
N+1:GOTO 13100
13200 FOR I=0 TO NN:T1$=T1$+CHR$(TI%(I))
:NEXT:T1$=T1$+CHR$(NN+1)
13210 PRINT "■■WAIT":GOSUB14000:GOSUB 15
000:GOTO 13000
```

This module is equivalent to the entry module of Unifile but is more complex for two reasons:

1) There must be provision within the module to tell the program when an entry is finished. This is done by entering an asterisk without other input.

2) Since there is no regular structure to the file, regular prompts for the names of items to be input cannot be provided. There is, however, provision in the program for items to be named. Such names, and the numbers to be given to them are defined by a subsequent module—in this module the item name can be attached to an item by first of all entering the '↑' symbol followed by the number previously given to the desired type name.

*Commentary*

Lines 13100–13140: If an entry begins with the '↑' symbol then the characters following it are taken to be the number of an item name to be attached to the item about to be input. The input prompt is now repeated under that type name. The type number is stored at the end of the item, its value increased by ten so that it will always be a two digit number (there are 50 possible type numbers/names).

Line 13200: To the string of items which has been built up is now added a number of characters whose code value is equal to the position of the last character of each item. To the very end of the string is added another character whose code value is equal to the number of items in the entry. Note that when saving the entries onto tape, these characters must be translated into numbers since the characters may fall outside the range of those which can be saved in character form.

*Testing Module 4.2.3*

The module cannot be fully tested but a running check can be made by entering temporary RETURNs at 14000 and 15000. You should then be able to enter items and terminate the entry with an asterisk.

MODULE 4.2.4

```
14000 REM#*******************************
14010 REM BINARY SEARCH
14020 REM********************************
14030 IF IT=0 THEN SS=0:RETURN
14040 PO=INT(LOG(IT)/LOG(2)):SS=2↑PO-1
14050 FOR I=PO TO 0 STEP-1
14060 IF A$(SS)<T1$ THEN SS=SS+2↑I
14070 IF A$(SS)>T1$ THEN SS=SS-2↑I
14080 IF SS<0 THEN SS=0
14090 IF SS>IT-1 THEN SS=IT-1
14100 NEXT I:IF A$(SS)<T1$ THEN SS=SS+1
14110 RETURN
```

A standard binary search module as in Unifile.

MODULE 4.2.5

```
15000 REM#*******************************
15010 REM INSERT
15020 REM********************************
15030 IF IT=0 THEN GOTO 15050
15040 FOR I=IT TO SS+1 STEP -1:A$(I)=A$(
I-1):NEXT
15050 A$(SS)=T1$:IT=IT+1:RETURN
```

A straightforward insertion module.

*Testing Module 4.2.4 and 4.2.5*

You should now be able to enter items and have them saved in the main file array (A$). This can only be checked in direct mode.

MODULE 4.2.6

```
19000 REM#******************************
19010 REM ITEM TYPE NAMES
19020 REM*******************************
19030 FOR I=0 TO 49 STEP 10
19040 PRINT"■■■■■■■■■■■■■■■■ITEM NAMES
"
19050 PRINT "■";:FOR J=I TO I+10:PRINT J
+1;")";B$(J):NEXT J
19060 PRINT "■■COMMANDS AVAILABLE:"
19070 PRINT " ■>■'ZZZ'=QUIT"
19080 PRINT " ■>■'III'=ITEM/DELETE"
19090 PRINT " ■>■'NNN'=NEXT PAGE"
19100 INPUT "■WHICH DO YOU REQUIRE:";Q$:
IF Q$="ZZZ" THEN RETURN
19110 IF Q$="NNN" THEN NEXT I:RETURN
19120 IF Q$<>"III" THEN GOTO 19000
19130 INPUT "■POSITION NUMBER:";Q
19140 PRINT "■NAME OF TYPE OR ■RETURN■ T
O DELETE:"
19150 Q$="":INPUT Q$:B$(Q-1)=Q$:GOTO1904
0
```

This is a new module enabling the user to define item types. The module simply displays the contents of the array B$ in groups of 11 and gives the user the option to input a type name to a particular position in the array. Once entered, a type name can be attached to an item or input as described under Module 4. Type names can be redefined simply by entering a new name in the position occupied by an old one or deleted by pressing RETURN when asked for a type name.

*Testing Module 4.2.6*

Enter some type names then go back to the main input module and enter '↑' followed by the number of a type name you have defined. The prompt should be repeated under the desired type name.

MODULE 4.2.7

```
20000 REM#********************************
20010 REM DATA FILES
20020 REM*********************************
20030 PRINT "▓▓POSITION TAPE CORRECTLY,
THEN ▓ENTER▓--"
20040 INPUT "MOTOR WILL STOP AUTOMATICAL
LY:";Q$:POKE 192,7:POKE 1,39
20050 PRINT "▓▓COMMANDS AVAILABLE:":PRIN
T "▓▓ 1)SAVE DATA":PRINT " 2)LOAD DATA"
20060 INPUT "▓▓WHICH DO YOU REQUIRE:";Q:
ON Q GOTO 20070,20150:RETURN
20070 POKE 1,7:FOR I=1 TO 2000:NEXT
20080 OPEN 1,1,1,"UNIFILE":PRINT#1,IT
20090 FOR S1=0 TO IT-1:PRINT#1,FNA(S1)
20100 FOR I=1 TO FNA(S1)-1:PRINT#1,FNB(S
1):NEXT I
20110 PRINT#1,LEFT$(A$(S1),LEN(A$(S1))-F
NA(S1)):NEXT S1
20120 FOR I=0 TO 49:IF B$(I)="" THEN B$(
I)=" "
20130 PRINT#1,B$(I):NEXT
20140 CLOSE1:RETURN
20150 OPEN 1,1,0,"UNIFILE":INPUT#1,IT
20160 FOR S1=0 TO IT-1:INPUT#1,NN
20170 TT$="":FORI=1TO NN-1:INPUT#1,TT:TT
$=TT$+CHR$(TT):NEXTI:TT$=TT$+CHR$(NN-1)
20180 GET#1,T$:IF T$<>CHR$(13) THEN A$(S
1)=A$(S1)+T$:GOTO 20180
20185 A$(S1)=A$(S1)+TT$:NEXT S1
20190 FOR I=0 TO 49:INPUT#1,B$(I):NEXT
20200 CLOSE1:RETURN
```

A standard data-file module.

MODULE 4.2.8

```
21000 REM#********************************
21010 FUNCTIONAL SUBROUTINES
21020 REM*********************************
21030 SS$=MID$(A$(S2),PP+1,FNB(S2)-PP):R
ETURN
21040 FF=0:T1$=RIGHT$(T1$,LEN(T1$)-3):FO
R S2= S1TO IT-1:FOR J=1 TO LEN(A$(I))
```

```
21050 IF MID$(A$(S2),J,LEN(T1$))=T1$ THE
N FF=1:S1=S2:J=LEN(A$(S2)):S2=IT-1
21060 NEXT J,S2:RETURN
21070 FF=0:FOR S2=S1 TO IT-1:PP=0:FOR I=
1 TO FNA(S2)-1:GOSUB 21030
21080 IF SS$=T1$ THEN FF=1:S1=S2:I=FNA(S
2)-1:S2=IT-1:GOTO 21120
21090 IF LEN(SS$)<4 THEN 21110
21100 IF MID$(SS$,LEN(SS$)-2,1)="↑" THEN
 SS$=LEFT$(SS$,LEN(SS$)-3):GOTO 21080
21110 PP=FNB(S2)
21120 NEXT I,S2:RETURN
```

The module consists of three routines which are more economically placed here since they are called by different parts of the program.

*Commentary*

Line 21030: This line can be called from within two loops: an S2 loo specifying the line in the main file and an I loop specifying the item number within the particular entry. It then extracts, in the form of SS$, item I of entry S2.

Lines 21040–21060: Equivalent to the special search routine in Unifile.

Lines 21070–21120: A straightforward item search. The I loop uses FNA to discover how many items there are in the entry (FNA(S2) = the number of items there are in entry S2 plus 1 for the indicator at the end) and then calls up 21030 to extract the individual items. Items with type indicator are compared with and without the type suffix.

*Testing Module 4.2.8*

The module cannot be checked until the following module has been entered.

MODULE 4.2.9

```
17000 REM#******************************
17010 REM SEARCH
17020 REM*******************************
17030 S1=0:FF=0:PRINT "█████████████████
██SEARCH"
17040 PRINT "██COMMANDS AVAILABLE:"
17050 PRINT "█ █>█INPUT ITEM FOR NORMAL
SEARCH"
17060 PRINT " █>█PRECEDE WITH 'III' FOR
INITIAL SEARCH"
```

```
17070 PRINT " ■>■PRECEDE WITH 'SSS' FOR
SPECIAL SEARCH"
17080 PRINT " ■>■■ENTER■ FOR FIRST ITEM
ON FILE"
17090 PRINT " ■>■'MMM' FOR MULTIPLE SEAR
CH"
17100 PRINT "■■■●●●●●●●●●●●●●●●●●●●●●●●●●
●●●●●●●●●●●●●●●●T]"
17110 T1$="":INPUT "■SEARCH COMMAND:";T1
$:IF LEFT$(T1$,1)<>"↑"THEN 17130
17120 LET TN=VAL(MID$(T1$,2))+10:GOTO 17
110
17130 IF TN<>0 THEN LET T1$=T1$+"↑"+MID$
(STR$(TN),2):TN=0
17140 IF LEFT$(T1$,3)<>"III" THEN 17170
17150 T1$=RIGHT$(T1$,LEN(T1$)-3):GOSUB 1
4000:S1=SS:IF S1>IT-1 THEN RETURN
17160 T1$="III"+T1$:GOTO 17310
17170 IF LEFT$(T1$,3)<>"SSS" THEN 17200
17180 GOSUB 21040:IF FF=1 THEN T1$="SSS"
+T1$:GOTO 17310
17190 RETURN
17200 IF LEFT$(T1$,3)<>"MMM" THEN 17270
17210 GETZ$:INPUT "■■NUMBER OF ITEMS TO
SEARCH FOR:";NN
17220 FOR K=0 TO NN-1:PRINT "SEARCH ITEM
";K+1;:INPUT M$(K):NEXT K
17230 FOR K=0 TO NN-1:T1$=M$(K):S3=S1:GO
SUB21070:IF FF=0 THEN RETURN
17240 IF S3<>S1 THEN 17230
17250 NEXT K
17260 LET T1$="MMM":GOTO 17310
17270 IF T1$="" THEN 17310
17280 GOSUB 21070
17290 IF FF=1 THEN 17310
17300 RETURN
17310 IF S1>IT-1 THEN S1=IT-1
17320 IF IT=0 THEN RETURN
17330 IF S1<0 THEN S1=0
17340 PRINT "■■■ENTRY ";S1+1;":-■":PP=0:
S2=S1:FOR I=1 TO FNA(S1)-1
17345 IF I/12=INT(I/12) THEN INPUT "■MOR
E";TT$
17350 GOSUB 21030:IF LEN(SS$)<4 THEN 173
80
```

```
17360 IF MID$(RIGHT$(SS$,3),1,1)<>"↑" TH
EN 17380
17370 PRINT B$(VAL(RIGHT$(SS$,2))-11);":
";:SS$=LEFT$(SS$,LEN(SS$)-3)
17380 PRINT SS$
17390 PP=FNB(S1):NEXT I:S1=S1+1
17400 PRINT "▒▒▒SEARCH█ ▒COMMANDS AVAILA
BLE:"
17410 PRINT "▒ █>▒▒ENTER█ FOR NEXT ITEM"
17420 PRINT " █>▒'AAA' TO AMEND'
17430 PRINT " █>▒'CCC' TO CONTINUE SEARC
H"
17440 PRINT " █>▒'#' FOLLOWED BY NO. TO
MOVE POINTER"
17450 PRINT " █>▒'ZZZ' TO QUIT FUNCTION'
17460 P$="":INPUT "▒▒WHICH DO YOU REQUIR
E:";P$
17470 IF T1$="MMM" AND S1<IT THEN 17230
17480 IF P$="CCC" AND S1<IT THEN 17140
17490 IF P$="" THEN 17310
17500 IF P$="AAA" THEN GOSUB 16000:GOTO
17310
17510 IF P$="ZZZ" THEN RETURN
17520 IF LEFT$(P$,1)="#" THEN S1=S1+VAL(
MID$(P$,2))-1:GOTO 17310
17530 S1=S1-1:GOTO 17310
```

Similar to the Search module in Unifile, but making provision for the multiple search and for type names.

*Commentary*

Lines 17110–17130: Note the way in which this routine detects whether an item with a type number attached is being searched for and then requests input under that type heading, tagging the type number onto the end of the item.

Lines 17140–17160: An initial search as in Unifile.

Lines 17170–17190: Special search making use of the routine in the previous module.

Lines 17200–17260: The new multiple search routine. It requests the user to specify the number of items to be searched for, then to input the individual item (type numbers are not dealt with). A search routine is called up at 21070–21120. Before each search item is specified, a record is taken in the form of the variable S3 of the value of the search pointer S1. When the routine at 21070 returns to this routine the value of S3 is again compared with S1. If S1 is different from S3 then it is clear that the two items were not present in the same entry. On first finding one of the specified search items, the search is reset to the first of the specified search items in order to ensure that the whole list of search items is compared with the items in the entry.

Line 17280: If the search has reached this point the input is assumed to be an item to be searched for with a normal search and the search routine at 21070 is called up. Note the use of the flag FF in all these search routines to indicate whether something has in fact been found.

Lines 17340–17390: Starting at the beginning of the chosen record, the routine at 21030 is called up to extract individual items. Type names are printed where the '↑' symbol is present 2 characters from the end of the item.

*Testing Module 4.2.9*

You should now be able to page backwards and forwards through the entries and search using the methods described in the commentary.

MODULE 4.2.10

```
18000 REM#*******************************
18010 REM TELESCOPE FILE
18020 REM********************************
18030 FOR I=S1 TO IT-1:A$(I)=A$(I+1):NEX
T:IT=IT-1:RETURN
```

This one line routine telescopes the file when deletions are being made.

MODULE 4.2.11

```
16000 REM#*******************************
16010 REM CHANGE ENTRIES
16020 REM********************************
16030 S1=S1-1:T1$="":NN=-1:TN=0
16040 PP=0:S2=S1:FOR I=1 TO FNA(S1)-1
16050 PRINT "□■ENTRY ";S1+1;":-"
16060 GOSUB 21030:IF LEN(SS$)<4 THEN 160
90
16070 IF MID$(SS$,LEN(SS$)-2,1)<>"↑" THE
N 16090
16080 PRINT B$(VAL(RIGHT$(SS$,2))-11);":
";LEFT$(SS$,LEN(SS$)-3):GOTO 16100
16090 PRINT SS$
16100 PRINT"▩■■■■■■COMMANDS AVAILABLE:"
```

```
16110 PRINT " ■>■■RETURN■ LEAVES ITEM UN
CHANGED"
16120 PRINT " ■>■NEW ITEM ENDING WITH '*
'"
16130 PRINT " ■>■REPLACEMENT OF ITEM DIS
PLAYED"
16140 PRINT " ■>■'ZZZ' QUITS WITHOUT CHA
NGES"
16150 PRINT " ■>■'DDD' DELETES THE WHOLE
 ENTRY"
16160 PRINT " ■>■'RRR' REMOVES THIS ITEM
 FROM ENTRY"
16170 Q$="":INPUT "■WHICH DO YOU REQUIRE
:";Q$
16180 IF Q$="ZZZ" THEN RETURN
16190 IF Q$="RRR" THEN GOTO 16300
16200 IF Q$="DDD" THEN GOSUB 18000:RETUR
N
16210 IF LEN(T1$)+LEN(Q$)+NN+2<255 THEN
GOTO 16230
16220 PRINT "■ENTRY TOO LONG":FOR J=1 TO
 3000:NEXT:RETURN
16230 EX=0:IF RIGHT$(Q$,1)="*" THEN EX=1
:Q$=LEFT$(Q$,LEN(Q$)-1)
16240 IF Q$="" THEN T1$=T1$+S$$:TI%(NN+1
)=LEN(T1$):NN=NN+1:GOTO 16300
16250 IF LEFT$(Q$,1)<>"↑" THEN 16280
16260 TN=VAL(MID$(Q$,2))+10:PRINT B$(TN-
11);":";:Q$="":INPUT Q$
16270 Q$=Q$+"↑"+MID$(STR$(TN),2)
16280 T1$=T1$+Q$:TI%(NN+1)=LEN(T1$):NN=N
N+1
16290 IF EX=1 THEN 16050
16300 PP=FNB(S1):NEXT I
16310 FOR I=0 TO NN:T1$=T1$+CHR$(TI%(I))
:NEXT:T1$=T1$+CHR$(NN+1)
16320 PRINT "■■WAIT":GOSUB 18000:GOSUB 1
4000:GOSUB 15000:RETURN
```

Equivalent to the Unifile change module.

*Commentary*

Line 16230: Entering an item ending with a * indicates that a new item is to be placed into the entry before the item currently on display. The variable EX (EXtra) is set to one to show that this has happened. In Line 16290 this variable is used to ensure that the item on display is not lost.

*Testing Module 4.2.11*

You should now be able to delete entries, to change items or to insert items. If this module is functioning correctly then the program is ready for use.

***Summary***

Given that their applications will be different, this program has all the strengths of the original Unifile program and I hope you find it useful.

In addition, I hope that entering the program has given you some insights into the advantages of a modular style of programming. On the basis of the original Unifile modules the original version of this program took less than a single morning to write for the simple reason that the clear structure provided by a modular program makes it absolutely clear where any necessary changes have to be made.

Provided that you are not absolutely desperate for memory space, you will save time and tears in your programming by setting out your programs in clearly defined functional units. Not only does this make the programs more readable, it increases the likelihood that you will be able to call the same routine from different parts of the program, eases replacement of functions you think you can improve on at a later date and, not least, makes it easier to lift whole sections out of the program for subsequent use in other applications.

***Going Further***

1) The multiple search routine makes no provision for specifying the type names of items. Lines 17110–17130 provide a clear example of how such an ability could be provided.

2) Professional databases usually have the ability to search for entries which have, say, four out of eight search targets present. Could you adapt the present program to achieve the same.

## 4.3 NNUMBER

Having entered two programs which are capable of dealing with a variety of needs in the field of storing non-numeric data, we turn our attention now to the problems of keeping track of numbers. Although most numeric applications need to be specifically addressed to a particular problem, NNumber (short for Name and Number) is very much like the two Unifile programs in that it is intended to be a general purpose tool for applications where you need to store the names of items, units of quantity associated with them and to be able to add together the items in varying quantities. In case it seems to you that you never want to do that, perhaps I should say that the idea the program grew out of was a calorie counter that enabled the user to store a dictionary of up to 500 foods and to calculate with ease the calorific cost of a day's or a week's meals. The present program is just as capable, without alteration, of calculating invoices as it is of helping with weight watching.

Because the style of the program is very similar to the two Unifiles, and many of the functions similar, comments and testing suggestions have been abbreviated as much as possible.

**NNumber: Table of variables**

| | |
|---|---|
| A$(499,1) | Main dictionary array. |
| C(499) | Values associated with units specified in main dictionary. |
| CT | Temporary variable used to cumulate the sum of items in current list. |
| CU | Number of items in current list. |
| NN$ | General name for items being recorded. |
| IN | Initialisation flag. |
| IT | Number of items stored in main dictionary. |
| NN | Temporary storage for value associated with new item being input. |
| PO | Used to determine number of comparisons in binary search module. |
| QQ$ | General name for quantities associated with items being recorded. |
| R$ | Separator used in data file saving. |
| SS | Search pointer for binary search. |
| T(49) | Values associated with items stored in T$. |
| T$(49,1) | Storage of current list. |
| T1$ | Temporary storage of item name being input. |
| T2$ | Temporary storage of units associated with T1$. |

MODULE 4.3.1

```
11000 REM#*******************************
11010 REM MENU
11020 REM********************************
11030 POKE 53281,13:PRINT "[illegible]
[illegible]NAME AND NUMBER[illegible]"
11040 PRINT "[illegible]COMMANDS AVAILABLE:"
```

```
11050 PRINT "[illegible]   1)DISPLAY CURRENT LIS
T"
11060 PRINT "   2)INPUT TO CURRENT LIST"
11070 PRINT "   3)START FRESH LIST"
11080 PRINT "   4)DELETE FROM CURRENT LI
ST"
11090 PRINT "   5)EXTEND DICTIONARY"
11100 PRINT "   6)DISPLAY DICTIONARY"
11110 PRINT "   7)DATA FILES"
11120 PRINT "   8)INITIALISE"
11130 PRINT "   9)STOP"
11140 INPUT "[illegible]WHICH DO YOU REQUIRE:";Z:
PRINT "[illegible]";
11150 IF IN<>0 OR Z=8 OR Z=9 THEN GOTO 1
1180
11160 PRINT "[illegible]NOT I
NITIALISED YET"
11170 FOR I=1 TO 2000:NEXT:GOTO 11000
11180 IF IT>0 OR Z=2 OR Z=3 OR Z=5 OR Z=
7 OR Z=8 OR Z=9 THEN 11210
11190 PRINT "[illegible]NO
 DATA YET"
11200 FOR I=1 TO 2000:NEXT:GOTO 11000
11210 ON Z GOSUB 13000,14000,14120,19000
,15000,18000,20000,12000,11230
11220 GOTO 11000
11230 PRINT "[illegible]NA
ME AND NUMBER"
11240 PRINT "[illegible]PROGRAM TERMI
NATED":END
```

Standard menu module.

MODULE 4.3.2

```
12000 REM********************************
12010 REM INITIALISE
12020 REM********************************
12030 CLR:DIM A$(499,1),C(499),T$(49,1),
T(49):CURR=0:IT=0:IN=1:R$=CHR$(13)
12035 INPUT "[illegible]ARE YOU LOADING FROM TAPE
 (Y/N):";Q$:IF Q$="Y" THEN 11000
12040 INPUT "[illegible]NAME FOR ITEMS:";NN$
12050 INPUT "[illegible]NAME FOR ASSOCIATED QUANTI
TY:";QQ$:GOTO 11000
```

Initialisation of variables. The module also allows the user to specify the name of the item type to be stored and the general name of the units e.g. Food/Units, Product/Package type.

MODULE 4.3.3

```
15000 REM#*****************************
15010 REM EXTEND DICTIONARY
15020 REM******************************
15030 IF ITEMS<500 THEN 15050
15040 PRINT "NO MORE ROOM IN DICTIO
NARY":FOR I=1 TO 2000:NEXT:RETURN
15050 PRINT "NEW ITEMS FOR DI
CTIONARY"
15060 PRINT "";NN$;:INPUT ":(NAME OR
'ZZZ' TO QUIT):";T1$
15070 IF T1$="ZZZ" THEN RETURN
15080 PRINT "";QQ$;:INPUT ":";T2$
15090 PRINT "QUANTITY PER ";T2$;:INPUT
NN
15100 INPUT "ARE THESE CORRECT (Y/N):"
;Q$:IF Q$="N" THEN GOTO 15050
15110 GOSUB 16000:GOSUB 17000:IT=IT+1:GO
TO 15050
```

The input module for the main dictionary. Having specified the general name for the items and their units, the user is requested to input item name, unit name and basic quantity per unit (ie calories, price, volume).

MODULE 4.3.4

```
16000 REM#*****************************
16010 REM BINARY SEARCH
16020 REM******************************
16030 IF IT=0 THEN SS=0:RETURN
16040 PO=INT(LOG(IT)/LOG(2)):SS=2↑PO-1
16050 FOR I=PO TO 0 STEP -1
16060 IF A$(SS,0)<T1$ THEN SS=SS+2↑I
16070 IF A$(SS,0)>T1$ THEN SS=SS-2↑I
16080 IF SS<0 THEN SS=0
16090 IF SS>IT-1 THEN SS=IT-1
16100 NEXT I
```

```
16110 IF A$(SS,0)<T1$ THEN SS=SS+1
16120 RETURN
```

A standard binary search module.

MODULE 4.3.5

```
17000 REM#*******************************
17010 REM INSERT ITEM
17020 REM********************************
17030 IF IT=0 THEN GOTO 17060
17040 FOR I=IT TO SS+1 STEP-1:A$(I,0)=A$
(I-1,0):A$(I,1)=A$(I-1,1)
17050 C(I)=C(I-1):NEXT
17060 A$(SS,0)=T1$:A$(SS,1)=T2$:C(SS)=NN
:RETURN
```

A standard insertion module.

MODULE 4.3.6

```
18000 REM#*******************************
18010 REM USER SEARCH
18020 REM********************************
18030 SS=0:T1$="*1"
18040 PRINT "██████████████████SEARCH█"
18050 PRINT "██ITEM NUMBER:";SS+1
18060 PRINT "██";NN$;":";A$(SS,0)
18070 PRINT "█";QQ$;":";A$(SS,1)
18080 PRINT "█QUANTITY PER ";A$(SS,1);":
";C(SS)
18090 PRINT "██●●●●●●●●●●●●●●●●●●●●●
●●●●●●●●●●●●●●●●"
18100 PRINT "█COMMANDS AVAILABLE:"
18110 PRINT "███   >ITEM TO BE SEARCHED F
OR"
18120 PRINT "    >'*' THEN NUMBER TO MOVE
 POINTER"
18130 PRINT "    >'DDD' TO DELETE ITEM"
18140 PRINT "    >'ZZZ' TO QUIT"
18150 T1$="":INPUT "███WHICH DO YOU REQUI
RE:";T1$
18160 IF T1$<>"DDD" THEN 18190
18170 FOR I=SS TO IT-1:A$(I,0)=A$(I+1,0)
:A$(I,1)=A$(I+1,1):C(I)=C(I+1):NEXT
```

```
18180 IT=IT-1:GOTO 18040
18190 IF T1$="ZZZ" THEN RETURN
18200 IF LEFT$(T1$,1)<>"*" THEN GOSUB 16
000:GOTO 18220
18210 SS=SS+VAL(MID$(T1$,2))
18220 IF SS>IT-1 THEN SS=IT-1
18230 IF SS<0 THEN SS=0
18240 GOTO 18040
```

The main user search module.

*Testing Modules 4.3.1–4.3.6*

You should now be able to enter data and to check its proper insertion into the main dictionary (sorted by item name) by using the search module.

MODULE 4.3.7

```
20000 REM******************************
20010 REM DATA FILES
20020 REM******************************
20030 PRINT "■■POSITION TAPE CORRECTLY,
THEN ■RETURN■——"
20040 INPUT "MOTOR WILL STOP AUTOMATICAL
LY:";Q$:POKE 192,7:POKE 1,39
20050 PRINT "■■COMMANDS AVAILABLE:":PRIN
T "■■1)SAVE DATA":PRINT "■2)LOAD DATA"
20060 INPUT "■■WHICH DO YOU REQUIRE:";Q:
 ON Q GOTO 20070,20140:RETURN
20070 POKE 1,7:FOR I=1 TO 2000:NEXT
20080 OPEN 1,1,2,"NNUMBER":PRINT#1,NN$,R
$,QQ$,R$,CU,R$,IT
20090 IF CU=0 THEN 20110
20100 FOR I=0 TO CU-1:PRINT#1,T$(I,0),R$
,T$(I,1),R$,T(I):NEXT
20110 IF IT=0 THEN 20130
20120 FOR I=0 TO IT-1:PRINT#1,A$(I,0),R$
,A$(I,1),R$,C(I):NEXT
20130 CLOSE1:RETURN
20140 OPEN 1,1,0,"NNUMBER":INPUT#1,NN$,Q
Q$,CU,IT
20150 IF CU=0 THEN 20170
20160 FOR I=0 TO CU-1:INPUT#1,T$(I,0),T$
(I,1),T(I):NEXT
```

```
20170 IF IT=0 THEN 20190
20180 FOR I=0 TO IT-1:INPUT#1,A$(I,0),A$
(I,1),C(I):NEXT
20190 CLOSE1:RETURN
```

A standard data-file module. Save your data!

MODULE 4.3.8

```
14000 REM#********************************
14010 REM EXTEND CURRENT LIST
14020 REM*********************************
14030 IF CU=50 THEN PRINT "██████████████
████CURRENT LIST NOW FULL.":RETURN
14040 PRINT "██████████ADDITIONS TO CURR
ENT LIST█"
14050 PRINT "███";NN$;" ('ZZZ' TO QUIT):
";:INPUT T1$:IF T1$="ZZZ" THEN RETURN
14060 GOSUB 16000:IF A$(SS,0)=T1$ THEN G
OTO 14080
14070 PRINT "███FOOD UNKNOWN, PLEASE CHE
CK.":FOR I=1 TO 2000:NEXT:RETURN
14080 PRINT "█";QQ$;":";A$(SS,1):INPUT "
█QUANTITY:";Q
14090 INPUT "███ARE THESE CORRECT (Y/N):
";Q$:IF Q$="N" THEN 14000
14100 T$(CU,0)=A$(SS,0):T$(CU,1)=STR$(Q)
+" "+A$(SS,1):T(CU)=Q*C(SS):CU=CU+1
14110 GOTO 14000
```

Up to this point you have been able to input to the main dictionary and manipulate the data but the main point of the program is what can be extracted from the dictionary and stored in a temporary list known as the current list. This module begins the process by allowing the user to name an item in the dictionary and to specify how many units of that item are to be added to the current list. The binary search module is called up to check that the item input is actually present in the dictionary.

MODULE 4.3.9

```
13000 REM#********************************
13010 REM DISPLAY CURRENT LIST
13020 REM*********************************
```

```
13030 IF CU=0 THEN RETURN
13040 PRINT"□":CT=0:FOR I=0 TO CU-1:PRIN
T "▣▣";NN$;":";T$(I,0)
13050 PRINT "▣";QQ$;":";T$(I,1):PRINT "▣
QUANTITY:";T(I)
13060 PRINT "▣▣OOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOO"
13070 INPUT "PRESS ▣RETURN▣ FOR NEXT ITE
M:";Q$:PRINT "TT"
13080 PRINT "
             TTT"
13090 CT=CT+T(I):NEXT I
13100 PRINT "▣▣TOTAL:▣";CT
13110 INPUT "▣▣PRESS ▣RETURN▣ TO GO BACK
 TO MENU";Q$:RETURN
```

Having input items to the current list, this module allows the current list to be output to the screen and creates a total of the values associated with the items in the current list.

MODULE 4.3.10

```
19000 REM********************************
19010 REM CURRENT LIST DELETIONS
19020 REM********************************
19030 IF CU=0 THEN RETURN
19040 FOR I=0 TO CU-1:Q$="":PRINT "▣▣";T
$(I,0):PRINT "▣";T$(I,1)
19050 INPUT "▣▣DDD=DELETE▣ ▣RETURN=NEXT▣
 ▣ZZZ=QUIT:";Q$:IF Q$="ZZZ" THEN RETURN
19060 IF Q$<>"DDD" THEN NEXT I:RETURN
19070 FOR J=I TO CU-1:T$(J,0)=T$(J+1,0):
T$(J,1)=T$(J+1,1):T(J)=T(J+1):NEXT
19080 CU=CU-1:RETURN
19090 STOP
```

The purpose of this module is to allw the user to page through the entries to the current list and to delete at will. There are no complex search functions, it is simply a matter of paging through each item, one at a time.

### *Testing Modules 4.3.8–4.3.10*

You should now be able to set up a current list, to display it and delete from it at will.

MODULE 4.3.11

```
14120 REM #*****************************
14130 REM INITIALISE CURRENT LIST
14140 REM ******************************
14150 FOR I=1 TO 50:T$(I,0)="":T$(I,1)="
":T(I)=0:CU=0:RETURN
```

Since the current list is only meant to be a temporary one, which may be reset frequently, this module empties the current list arrays and zeros the current list pointer. If the module functions correctly, the program is ready for use.

***Summary***

This program is yet another example of the power of modular programming. Despite the fact that the application is very different, many of the modules have been lifted, with or without modification, from the previous two programs.

As you progress as a programmer you will quickly learn that, written into functional units which are properly separated from each other, a collection of methods of doing things is even more important than a collection of programs. A library of programs will stand you in good stead until a new application comes along. A library of methods, properly expressed in working modules, will enable you to tackle those new applications with hardly any effort at all. New methods are all around you in magazines and books like this one (of course there are no books quite like this one) so do try to keep track of them if they look good, even if you can't quite see their relevance at the moment. Within a week or two you may well find that they are just what you are looking for to complete that program that is giving you so much trouble.....

# CHAPTER 5
# Home Education

One field where micro computers are really only just beginning to make their mark is that of education. No school is complete today without one or two computers scattered around. But it's not just in the classroom that computer-aided learning is relevant—affordable computing power means that all the benefits can be brought home. In this chapter there are three programs which provide a sampler of educational applications, whatever your age.

## 5.1 MULTIQ

This program is a favourite of mine. When I wrote it I was satisfied that it was a competent piece of work that would do the job that it was designed for. It was not until I entered a mass of questions and tried it out on people that I realised that such programs make learning as addictive as any game.

Like Unifile, this program is a chameleon, designed to change its colour to suit your need. At one moment it may be a French tutor, a few minutes later you may have it setting complex questions on 19th century history. The aim of the program is to allow you to do all this and more, without making any changes in the program itself.

**Multiq: Table of variables**

| | |
|---|---|
| AA | Temporary variable storing answer selected by user. |
| A$(1,499) | Main array containing questions and answers. |
| D(1,9) | Pointer to beginning of groups of item types in main array. |
| D$(9) | Array containing item types. |
| IT | Number of items stored in main array. |
| NA$(1) | Array containing general names for questions and answers. |
| P1,P2 | Pointers to range of files to be drawn on to generate questions. |
| PP | Pointer to wrong answer being selected from array. |
| Q(4) | Used in setting up multiple choice test. |
| Q1 | Position of random answer drawn from file. |
| Q2 | Position of correct answer in possible answer array Q. |

| | |
|---|---|
| QT | Total questions asked. |
| QU | Indicator to type of questions required in question generating module. |
| R$ | Data file separator. |
| RR | Right answers. |
| SU | Temporary variable used in calculating groups in array D. |
| TY | Number of type names entered. |

MODULE 5.1.1

```
11000 REM#********************************
11010 REM MENU
11020 REM*********************************
11030 POKE 53281,15:PRINT "■■■■■■■■■■■■■■
■■MULTIQ"
11040 PRINT "■COMMANDS AVAILABLE:"
11050 PRINT "■ ■1)INPUT NEW ITEMS"
11060 PRINT " 2)SEARCH/DELETE"
11070 PRINT " 3)ENTER NEW TYPES"
11080 PRINT " 4)GENERATE QUESTIONS"
11090 PRINT " 5)DISPLAY OR RESET SCORE"
11100 PRINT " 6)DATA FILES"
11110 PRINT " 7)INITIALISE"
11120 PRINT " 8)STOP"
11130 INPUT "■■WHICH DO YOU REQUIRE:";Z:
PRINT "■";
11140 ON Z GOSUB 13000,16000,12100,17000
,18000,19000,12000,11150:GOTO 11000
11150 PRINT "■■■■■■■■■■■■■■■■■■■■■■■■■■■
CLASSROOM CLOSED":END
```

A standard menu module.

MODULE 5.1.2

```
12000 REM#********************************
12010 REM INITIALISE
12020 REM*********************************
12030 CLR:DIM NA$(1),Q(4),A$(1,499),D$(9
),D(1,9):R$=CHR$(13)
12040 INPUT "■■ARE YOU LOADING FROM TAPE
 (Y/N):";Q$:IF Q$="Y" THEN 11000
12050 PRINT "■■■■■■■■■■■■■■■TEST STRUCTUR
E"
```

```
12060 INPUT "■■NAME FOR ANSWER:";NA$(0)
12070 INPUT "■■NAME FOR QUESTION:";NA$(1
)
12080 INPUT "■■ARE THESE CORRECT (Y/N):"
;Q$:IF Q$="N" THEN 12050
12090 D$(0)="UNTYPED":TY=1
12100 PRINT "■■■■■■■■■■■■■■■■■■TYPES"
12110 PRINT "■■INPUT 'ZZZ' TO QUIT:"
12120 PRINT "■■TYPES INPUT SO FAR:-";:IF
 TY=1 THEN PRINT "■NONE"
12130 IF TY<>1 THEN PRINT "■":FOR I=0 TO
 TY-1:PRINT "■";I+1;"- ■";D$(I):NEXT
12140 INPUT "■■NEW TYPE:";Q$:IF Q$="ZZZ"
 THEN 11000
12150 INPUT "■■IS THIS CORRECT (Y/N):";Q
1$:IF Q1$="N" THEN 12100
12160 IF TY=10 THEN PRINT "■■NO MORE ROO
M":FOR I=1 TO 2000:NEXT:GOTO 11000
12170 D$(TY)=Q$:TY=TY+1:GOTO 12100
```

This module initialises the main variables and allows the user to specify the general name for questions and answers. The user may also specify up to nine type names which will be used later on to make the tests more difficult.

MODULE 5.1.3

```
13000 REM#******************************
13010 REM INPUT OF NEW ITEMS
13020 REM*******************************
13030 PRINT "■■■■■■■■■■■■■■■■■■NEW ITEMS"
:PRINT "■■'ZZZ' TO QUIT"
13040 IF IT>=500 THEN PRINT "■■NO MORE R
OOM:FOR I=1 TO 2000:NEXT:RETURN
13050 PRINT "■■";NA$(0);":";:INPUT T1$:I
F T1$="ZZZ" THEN 13160
13060 PRINT "■■";NA$(1);":";:INPUT T2$:I
F T2$="ZZZ" THEN RETURN
13070 IF TY=1 THEN T=0:GOTO 13100
13080 PRINT "■TYPE:":FOR I=1 TO TY:PRINT
 I;D$(I-1):NEXT I
13090 INPUT "■■WHICH IS IT:";T:T=T-1
13100 PRINT "■■■■■■■■■■■■■■■■■■NEW ITEMS"
```

```
13110 PRINT "■■■";NA$(0);"■";T1$:PRINT "
■■■";NA$(1);"■";T2$
13120 PRINT "■■■TYPE■";D$(T):INPUT "■■AR
E THESE CORRECT (Y/N):";Q$
13130 IF Q$<>"Y" THEN 13000
13140 D(0,T)=D(0,T)+1:T1$=CHR$(48+T)+T1$
13150 GOSUB 14000:GOSUB 15020:GOTO13000
13160 SU=0:FOR I=0 TO 9:D(1,I)=SU:SU=SU+
D(0,I):NEXT:RETURN
```

This module accepts input of questions and answers under the headings specified by the user, allowing a type to be attached if types have been entered.

*Commentary*

Line 13070: If TY is equal to 1, no type names have been entered and the type is set to 'Untyped'.

Line 13140: The relevant element in the first column of the array D is increased by 1, registering the fact that the type group has increased. The type number is attached to the front of the answer in the form of a character between 0 and 9.

Line 13160: The second column of the array D is adjusted so that it contains the start position of each type group.

MODULE 5.1.4

```
14000 REM#*******************************
14010 REM BINARY SEARCH
14020 REM********************************
14030 IF IT=0 THEN SS=0:RETURN
14040 PO=INT(LOG(IT)/LOG(2)):SS=2↑PO-1
14050 FOR I=PO TO 0 STEP-1
14060 IF A$(0,SS)<T1$ THEN SS=SS+2↑I
14070 IF A$(0,SS)>T1$ THEN SS=SS-2↑I
14080 IF SS<0 THEN SS=0
14090 IF SS>IT-1 THEN SS=IT-1
14100 NEXT I:IF A$(0,SS)<T1$ THEN SS=SS+
1
14110 RETURN
```

A standard binary search module. Note that since the items are sorted by answer and the answers have the type group character attached to the front, the actual sort is first of all by type—untyped groups will occur first in the file.

MODULE 5.1.5

```
15000 REM#*****************************
15010 REM INSERT
15020 REM******************************
15030 IF IT=0 THEN 15060
15040 FOR I=IT TO SS+1 STEP -1
15050 FOR J=0 TO 1:A$(J,I)=A$(J,I-1):NEX
T J,I
15060 A$(0,SS)=T1$:A$(1,SS)=T2$:IT=IT+1:
RETURN
```

A standard insertion module.

*Testing Modules 5.1.1–5.1.5*

You should now be able to enter data and store it in the main arrays, sorted by type number.

MODULE 5.1.6

```
19000 REM#*****************************
19010 REM DATA FILES
19020 REM******************************
19030 PRINT "■■POSITION TAPE CORRECTLY T
HEN ■RETURN■--"
19040 INPUT "MOTOR WILL STOP AUTOMATICAL
LY:";Q$:POKE 192,7:POKE 1,39
19050 PRINT "■■COMMANDS AVAILABLE:":PRIN
T "■ 1)SAVE DATA",,," 2)LOAD DATA"
19060 INPUT "■WHICH DO YOU REQUIRE:";Q:O
N Q GOTO 19070,19130:RETURN
19070 POKE 1,7:FOR I=1 TO 2000:NEXT
19080 OPEN1,1,2,"MULTIQ":PRINT#1,IT;R$;T
Y
19090 FOR I=0 TO TY-1:PRINT#1,D$(I);R$;D
(0,I);R$;D(1,I):NEXT
19100 FOR I=0 TO IT-1:PRINT#1,A$(0,I);R$
;A$(1,I):NEXT
19110 PRINT#1,NA$(0);R$;NA$(1)
19120 CLOSE1:RETURN
19130 OPEN1,1,0,"MULTIQ":INPUT#1,IT,TY
19140 FOR I=0 TO TY-1:INPUT#1,D$(I),D(0,
I),D(1,I):NEXT
19150 FOR I=0 TO IT-1:INPUT#1,A$(0,I),A$
(1,I):NEXT
19160 INPUT#1,NA$(0),NA$(1)
19170 CLOSE1:RETURN
```

A standard data-file module.

MODULE 5.1.7

```
16000 REM********************************
16010 REM USER SEARCH/DELETE
16020 REM********************************
16030 SS=0
16040 PRINT "█████████████████SEARCH"
16050 IF SS>IT-1 THEN SS=IT-1
16060 IF SS<0 THEN SS=0
16070 PRINT "███████████████████████████
ITEMS=";IT-1
16080 PRINT "██COMMANDS AVAILABLE:"
16090 PRINT "██ >██RETURN█ FOR NEXT ITEM
"
16100 PRINT "█ >█POS/NEG NUMBER TO MOVE
POINTER"
16110 PRINT "█ >█'DDD' TO DELETE ITEM"
16120 PRINT "█ >█'ZZZ' TO QUIT FUNCTION"
16130 PRINT "█";:FOR I=1 TO 10
16140 PRINT "██
                    ";:NEXT
16150 PRINT "█IIIIIIIIIIENTRY NO:-";SS+1:
PRINT "██";MID$(A$(0,SS),2)
16160 PRINT "██";A$(1,SS):PRINT "██";D$(
VAL(LEFT$(A$(0,SS),1)))
16170 Q1$="":INPUT"████WHICH DO YOU REQU
IRE:";Q1$
16180 IF Q1$<>"DDD" THEN 16210
16190 D(0,TEMP)=D(0,TEMP)-1:FOR I=SS TO
IT-1:A$(0,I)=A$(0,I+1)
16200 A$(1,I)=A$(1,I+1):NEXT I:IT=IT-1:G
OSUB 13160:GOTO 16040
16210 IF Q1$="ZZZ" THEN RETURN
16220 IF Q1$="" THEN SS=SS+1:GOTO 16040
16230 SS=SS+VAL(Q1$):GOTO 16040
16240 GOTO 16240
```

A straightforward user search module.

MODULE 5.1.8

```
17000 REM#******************************
17010 REM RANDOM QUESTIONS
17020 REM*******************************
17030 QU=0
17040 PRINT "QUESTIONS"
17050 PRINT "DO YOU WISH ANSWERS TO B
E DRAWN ONLY"
17060 INPUT "FROM THE SAME QUESTION TYPE
 (Y/N):";Q$:PRINT "";
17070 IF Q$="Y" THEN QU=1
17080 P1=0:P2=IT:Q1=INT(RND(0)*IT-1):Q2=
INT(RND(0)*5):Q(Q2)=Q1
17090 IF QU=0 OR D(0,VAL(LEFT$(A$(0,Q1),
1)))<5 THEN 17110
17100 P1=D(1,VAL(LEFT$(A$(0,Q1),1))):P2=
D(0,VAL(LEFT$(A$(0,Q1),1)))
17110 FOR I=0 TO 4:IF I=Q2 THEN 17150
17120 PP=P1+INT(RND(0)*P2):IF PP=Q(Q2) T
HEN 17120
17130 FORJ=0 TO I:IF PP=Q(J) THEN 17120
17140 NEXT J:Q(I)=PP
17150 NEXT I
17160 PRINT "";NA$(1);":";A$(1,Q(Q2
))
17170 PRINT "";NA$(0);":"
17180 FOR I=0 TO 4:PRINT I+1;")   ";MID$(
A$(0,Q(I)),2):NEXT
17190 PRINT "WHICH DO YOU THINK IS THE
 RIGHT ANSWER?"
17200 INPUT "TYPE IN THE NUMBER:";AA:Q
T=QT+1
17210 IF AA-1=Q2 THEN 17250
17220 PRINT "WRONG! THE CORRECT ANSWER
 WAS:"
17230 PRINT MID$(A$(0,Q(Q2)),2):GOTO 172
70
17240 POKE 53281,0
17250 PRINT "":FOR I=1 TO 11:PRI
NT "                    RIGHT!":NEXT
17260 FOR I=1 TO 15:POKE 53281,I:FOR J=1
 TO 200:NEXT J,I:RR=RR+1
17270 PRINT "RETURN FOR NEW QUESTION
```

```
 OR 'ZZZ' TO       QUIT FUNCTION:"
17280 Q$="":INPUT Q$:IF Q$="ZZZ" THEN RE
TURN
17290 PRINT "J";:GOTO 17080
```

This is the only really original module in the program. Its purpose is to generate a random question, to display five possible answers on the screen and to accept an input from the user specifying the right answer—or at least attempting to.

*Commentary*

Lines 17050–17070: This routine sets the difficulty of the test. Possible answers will either be drawn from the whole of the file, in which case many of them will probably be inappropriate and make the task of selecting the right answer much easier. Alternatively, answers can be drawn only from the same type, making the task more difficult, since all the answers will look at least possible.

Line 17080: A random question is drawn from the file and a random position for its position number chosen in the array Q.

Lines 17090–17100: If QU is zero, or if there are not five answers in the type group, then the range of possible alternative answers is the whole file P1-P2. If QU is 1 and there are at least five answers in the same group as the right answer, P1 and P2 are reset to the beginning and end of that group.

Lines 17110–17150: Four random answers are chosen from the range set by P1 and P2. A check is made that the randomly selected answer is not the same as the correct answer or a previously selected random answer.

Lines 17160–17200: The question is printed, together with the five possible answers and the user is asked to specify which is correct. The variable QT, recording the total number of questions, is incremented.

Lines 17240–17260: The user is rewarded by a multi-coloured series of screen background changes and the variable RR is incremented.

*Testing Module 5.1.8*

If you have some data ready to reload, you should now be able to generate your own multiple choice tests. You will only be able to generate the harder form of test if you have entered sufficient data for the program to regularly light upon a group of five answers of the same type.

MODULE 5.1.9

```
18000 REM#*******************************
18010 REM SCORE
18020 REM********************************
18030 PRINT "█████████████████SCORE":I
F QT=0 THEN RETURN
18040 PRINT "███TOTAL QUESTIONS:";QT
18050 PRINT "█CORRECT ANSWERS:";RR
18060 PRINT "██SCORE:";INT(((RR-QT/5)/(Q
T*.8))*100);"%"
18070 INPUT "██DO YOU WISH TO ZERO SCORE
 (Y/N):";Q$
18080 IF Q$="Y" THEN QT=0:RR=0
18090 RETURN
```

This module keeps the score, adjusted for the fact that one in five answers would be correct if the answers were only chosen randomly.

### *Summary*

This is quite a powerful program, but remember that you will only confirm that for yourself by entering enough data to make it enjoyable. The program is also a reminder that whenever possible, if you are going to write a complex program, you may as well go a little further and write a multi-purpose one, saving yourself a great deal of work in the future.

### *Going further*

1) As presently constituted, the program checks that the same answer is not displayed twice for a question. What it does not do is to check that two different answers might actually be the same. Could you insert a check to deal with this.
2) The question of rewards for success is an interesting one—adults seem to find success its own reward when playing with, I mean using, this program. For children, however, all manner of rewards are possible. What about tagging a short game onto the program which can only be accessed for three minutes or so when a number of answers are answered correctly.

## 5.2 WORDS

Once you have a program that works well, you soon find that it suggests other uses to you. Such was the case with MultiQ, and the result was this program, which can be used as an enjoyable word-learning aid for children in the earliest stages of reading. The only real difference between this program and MultiQ is that the questions take the form of pictures and the answers are possible words to go with the pictures.

As for the pictures themselves, they are no more than the output from another program in the book, Artist, picked up from tape and loaded into this program's dictionary. The capacity of the program, as presented here is 50 pictures, though another set could be picked up from tape if so desired. Designs meant to be used by this program must use only the bottom 10 lines of the screen, since the top part of the screen is needed for the questions and prompts.

**Words: Table of Variables**

| | |
|---|---|
| A% | Stores data for design characters and colours. |
| B% | Co-ordinates of corners of designs. |
| FNA(SS) | Value of element in A% whose position is dictated by the location of the corners of the design. |
| FNB(SS) | Actual character code derived from FNA. |
| WW$ | Answer input in response to question module. |

MODULE 5.2.1

```
11000 REM#********************************
11010 REM MENU
11020 REM*********************************
11030 POKE 53281,15:PRINT "▒▒▒▐▐▐▐▐▐▐▐▐▐
▐▐▒WORDS"
11040 PRINT "▒COMMANDS AVAILABLE:"
11050 PRINT "▒ ▒1)INPUT NEW ITEMS"
11060 PRINT " 2)SEARCH/DELETE"
11070 PRINT " 3)GENERATE QUESTIONS"
11080 PRINT " 4)DISPLAY OR RESET SCORE"
11090 PRINT " 5)DATA FILES"
11100 PRINT " 6)INITIALISE"
11110 PRINT " 7)STOP"
11120 INPUT "▒▒WHICH DO YOU REQUIRE:";Z:
PRINT "▒";
11130 ON Z GOSUB 13000,15000,16000,17000
,18000,12000,11140:GOTO 11000
11140 PRINT "▒▒▒▒▒▒▒▒▒▒▒▒▐▐▐▐▐▐▐▐▐▐▐▐▐
▐▐▒GOODBYE":END
```

A standard menu module.

MODULE 5.2.2

```
12000 REM#*******************************
12010 REM INITIALISE
12020 REM********************************
12030 CLR:DIM Q(4),A$(49),A%(49,255),B%(
49,4):IT=0:R$=CHR$(13)
12040 GOTO 11000
12060 GOTO 11000
```

Initialisation of the main variables.

MODULE 5.2.3

```
13000 REM#*******************************
13010 REM INPUT OF NEW ITEMS
13020 REM********************************
13030 PRINT"▯▚▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▚NEW ITEMS"
13040 IF IT>=100 THEN PRINT "▚▚NO MORE R
OOM:FOR I=1 TO 2000:NEXT:RETURN
13050 INPUT "▚▚POSITION TAPE CORRECTLY,
THEN ▚RETURN▚:";Q$
13060 PRINT "▯":OPEN 1,1,0,"ARTIST":FOR
I=0 TO 4:INPUT#1,B%(IT,I):NEXT
13070 FOR I=0 TO B%(IT,4)-1:INPUT#1,C1,C
2
13080 A%(IT,I)=256*C1+C2-32767:NEXT I:CL
OSE1:SS=IT
13090 PRINT "▯";:GOSUB 14000
13100 INPUT "▚▚DO YOU WANT THIS (Y/N):";
Q$:IF Q$="N" THEN RETURN
13110 INPUT "WORD TO GO WITH PICTURE:";W
$
13120 INPUT "IS THIS CORRECT (Y/N):";Q$:
IF Q$="N" THEN 13090
13130 A$(IT)=W$:IT=IT+1
13140 INPUT "ANOTHER PICTURE (Y/N):";Q$:
IF Q$="Y" THEN 13000
13150 RETURN
```

This module picks up from tape designs created by Artist and allows the user to tag the right word onto them.

MODULE 5.2.4

```
14000 REM#******************************
14010 REM PRINT DESIGN
14020 REM*******************************
14030 PP=0:FOR I=B%(SS,1)+1 TO B%(SS,3)-
1
14040 FOR J=B%(SS,0)+1 TO B%(SS,2)-1
14050 PP=PP+1:T1=A%(SS,PP)+32767:POKE 10
24+40*I+J,INT(T1/256)
14060 POKE55296+40*I+J,T1-256*INT(T1/256
):NEXT J,I
14070 RETURN
```

This module makes use of the two defined functions to extract the correct characters and colours from the numeric values saved by Artist and to POKE these onto the screen and into the colour memory.

*Commentary*

Line 14030: B%(SS,1) and B%(SS,3) record the vertical co-ordinates of the top left and bottom right corners of the design.

Line 14040: B%(SS,0) and B%(SS,2) record the horizontal co-ordinates of the same corners.

*Testing Modules 5.2.1–5.2.4*

You should now be able to load designs created by Artist, see them reprinted on the screen and then loaded into the main array with a chosen word associated with them.

MODULE 5.2.5

```
18000 REM#******************************
18010 REM DATA FILES
18020 REM*******************************
18030 PRINT "■■POSITION TAPE CORRECTLY T
HEN ■RETURN■--"
18040 INPUT "MOTOR WILL STOP AUTOMATICAL
LY:";O$:POKE 192,7:POKE 1,39
18050 PRINT "■■COMMANDS AVAILABLE:":PRIN
T "■ 1)SAVE DATA",,," 2)LOAD DATA"
18060 INPUT "■WHICH DO YOU REQUIRE:";Q:O
N Q GOTO 18070,18130:RETURN
18070 POKE 1,7:FOR I=1 TO 2000:NEXT
```

```
18080 OPEN1,1,1,"WORDS":PRINT#1,IT
18090 FOR I=0 TO IT-1:PRINT#1,A$(I);R$;B
%(I,4)
18100 FOR J=0 TO B%(I,4)-1:PRINT#1,A%(I,
J):NEXT J
18110 FOR J=0 TO 3:PRINT#1,B%(I,J):NEXT
J,I
18120 CLOSE1:RETURN
18130 OPEN1,1,0,"WORDS":INPUT#1,IT
18140 FOR I=0 TO IT-1:INPUT#1,A$(I),B%(I
,4)
18150 FOR J=0 TO B%(I,4)-1:INPUT#1,A%(I,
J):NEXT J
18160 FOR J=0 TO 3:INPUT#1,B%(I,J):NEXT
J,I
18170 CLOSE1:RETURN
```

A standard data-file module.

MODULE 5.2.6

```
15000 REM********************************
15010 REM USER SEARCH/DELETE
15020 REM********************************
15030 SS=0:IF IT=0 THEN RETURN
15040 PRINT "[illegible]SEARCH"
15050 IF SS>IT-1 THEN SS=IT-1
15060 IF SS<0 THEN SS=0
15070 PRINT "[illegible]
ITEMS=";IT
15080 PRINT "[illegible]COMMANDS AVAILABLE:"
15090 PRINT "[illegible] >[illegible]RETURN[illegible] FOR NEXT ITEM
"
15100 PRINT "[illegible] >[illegible]POS/NEG NUMBER TO MOVE
POINTER"
15110 PRINT "[illegible] >[illegible]'DDD' TO DELETE ITEM"
15120 PRINT "[illegible] >[illegible]'ZZZ' TO QUIT FUNCTION"
15130 PRINT "[illegible]";A$(SS):GOSUB 14000
15140 Q$="":INPUT "[illegible]WHICH DO Y
OU REQUIRE:";Q$
15150 IF Q$<>"DDD" THEN 15190
15160 FOR I=SS TO IT-1:FOR J=0 TO 255:A%
(I,J)=A%(I+1,J):NEXT J,I
15170 FOR I=SS TO IT-1:A$(I)=A$(I+1):FOR
```

```
J=0 TO 4:B%(I,J)=B%(I+1,J):NEXT J,I
15180 IT=IT-1:GOTO 15040
15190 IF Q$="ZZZ" THEN RETURN
15200 IF Q$="" THEN SS=SS+1:GOTO 15040
15210 SS=SS+VAL(Q$):GOTO 15040
15220 GOTO 15220
```

A simple user search module.

MODULE 5.2.7

```
16000 REM#*******************************
16010 REM RANDOM QUESTIONS
16020 REM********************************
16030 Q1=INT(RND(0)*IT):Q2=INT(RND(0)*5)
:Q(Q2)=Q1
16040 FOR I=0 TO 4:IF I=Q2 THEN 16090
16050 PP=INT(RND(0)*IT)
16060 IF PP=Q1 THEN 16050
16070 FOR J=0 TO I:IF PP=Q(J) THEN16050
16080 NEXT J:Q(I)=PP
16090 NEXT I
16100 SS=Q1:PRINT "▢":GOSUB 14000
16110 PRINT "▓▓";:FOR I=0 TO 4:PRINT "▓"
;A$(Q(I)):NEXT
16120 PRINT "▓▓TYPE IN THE RIGHT WORD FO
R THE PICTURE:":INPUT WW$:QT=QT+1
16130 IF WW$<>A$(Q1) THEN PRINT "WRONG!
THE ANSWER WAS ▓";A$(Q1):GOTO 16160
16140 PRINT "▓▓RIGHT▓RIGHT▓RIGHT▓RIGHT▓R
IGHT▓RIGHT▓RIGHT▓RIGHT":RR=RR+1
16150 FOR I=0 TO 15:POKE 53281,I:FOR J=1
 TO 150:NEXT J,I
16160 INPUT "▓MORE (Y/N):";Q$:IF Q$="Y"
THEN 16000
16170 RETURN
```

This module is equivalent to the random question generator of MultiQ but simpler in that it always chooses possible answers from the whole of the file of items present.

MODULE 5.2.8

```
17000 REM#*******************************
17010 REM SCORE
```

```
17020 REM*******************************
17030 IF QT=0 THEN RETURN
17040 PRINT "[illegible]SCORE"
17050 PRINT "[illegible]TOTAL QUESTIONS:";QT
17060 PRINT "[illegible]CORRECT ANSWERS:";RR
17070 PRINT "[illegible]SCORE:";INT(((RR-QT/5)/(Q
T*.8))*100);"%"
17080 INPUT "[illegible]DO YOU WISH TO ZERO SCORE
 (Y/N):";Q$
17090 IF Q$="Y" THEN QT=0:RR=0
17100 RETURN
```

The same function as the score module in MultiQ.

***Summary***

This again is a program which requires some work if it is to be of any use, since the small designs it uses take some time to build up in quantity. One easy answer would be to get together with some other 64 owners and swap tapes of designs. Micro-computing doesn't have to isolate you from the rest of the world!

***Going further***

1) The question of rewards rears its ugly head even more pronouncedly with this program—try to think of ways in which a correct answer can be more excitingly rewarded.

## 5.3 TYPIST

Not all education is about manipulating complex data. Much of the most important learning we do involves the training of responses and computers excel at that, which is why pilots, navigators and the like now begin their careers in front of computer simulators rather than the expensive and risky real thing. This program is not quite as grandiose as that, but it is nevertheless an extremely effective learning tool.

Of all the programs that I have written, this one must come close to being my favourite. Its presence here proves that a program doesn't have to be long to be useful. This one is short, neat and good at what it sets out to do, which was to help me improve my touch typing. Of all the versions that have been written, the 64 version is by far the best, so I hope it finds a place in your collection.

**Typist: Table of Variables**

| | |
|---|---|
| C$ | Line of spaces used to clear lines of text. |
| CH | Number of characters in tests so far. |

| | |
|---|---|
| RIGHT | Number of correct characters. |
| SUM | Number of characters entered. |
| TI | System variable counting time elapsed in 60ths of a second. |
| TT$ | Temporary storage for time taken. |

MODULE 5.3.1

```
11000 REM********************************
11010 REM PRINT KEYBOARD
11020 REM********************************
11030 POKE 53281,6
11040 PRINT "▒▒█     ████████████████████████████████
██████████████████████"
11050 A$="←1234567890+-£"
11060 PRINT "    ";:FOR I=1 TO 14:PRINT
"▒█ ▒█";MID$(A$,I,1);:NEXT
11070 PRINT "▒█ ▒C█ ▒D█ ▒█"
11080 PRINT "    █▒
           "
11090 A$="QWERTYUIOP@*↑"
11100 PRINT "      ▒ ▒CC█";:FOR I=1 TO LEN
(A$):PRINT "█▒ █▒";MID$(A$,I,1);:NEXT
11110 PRINT "▒█ ▒RR█ ▒█"
11120 PRINT "    █▒
           "
11130 A$="ASDFGHJKL:;="
11140 PRINT "    ▒ ▒R█   ▒S█";
11150 FOR I=1 TO LEN(A$):PRINT "█▒ █▒";M
ID$(A$,I,1);:NEXT
11160 PRINT "▒█ ▒RR█ ▒█"
11170 A$="ZXCVBNM,./"
11180 PRINT "    █▒
           "
11190 PRINT "    ▒█ ▒R█ ▒SH█ █";
11200 FOR I=1 TO LEN(A$):PRINT "█▒ █▒";M
ID$(A$,I,1);:NEXT
11210 PRINT "▒█ ▒SH█ ▒↑█ ▒←█ "
11220 PRINT "    █▒
           "
11230 PRINT "    █▒          ▒
  █          "
```

The purpose of this module is simply to print out a fairly crude copy of the 64's keyboard on the screen, thus allowing the user to look at the screen rather than at the keyboard when an input is being made.

*Commentary*

Lines 11050–11070: This section, like those that follow,prints out the irregular ends of the keyboard (things like the RETURN and RESTORE keys) which spoil the regular pattern, then prints out a line of black inverse spaces and superimposes upon that line the names of the alphanumeric keys in the appropriate places. A black inverse line is then placed underneath the row of keys.

*Testing Module 5.3.1*

The module should print a copy of the keyboard in the top half of the screen.

MODULE 5.3.2

```
12000 REM********************************
12010 REM ACCEPT INPUT
12020 REM********************************
12030 SUM=0:CH=0:RIGHT=0:RESTORE:TT$="00
0000"
12040 C$="
          "

12050 READ A$:IF A$="STOP" THEN GOTO 120
30
12060 PRINT "▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒"
12070 FOR LINE=1 TO 3:PRINT C$;:NEXT:PRI
NT "▒▒▒▒"
12080 IF LEN(A$)>39 THEN PRINT "STRING T
OO LONG":STOP
12090 PRINT "■";A$:PRINT "▒▒▒";:FOR I=1
TO LEN(A$)
12100 GET T$:IF T$="" THEN 12100
12110 IF T$="▒" OR T$="▒" OR T$="▒" OR T
$="▒" OR T$=CHR$(13) THEN GOTO 12100
12120 IF I=1 THEN TI$=TT$
12130 SUM=SUM+1:PRINT T$;" ▒";
12140 IF T$<>MID$(A$,I,1) THEN PRINT "←▒
▒";:GOTO 12100
12150 RIGHT=RIGHT+1:NEXT I:PRINT "▒":CH=
CH+LEN(A$):TT$=TI$
12160 PRINT "▒▒▒▒";STR$(INT(RIGHT/SUM*10
00)/10);"%     "
12170 PRINT STR$(INT(SUM/(TI/6000))/100)
;" CPS    "
12180 INPUT "MORE (Y/N):";Q$
12190 POKE 780,0:POKE 781,21:POKE 782,0:
```

```
SYS 65520:PRINT C$
12200 IF Q$<>"N" THEN 12050
12210 END
```

This module prints out a line of text to be copied, then accepts key by key input, keeping track of time, success rate and indicating errors.

*Commentary*

Line 12050: Text to be copied is stored in DATA statements at the end of the program. These DATA statements should be terminated with a line reading simply STOP, as in the example lines given, which causes the program to begin READing again.

Lines 12060–12090: These lines use the string of spaces (C$) to clear the area where text is to be printed, print the text and move the print position down to accept input on the line beneath.

Line 12110: Cursor move arrows are not accepted as an input.

Line 12120: The program keeps track of the time taken to input the text, but timing commences only after the first letter of each line is input and is suspended between lines. The total time taken so far is stored in TT$, to which TI$ (see last program) is set at the beginning of each line.

Lines 12130–12150: The last letter input is printed. If it is wrong it is indicated by an error and the print position is returned to that point. Total number of keys pressed and the number correct are recorded.

Lines 12160–12180: On finishing the line the percentage success rate is displayed. The system variable TI, which stores the same value as TI$, but expressed in 60ths of a second, is used to calculate the number of characters per second input. The seemingly convoluted formula ensures that two decimal places are normally printed.

Line 12190: This line demonstrates an alternative method of dictating the position at which the next character is to be printed. To use this method, zero must be POKEd into location 780, the row position into 781 and the column into 782. Calling the ROM routine at 65520 then moves the print position to that point. This method can be used to replace strings using cursor control characters. Here it is simply used to dictate that the MORE. prompt is overprinted with a line of spaces if the program is continuing.

*Testing Module 5.3.2*

This module cannot be tested until some DATA is entered for the program to READ. Enter another module at 13000, consisting of the text you wish to practise on, terminate it with a DATA line reading STOP and then run the program. You should be faced with the first line of text stored as DATA and be tested as described in the commentary.

EXAMPLE OF PRACTICE TEXT

```
13000 REM#*****************************
13010 REM DATA FOR TESTS
13020 REM******************************
13030 DATA "ASDF :LKJ ASDF :LKJ ASDF ;LK
J AS"
13040 DATA "ASDF :LKJ ASDF :LKJ ASDF ;LK
J AS"
13050 DATA "A AD ADD ADDS; ASK LAD ALL F
ALLS"
13060 DATA "A AD ADD ADDS; ASK LAD ALL F
ALLS"
13070 DATA "A AD ADD ADDS; ASK LAD ALL F
ALLS"
13080 DATA STOP
```

***Summary***

This is a program which can only really work for you if you use it seriously. One way of making the best use of it is to get hold of a book of typing exercises and use that as the basis of the data you enter. Given the effort you will find it an effective tool in improving your touch typing.

***Going further***

Correct technique in typing depends upon using the right finger for the key. Drawing upon a typing tutor it should be a simple matter to colour the keys on the key board to give an indication of which finger should be used. At the very least it would be good practice in using the colour characteristics within a string.

# CHAPTER 6
# High Micro-Finance

Despite the jokes about £1 million gas bills, one thing that computers do superbly well is to handle financial information. It is not only the fact that they are able to store and process the data so much more quickly than a human being, it is as much to do with their ability to present the facts in clear, understandable ways. In this chapter you will find three home finance programs that use both the 64's calculating abilities and its flexible screen handling to take some of the mystery out of money.

## 6.1 BANKER

Our first program is Banker, a simple tool which is designed to allow you to keep up with the state of your bank account before the dreaded envelope from the bank drops onto the doormat. The program deals with payments and receipts, regular payments and one-off items, producing a neatly formatted statement for any month you care to specify. In the course of the program you will begin to tackle some of the problems of setting out numeric data in a comprehensible form on the screen.

**Banker: Table of Variables**

| | |
|---|---|
| A(99,1) | Storage of payment amounts and day of payment. |
| A$(99,1) | Storage of payment names and months in which payment is to be made. |
| CD | Flag indicating whether payment is credit or debit. |
| CR$ | Separator for data files. |
| IN | Initialisation flag. |
| M | Month number minus 1. |
| MM | Temporary variable used in formatting payment amounts. |
| MM$ | Used to contain the formatted payment amount. |
| MO$ | Storage of names of months. |
| PA | Number of payments stored. |
| R$ | Temporary string storing months in which payment is to be made. |
| S | Temporary storage for day of payment in month specified. |

SUM Used to cumulate amounts for running total in statement.

MODULE 6.1.1

```
11000 REM#*******************************
11010 REM MENU
11020 REM********************************
11030 POKE 53281,7:PRINT "▯▮▮▮▮▮▮▮▮▮▮▮▮▮
▮▮▮▮▮▮BANKER▮▮"
11040 PRINT "▮▮COMMANDS AVAILABLE:"
11050 PRINT "▮   1)NEW PAYMENTS"
11060 PRINT "▮   2)EXAMINE/DELETE PAYMEN
TS"
11070 PRINT "▮   3)PRINT STATEMENT"
11080 PRINT "▮   4)DATA FILES"
11090 PRINT "▮   5)INITIALISE"
11100 PRINT "▮   6)STOP"
11110 INPUT "▮▮▮WHICH DO YOU REQUIRE:";Z
:PRINT "▯";
11120 IF Z=5 OR Z=6 OR IN=1 THEN 11140
11130 PRINT"▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮NOT INITIALI
SED.":FOR I=1 TO 2000:NEXT:GOTO 11000
11140 IF PA<>0 OR (Z<>2 AND Z<>3) THEN 1
1160
11150 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮SORRY, NO DATA
 YET.":FOR I=1 TO 2000:NEXT:GOTO 11000
11160 ON Z GOSUB 13000,14000,15000,16000
,12000,11180
11170 GOTO 11000
11180 PRINT "▯▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
▮▮BANKER"
11190 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮CLOSED FOR BUS
INESS"
11200 END
```

A standard menu module.

MODULE 6.1.2

```
12000 REM#*******************************
12010 REM VARIABLES
12020 REM********************************
12040 CLR:IN=1:DIM A$(99,1),A(99,1):A(0,
1)=999
12050 RESTORE
```

```
12060 S$="          "
12070 DIM MO$(11):FOR I=0 TO 11:READ A$:
MO$(I)=A$:NEXT:CR$=CHR$(13)
12080 DATA JANUARY,FEBRUARY,MARCH,APRIL,
MAY,JUNE,JULY,AUGUST,SEPTEMBER,OCTOBER
12090 DATA NOVEMBER,DECEMBER.
12100 GOTO 11000
```

Initialises variables and then places month names into MO$.

MODULE 6.1.3

```
13000 REM*******************************
13010 REM ENTER NEW ITEMS
13020 REM*******************************
13030 PRINT "NEW ITEMS
"
13040 PRINT "  1)CREDIT":PRINT "  2)DEBI
T"
13050 INPUT "WHICH DO YOU REQUIRE:";CD
:CD=CD-1
13060 INPUT "NAME OF PAYMENT:";Q$
13070 INPUT "AMOUNT:";Q
13080 INPUT "MONTHS (E.G. 01040710):";R
$:PRINT "";
13090 FOR I=1 TO LEN(R$) STEP 2:LET M=VA
L(MID$(R$,I,2))-1
13100 IF M>=0 AND M<=11 THEN GOTO 13130
13110 PRINT:PRINT "INVALID MONTH INPUT
                    "
13120 FOR J=1 TO 2000:NEXT:GOTO 13080
13130 PRINT MO$(M);"/";:NEXT:PRINT
13140 INPUT "DAY OF PAYMENT:";S
13150 INPUT "ARE THESE CORRECT (Y/N):
";T$:IF T$="N" THEN PRINT"";:GOTO 13000
13160 PA=PA+1:FOR J=PA-1 TO 0 STEP -1
13170 IF S<A(J,1) THEN FOR K=0 TO 1:A$(J
+1,K)=A$(J,K):A(J+1,K)=A(J,K):NEXT K,J
13180 J=J+1:A$(J,1)="000000000000"
13190 FOR I=1 TO LEN(R$) STEP 2:M=VAL(MI
D$(R$,I,2))
13200 A$(J,1)=LEFT$(A$(J,1),M-1)+"1"+RIG
HT$(A$(J,1),12-M):NEXT
13210 A$(J,0)=Q$:A(J,0)=Q:A(J,1)=S
13220 IF CD=1 THEN A(J,0)=A(J,0)*-1
13230 RETURN
```

The purpose of this module is to allow the input of payment names and the associated data.

*Commentary*

Lines 13080–13130: The months in which a payment is made can vary between one for a one-off payment and 12 for a regular standing order. Months are input in the form of a string of two digit numbers which are read and checked, then the month names are printed on the screen as a check. This simple method of entry allows a high degree of flexibility without complex programming.

Line 13170: Payments are stored in a single array according to their date of payment. Insertion is accomplished by simply scanning the file from the highest day value.

Lines 13180–13200: These lines set up a month indicator consisting of 12 zeros. The months specified are then scanned and the corresponding positions in the indicator are set to 1.

Lines 13220: Payments and receipts are both input as positive sums. If a debit is specified by the variable CD, the amount is multiplied by minus one.

MODULE 6.1.4

```
14000 REM#******************************
14010 REM EXAMINE/DELETE ITEMS
14020 REM*******************************
14030 FOR I=0 TO PA-1:PRINT "□";
14040 PRINT "■PAYMENT:";A$(I,0)
14050 PRINT "■AMOUNT:";A(I,0)
14060 PRINT "■MONTHS:";:FOR J=1 TO 12
14070 IF MID$(A$(I,1),J,1)="1" THEN PRIN
T MO$(J-1);"/";
14080 NEXT J:PRINT
14090 PRINT "■DAY OF PAYMENT:";A(I,1)
14100 PRINT "■■■■COMMANDS ON FUNCTION K
EYS:■■":PRINT "■1 - NEXT ITEM":
14110 PRINT "■2 - QUIT":PRINT "■3 - DELE
TE ITEM"
14120 PRINT "■■WHICH DO YOU REQUIRE:?■"
14130 GET Q$:IF Q$="" THEN 14130
14140 IF ASC(Q$)<>140 THEN 14170
14150 FOR J=I TO PA-1:FOR K=0 TO 1:A$(J,
K)=A$(J+1,K):A(J,K)=A(J+1,K):NEXT K,J
14160 PA=PA-1:RETURN
14170 IF ASC(Q$)=133 THEN NEXT I
14180 RETURN
```

A simple user-search module which prints out payments and the months in which they are to be made and allows deletion using three of the function keys to input commands.

*Testing Modules 6.1.1–6.1.4*

You should now be able to input payments and the months in which they are to be made and to scan through them, deleting as you wish.

MODULE 6.1.5

```
16000 REM******************************
16010 REM DATA FILES
16020 REM******************************
16030 PRINT "POSITION TAPE, THEN ▓RETURN
■ ":INPUT"(AUTOMATIC MOTOR STOP):";Q$
16040 POKE 192,7:POKE 1,39
16050 PRINT "▓PLACE RECORDER IN CORRECT
MODE,":INPUT "THEN PRESS ▓RETURN■:";Q$
16060 POKE 192,7:POKE 1,39
16070 PRINT "▓▓FUNCTIONS AVAILABLE:":PRI
NT "▓▓1)SAVE DATA":PRINT "▓2)LOAD DATA"
16080 INPUT "▓▓WHICH DO YOU REQUIRE:";Q:
ON Q GOTO 16100,16150
16090 RETURN
16100 POKE 192,0:FOR I=1 TO 5000:NEXT
16110 OPEN 1,1,2,"BANKER"
16120 PRINT#1,PA
16130 FOR I=0 TO PA-1:PRINT#1,A$(I,0),CR
$,A$(I,1),CR$,A(I,0),CR$,A(I,1):NEXT
16140 CLOSE1:RETURN
16150 OPEN 1,1,0,"BANKER"
16160 INPUT#1,PA
16170 FOR I=0 TO PA-1:INPUT#1,A$(I,0),A$
(I,1),A(I,0),A(I,1):NEXT
16180 CLOSE1
16190 GOTO 11000
```

A standard data-file module.

MODULE 6.1.6

```
17000 REM#*******************************
17010 REM FORMAT MONEY
17020 REM********************************
17030 MM=MM+10000.001
17040 M$=MID$(STR$(MM),3,LEN(STR$(MM))-3
)
17050 FOR FF=1 TO 7:IF MID$(M$,FF,1)<>"0
" THEN RETURN
17060 M$=LEFT$(M$,FF-1)+" "+RIGHT$(M$,7-
FF):NEXT FF
```

This short module is a simple method of achieving a standardised format for the amounts of money that are to be printed by the next module.

*Commentary*

Line 17030: This module is called up from a variety of places in the next module of the program and works on values drawn from different variables. In order to achieve this, the value to be formatted must first be stored in the variable MM. The first step of the formatting process is, assuming that the amount will not exceed £9999.99p, that 10000.001 is added to the amount. This gives a standardised length of nine characters (including the decimal point), of which the first and last characters are redundant.

Lines 17040: The value is now converted to a string and stripped of its first and last character (the 1s). Note that when converting a number to a string using the STR$ function, a space is automatically added to the front so that the second character of the number will actually be in position 3 in the string.

Lines 17050–17060: In this particular program we do not require leading zeros, so this routine scans the string converting any leading zeros to spaces. The result is a string which is invariably seven characters long, including two decimal places. Such strings can be printed in columns in the confidence that the positions of their decimal points will always coincide.

MODULE 6.1.7

```
15000 REM#*******************************
15005 REM COMPILE STATEMENT
15010 REM********************************
```

```
15020 PRINT "■■■■■■■■■■■■■■■■STATEMENT■":
SUM=0
15030 INPUT "■NUMBER OF MONTH FOR STATEM
ENT:";Q
15040 IF Q=1 THEN 15080
15050 FOR Q1=1 TO Q-1:FOR I=0 TO PA-1:IF
 MID$(A$(I,1),Q1,1)<>"1" THEN 15070
15060 SUM=SUM+A(I,0)
15070 NEXT I,Q1
15080 PRINT "■■■■■■■■■■■■■■";MO$(Q-1)
15090 PRINT "■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■ITEM■■■TOTAL■"
15100 PRINT "■BALANCE C/F■■■■■■■■■■■■■■■■■
■■■■■■";
15110 IF SUM<0 THEN PRINT "■";
15120 LET MM=ABS(SUM):GOSUB 17000:PRINT
M$
15130 FOR I=0 TO PA-1
15140 IF MID$(A$(I,1),Q,1)<>"1" THEN 152
20
15150 LET MM=A(I,1):GOSUB 17000:PRINT "■
";MID$(M$,2,3);:PRINT "■■■■";
15160 IF A(I,0)<0 THEN PRINT "■";
15170 PRINT A$(I,0)
15180 PRINT "■■■■■■■■■■■■■■■■■■■■■■■■■■■";:
MM=ABS(A(I,0)):GOSUB 17000:PRINT M$;
15190 SUM=SUM+A(I,0):PRINT "■■■";:IF SUM
<0 THEN PRINT "■";
15200 MM=ABS(SUM):GOSUB 17000:PRINT M$
15210 GET A$:IF A$="" THEN 15210
15220 NEXT I:INPUT "■RETURN■ TO CONTINUE
:";X
15230 RETURN
```

This module produces neatly formatted monthly statements—the whole point of the program.

*Commentary*

Lines 15030–15070: The month indicators of all the payments stored are scanned to see if any payments have been made under the headings for months prior to the statement. Such payments are cumulated to provide a balance of the account at the beginning of the specified month.

Lines 15100–15120: The balance is printed, making use of the previous module to format it. Note how easily a negative balance can be indicated by printing the red control character.

Lines 15130–15220: The file of payments is now scanned for those which apply to the current month. Each time a relevant payment is found, the day is printed on the left hand side of the screen, using the format module to standardise the printing, but cutting off the added decimal points. Then the payment name is printed, then the amount, with the red control character added if a debit is referred to. Finally, the payment is added to the variable SUM and the current balance is printed next to the amount of the payment, again in red if the balance is negative. The 64's flexible cursor control makes the construction of such tables a matter of ease-if it doesn't look right, simply add one more or less cursor moves until it does.

Line 15210: Payments are printed one at a time, the next payment awaiting the pressing of any key. This prevents payments scrolling off the top of the screen before they can be examined.

*Testing Modules 6.1.6–6.1.7*

If you have saved some data you should now be able to load it back into the 64 and call up a monthly statement, with the amounts and payment titles neatly formatted in columns. If the formatting of the monthly statement is correct then the program is ready for use.

***Summary***

This straightforward program raises some interesting questions about the degree of sophistication required to make a program useful. Inputting the months in the form of a string is, in many ways, rather crude compared to specifying whether the payment is to be made monthly, quarterly or annually and leaving it to the program to insert the payment in the relevant months. Such an added facility would be easily possible but it would increase the program length and reduce the flexibility inherent in specifying months in a straightforward way, which allows even irregular months to be entered. When designing your own programs you will need to be constantly aware of this tension between what it is worth doing automatically and what it is worth leaving to the user—the answer may well vary from user to user but complexity for the sake of it can be costly in terms of memory and can actually reduce the usefulness of a program.

***Going further***

1) The deletion module is extremely crude in that it only allows the user to page through the entries one by one. Why not add a facility to specify a positive or negative jump, using one of the previous programs as an example.
2) Another improvement would be to add a binary search module to replace the present scan from the end of the file when inserting items.
3) The month indicators use a whole 12 bytes for each payment. Using what you have learned about AND and OR, you should be able to store and retrieve the same information from 2 bytes (ie one element in an integer array).

## 6.2 ACCOUNTANT

This program won't actually cook the books for you but it will make them very much easier to keep and will present them in an orderly format whenever you wish, with provision for single items, main headings and sub-headings in the printing of the actual accounts.

**Accountant: Table of variables**

| | |
|---|---|
| A$(1,99) | Main file of names of payments. |
| A(1,99) | Main file of amounts of payments. |
| C$ | Line of spaces used in clearing text. |
| C(1) | Array storing number of items on credit/debit side of accounts. |
| CD | Indicator of whether item is a credit or debit. |
| CR$ | Data file separator. |
| GR | Used to record the number of items under a single main heading. |
| HH$ | Temporary storage of main heading name in user search module. |
| IN | Initialisation indicator. |
| M$ | String in which formatted money amount is stored. |
| MM | Temporary variable used in formatting money. |
| PL | Place in file for insertion of new sub-heading. |
| SS | Temporary variable used to cumulate items under a main heading. |
| TT | Used to cumulate items in accounts. |

MODULE 6.2.1

```
11000 REM#*****************************
11010 REM MENU
11020 REM******************************
11030 POKE 53281,7:PRINT "[illegible]
```

```
ACCOUNTANT"
11040 PRINT "COMMANDS AVAILABLE:"
11050 PRINT "   1)INPUT NEW HEADINGS"
11060 PRINT "   2)CHANGE/DELETE ITEMS"
11070 PRINT "   3)PRINT ACCOUNTS"
11080 PRINT "   4)DATA FILES"
11090 PRINT "   5)INITIALISE ACCOUNTS"
11100 PRINT "   6)STOP
11110 INPUT "WHICH DO YOU REQUIRE:";Z
:PRINT "";
11120 IF Z<1 OR Z>4 OR IN=1 THEN 11140
11130 PRINT "NOT INITI
ALISED":FOR I=1 TO 2000:NEXT:GOTO 11000
11140 IF Z<4 AND Z>0 THEN GOSUB 13000:PR
INT "";
11150 ON Z GOSUB 14000,17000,19000,20000
,12000,11170:Z=0:GOTO 11000
11160 .
11170 PRINT "
ACCOUNTANT"
11180 PRINT "PROGRAM TERMIN
ATED":END
```

Standard menu module.

MODULE 6.2.2

```
12000 REM#*******************************
12010 REM INITIALISE
12020 REM********************************
12030 CLR:DIM A$(1,99),A(1,99):CR$=CHR$(
13)
12040 C$="
         "

12050 IN=1:GOTO 11000
```

Initialisation module.

MODULE 6.2.3

```
13000 REM#*******************************
13010 REM CREDIT OR DEBIT?
13020 REM********************************
13030 PRINT "1)CRED
```

```
IT■■■■■■■■■■■2)DEBIT"
13040 INPUT "■■■■■■■■■■■■■■WHICH DO YOU
REQUIRE:";CD:CD=CD-1:RETURN
```

Before the input of any item the user is asked to specify whether the item is a credit or debit.

MODULE 6.2.4

```
14000 REM#*******************************
14010 REM INPUT HEADINGS
14020 REM********************************
14030 PRINT "■■■■■■■■■■■■■■■■NEW ITEMS:
";:IF CD=0 THEN PRINT "CREDIT"
14040 IF CD=1 THEN PRINT "DEBIT"
14050 PRINT "■■■IS THE ITEM:":PRINT "■■
 1)A SINGLE ITEM"
14060 PRINT "■  2)A MAIN HEADING":PRINT
"■  3)A SUB-HEADING"
14070 PRINT "■  INPUT '0' IF YOU WISH TO
 QUIT"
14080 INPUT "■■PLEASE SPECIFY:";TYPE
14090 ON TYPE GOTO 15000,15000,16000
14100 RETURN
```

When inputting an item the user is asked to specify whether it is a main heading,a sub-heading or a single item. When the accounts are printed, main headings have no sums placed against them, sub-headings are placed underneath their respective main headings and a sub-total printed for the group, and single items stand alone with a sum against them.

MODULE 6.2.5

```
15000 REM#*******************************
15010 REM SINGLE ITEM OR MAIN HEADING
15020 REM********************************
15030 Q=0:INPUT "■■NAME OF ITEM:";Q$
15040 IF TYPE<>2 THEN INPUT "■AMOUNT FOR
 ITEM:";Q
15050 INPUT "■■IS THIS CORRECT (Y/N):";R
$:IF R$="N" THEN 14000
15060 Q$="%"+Q$:IF TYPE=2 THEN Q$="*"+MI
D$(Q$,2)
```

```
15070 A$(CD,C(CD))=Q$:A(CD,C(CD))=Q:C(CD
)=C(CD)+1:GOTO 14000
```

This module receives the input of main headings or single items.

*Commentary*

Line 15040: An amount is only requested if the item is not a main heading.

Line 15060: An indicator is tagged to the beginning of the item-% for a single item, * for a main heading. These will never be printed but will be used by the program to identify the different types.

Line 15070: Note how the variable CD is used to specify which side of the main arrays the name and amount will be stored.

MODULE 6.2.6

```
16000 REM################################
16010 REM SUB-HEADING
16020 REM********************************
16030 INPUT "■■INPUT NAME OF MAIN HEADIN
G:";Q$:Q$="*"+Q$
16040 FOR I=0 TO C(CD)-1:IF A$(CD,I)=Q$
THEN 16060
16050 NEXT:PRINT "SORRY, NO HEADING OF T
HAT NAME!":FOR I=1 TO 2000:NEXT
16060 PL=I+1:INPUT "■■NAME OF SUB-HEADIN
G:";Q$
16070 INPUT "■AMOUNT FOR SUB-HEADING:";Q
16080 INPUT "■■ARE THESE CORRECT (Y/N):"
;R$:IF R$="N" THEN GOTO 14000
16090 Q$="$"+Q$
16100 FOR I=C(CD)+1 TO PL+1 STEP -1:A$(C
D,I)=A$(CD,I-1):A(CD,I)=A(CD,I-1):NEXT
16110 A$(CD,PL)=Q$:A(CD,PL)=Q:C(CD)=C(CD
)+1:GOTO 14000
```

This module accepts the input of sub-headings.

*Commentary*

Lines 16030–16050: The name of the relevant main heading is requested and checked against the headings in the file. If the main heading is not present then an error message is generated. Note the way a loop is used to conduct the search, with the program dropping out of the loop if the item is found and the value of the loop variable I being used to determine the point at which the sub-heading will be inserted. Completing the loop means that the main heading is not present.

Lines 16090–16110: The sub-heading is tagged with a $ symbol and added to the main file immediately following the relevant main heading.

*Testing Modules 6.2.1–6.2.6*

You should now be able to input credit or debit items to the account and have them properly inserted into the correct side of the main file (credit side = 0, debit side = 1). This can only be checked in direct mode.

MODULE 6.2.7

```
20000 REM#******************************
20010 REM DATA FILES
20020 REM*******************************
20030 PRINT "▮▮POSTION TAPE CORRECTLY, T
HEN ▮RETURN▮---"
20040 INPUT "MOTOR WILL STOP AUTOMATICAL
LY:";Q$:POKE 192,7:POKE 1,39
20050 PRINT "▮▮▮COMMANDS AVAILABLE:":PRI
NT "▮▮1)SAVE DATA":PRINT "▮2)LOAD DATA"
20060 INPUT "▮▮WHICH DO YOU REQUIRE:";Q:
ON Q GOTO 20080,20140
20070 RETURN
20080 FOR I=0 TO 1:IF A$(I,0)="" THEN A$
(I,0)=" ":NEXT
20090 POKE 1,7:FOR I=1 TO 2000:NEXT
20100 OPEN 1,1,2,"ACCOUNTS"
20110 FOR I=0 TO 1:PRINT#1,C(I):IF C(I)=
0 THEN 20130
20120 FOR J=0 TO C(I)-1:PRINT#1,A$(I,J),
CR$,A(I,J):NEXT J
20130 NEXTI:CLOSE1:RETURN
20140 OPEN 1,1,0,"ACCOUNTS"
20150 FOR I=0 TO 1:INPUT#1,C(I):IF C(I)=
0 THEN 20170
20160 FOR J=0 TO C(I)-1:INPUT#1,A$(I,J),
A(I,J):NEXT J
20170 NEXT I:CLOSE1:RETURN
```

A standard data-file module.

MODULE 6.2.8

```
21000 REM#******************************
21010 REM FORMAT MONEY
21020 REM*******************************
21030 MM=MM+10000.001:M$=MID$(STR$(MM),3
,7)
21040 FOR P=1 TO 3
21050 IF MID$(M$,P,1)="0" THEN M$=LEFT$(
M$,P-1)+" "+RIGHT$(M$,7-P):NEXT
21060 RETURN
```

This module performs the same function as the formatting module in the last program.

MODULE 6.2.9

```
17000 REM#******************************
17010 REM CHANGES AND DELETIONS
17020 REM*******************************
17030 FOR I=0 TO C(CD)-1
17040 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮CHANGE OR DE
LETE▮"
17050 IF LEFT$(A$(CD,I),1)<>"$" THEN PRI
NT "▮▮";MID$(A$(CD,I),2);
17060 IF LEFT$(A$(CD,I),1)="*" THEN LET
HH$=MID$(A$(CD,I),2):PRINT
17070 IF LEFT$(A$(CD,I),1)="$" THEN PRIN
T "▮▮";HH$:PRINT"▮";MID$(A$(CD,I),2);
17080 IF A(CD,I)=0 THEN 17100
17090 PRINT"▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮";:MM=A(CD,I
):GOSUB 21000:PRINT M$
17100 PRINT "▮▮▮▮▮▮▮▮▮▮COMMANDS AVAIALAB
LE ON FUNCTION KEYS:"
17110 PRINT "▮▮  F1 - NEXT ITEM"
17120 PRINT "▮  F3 - CHANGE AMOUNT"
17130 PRINT "▮  F5 - RETURN TO MENU"
17140 PRINT "▮  F8 - DELETE ITEM"
17150 PRINT "▮▮▮WHICH DO YOU REQUIRE:?"
17160 GET Q$:IF Q$="" THEN 17160
17170 IF Q$=CHR$(140) THEN GOSUB 18000:R
ETURN
17180 IF Q$=CHR$(135) THEN RETURN
17190 IF Q$<>CHR$(134) OR LEFT$(A$(CD,I)
,1)="*" THEN GOTO 17240
```

```
17200 INPUT "[illegible]AMOUNT TO BE ADDED:";Q
17210 INPUT "[illegible]IS THAT CORRECT (Y/N):";R
$
17220 IF R$<>"N" THEN A(CD,I)=A(CD,I)+Q:
GOTO 17040
17230 PRINT "TTTT";:FOR L=1 TO 3:PRINT C
$;:NEXT:PRINT "TTTT";:GOTO 17200
17240 NEXT I:RETURN
```

This module allows the user to page through the items on the specified side of the accounts and to change or delete items.

*Commentary*

Lines 17050–17090: These conditions deal with formatting the different types of item. If the item is not a sub-heading then the item name is printed, its name also being stored in HH$ if it is a main heading. If the item is a sub-heading then the previously stored main heading title is printed above it to indicate the group in which it falls. Finally, if the item is a single item or a sub-heading then the previous module is used to format the amount associated with it before printing.

Lines 17190–17230: If the f3 key is pressed for a single item or sub-heading, the user has the option to change the amount associated with an item by inputting a positive or negative number. Note the use of C$ to clear the prompts if an error is made.

MODULE 6.2.10

```
18000 REM#*****************************
18010 REM DELETIONS
18020 REM******************************
18030 PL=I:IF LEFT$(A$(CD,PL),1)<>"*" TH
EN GR=1:GOTO 18060
18040 GR=0
18050 GR=GR+1:IF LEFT$(A$(CD,PL+GR),1)="
$" THEN GOTO 18050
18060 FOR K=PL TO C(CD)-GR-1:A(CD,K)=A(C
D,K+GR):A$(CD,K)=A$(CD,K+GR):NEXT
18070 C(CD)=C(CD)-GR:I=I-GR+1:RETURN
```

This module accomplishes any deletions specified in the previous module.

*Commentary*

Line 18030: PL is set equal to the value of the loop variable in the previous module. In the case of sub-headings and single items, the variable GR, which indicates the number of items to be deleted, is set equal to one.

Line 18040: In the case of main headings, the variable GR is incremented to take account of the main heading itself and all its sub-headings, since these must be deleted along with the main heading.

Line 18060: GR is used to determine how many items will be overwritten in the file and by how much the value in the relevant side of the array C must be reduced.

*Testing Modules 6.2.8–6.2.10*

You should now be able to input data and to page through it, changing the associated amounts or deleting items at will.

MODULE 6.2.11

```
19000 REM#********************************
19010 REM PRINT ACCOUNTS
19020 REM*********************************
19030 LET PA$="CREDIT":IF CD=1 THEN PA$=
"DEBIT"
19040 TT=0:SS=0:PRINT "[illegible]";PA
$;"[illegible]"
19050 FOR I=0 TO C(CD)-1:TT=TT+A(CD,I)
19060 PRINT "[illegible]";:IF I/2=INT(I/2) THEN PR
INT "[illegible]";
19070 IF LEFT$(A$(CD,I),1)="*" THEN PRIN
T
19080 IF LEFT$(A$(CD,I),1)="$" THEN PRIN
T "[illegible]";
19090 PRINT MID$(A$(CD,I),2)
19100 IF LEFT$(A$(CD,I),1)="*" THEN 1915
0
19110 PRINT "[illegible]";
19120 IF LEFT$(A$(CD,I),1)="%" THEN PRIN
T"[illegible]";
19130 MM=A(CD,I):GOSUB 21000:PRINT M$
19140 IF LEFT$(A$(CD,I),1)="$" THEN SS=S
S+A(CD,I)
19150 IF SS=0 OR LEFT$(A$(CD,I+1),1)="$"
 THEN 19180
19160 PRINT "[illegible]----------[illegible]
[illegible]";
```

```
19170 MM=SS:GOSUB21000:PRINT M$:SS=0
19180 GET GG$:IF GG$="" THEN 19180
19190 NEXT I:PRINT "▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
▌▌▌▌▌▌▌▌▌├--------"
19200 PRINT"▒TOTAL■:▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
▌▌▌";
19210 MM=TT:GOSUB 21000:PRINT M$
19220 PRINT "▒▒PRESS ANY KEY TO QUIT"
19230 GET GG$:IF GG$="" THEN 19230
19240 RETURN
19250 STOP
```

This module is parallel to the print statement module in the last program.

*Commentary*

Line 19060: To ensure that it is clear which sum goes with which amount, items and their associated amounts are printed alternately in black and green.

Line 19070: A clear line is left before a main heading is printed.

Line 19080: Sub-headings are inset two spaces.

Lines 19100–19140: For sub-headings and single items, the item name and the associated amount are printed. Sub-heading amounts are printed in a separate column and the total of sub-items under a main heading is cumulated in SS.

Lines 19150–19170: At the end of a group of sub-headings the total for the group is printed.

Line 19180: Once again items are printed one at a time, with the next item awaiting the pressing of any key.

Lines 19200–19210: The total for the relevant side of the accounts is printed.

*Testing Module 6.2.11*

Having input some data, you should now be able to display either side of the accounts. Note that only one side at a time can be displayed.

*Summary*

By now you should be becoming familiar with the techniques involved in adding and deleting items without disturbing the overall order of the file. You will also have learned something of the sheer fiddliness of displaying even simple figures on the screen in well formatted form. It is worth reviewing some of the methods used here before continuing because in the next program we shall be dealing with and displaying data of much greater complexity than anything encountered so far.

*Going further*

1) One useful added facility would be the ability to print the balance between the two sides of the account when either side is displayed.
2) As in the previous program, if you are going to store large numbers of items you will want to change the present user search module, which can only page through the items one by one. Be careful in doing this however, since the module must be able to detect the main headings as it passes through the file, especially for the purposes of deleting. Simply jumping through the file without regard for this need could result in disaster.

## 6.3 BUDGET

We now turn our attention to the most complex and difficult program you will encounter in this book. Budget is a powerful and flexible financial aid which enables the user to plan finances over a 12 month period and to examine the consequences of 'what ...if' decisions about income and expenditure. Intelligently used, it can provide some surprising insights into a family's finances over the year to come, quite apart from illustrating some of the problems of working with large bodies of numeric data. The arrays used by the program store some 800 different numeric values.

**Budget: Table of variables**

| | |
|---|---|
| BA(1,11) | Cash balance for each month. |
| BD(1,11) | Balance of budgeted payments over actual payments. |
| C1(1,11) | Main income. |
| C2(1,11) | Supplementary income. |
| CU | |

| | |
|---|---|
| | Temporary variable used in calculating cumulative surplus/deficit. |
| FO$ | Cursor control string used in formatting table. |
| H | Indicator showing which side of arrays is to be addressed. |
| I1 | Variable used to ensure proper handling of 12 month periods which pass beyond end of calendar year. |
| M1 | Temporary variable for month to start table display. |
| M2 | Temporary variable recording change of current month. |
| MM | Current month number. |
| MO(1,29) | Average monthly payment for each payment heading. |
| MO$(11) | Month names. |
| MY | Temporary variable used in formatting money figure. |
| MY$ | String storing formatted money figure. |
| N(1) | Number of items on both sides of arrays. |
| PA(1,29,11) | Amounts associated with payment headings. |
| PA$(1,29) | Names of payment headings. |
| PP | Temporary variable used to indicate position of item to be changed in array. |
| PT(1,11) | Monthly totals of expenditure. |
| R$ | Data-file separator. |
| T(1) | Temporary variable used to calculate total amount set aside in average budget allowance. |
| TT | Temporary variable used to calculate total payments for items included in average budget calculation. |
| Y | Month number of end of year. |

MODULE 6.3.1

```
21000 REM#*****************************
21010 REM DATA FILES
21020 REM******************************
21030 PRINT "■POSITION TAPE CORRECTLY, T
HEN ■RETURN■←-"
21040 INPUT "MOTOR WILL STOP AUTOMATICAL
LY:";Q$:POKE 192,7:POKE 1,39
21050 PRINT "■■COMMANDS AVAILABLE:"
21060 PRINT "■ 1)SAVE DATA":PRINT " 2)LO
AD DATA"
21070 INPUT "■WHICH DO YOU REQUIRE:";Q:O
N Q GOTO 21080,21130:RETURN
21080 POKE 1,7:FOR I=1 TO 2000:NEXT
21090 OPEN 1,1,1,"BUDGET":PRINT#1,MM,R$,
Y:FOR H=0 TO 1:PRINT#1,N(H)
```

```
21100 FOR I=0 TO 11:PRINT#1,C1(H,I),R$,C
2(H,I):NEXT I
21110 FOR I=0 TO N(H)-1:IF PA$(H,I)="" T
HEN PA$(H,I)=" "
21120 PRINT#1,PA$(H,I):FOR J=0 TO 11:PRI
NT#1,PA(H,I,J):NEXT J,I,H:CLOSE1:RETURN
21130 OPEN 1,1,0,"BUDGET":INPUT#1,MM,Y:F
OR H=0 TO 1:INPUT#1,N(H)
21140 FOR I=0 TO 11:INPUT#1,C1(H,I),C2(H
,I):NEXT I
21150 FOR I=0 TO N(H)-1
21160 INPUT#1,PA$(H,I):FOR J=0 TO 11:INP
UT#1,PA(H,I,J):NEXT J,I
21170 GOSUB 14000:NEXT H:CLOSE1:RETURN
```

The complexity of this data-file module should convince you of the need to save data at regular intervals to tide you over the errors which are inevitable in entering a complex program such as this one.

MODULE 6.3.2

```
11000 REM#*******************************
11010 REM MENU
11020 REM********************************
11030 POKE 53281,13:PRINT"[illegible]
[illegible]HOME BUDGET"
11040 PRINT "[illegible]FUNCTIONS AVAILABLE:"
11050 PRINT " [illegible]1)DISPLAY MONTHLY ANALYSI
S"
11060 PRINT " 2)CHANGES"
11070 PRINT " 3)NEW BUDGET HEADINGS"
11080 PRINT " 4)DELETE BUDGET HEADING"
11090 PRINT " 5)RESET HYPOTHETICAL FIGUR
ES"
11100 PRINT " 6)RESET MONTH"
11110 PRINT " 7)DATA FILES"
11120 PRINT " 8)INITIALISE"
11130 PRINT " 9)STOP"
11140 INPUT "[illegible]WHICH DO YOU REQUIRE:";Z:
PRINT "[illegible]";:IF Z<5 THEN 11160
11150 ON Z-4 GOSUB 15000,17000,21000,120
00,11190:GOTO 11000
11160 PRINT "[illegible]1)REAL DATA":PR
INT "2)HYPOTHETICAL DATA"
```

```
11170 INPUT "█WHICH DO YOU REQUIRE:";H:I
F H<1 OR H>2 THEN 11170
11180 PRINT "█":H=H-1:ON Z GOSUB 13000,1
9000,16000,20000:GOTO 11000
11190 PRINT "████████████████████HOME B
UDGET TERMINATED":END
```

A standard menu module with the addition of the facility to define whether the real or hypothetical side of the arrays is being addressed. The distinction between these two will be explained later.

MODULE 6.3.3

```
12000 REM#********************************
12010 REM INITIALISE
12020 REM*********************************
12030 CLR
12040 DIM PA$(1,29),MO(1,29),PA(1,29,11)
,PT(1,11),BD(1,11),C1(1,11),BA(1,11)
12050 DIM C2(1,11):R$=CHR$(13)
12060 DATA JANUARY,FEBRUARY,MARCH,APRIL,
MAY,JUNE,JULY,AUGUST,SEPTEMBER
12070 DATA OCTOBER,NOVEMBER,DECEMBER
12080 DIM MO$(11):RESTORE:FOR I=0 TO 11:
READ MO$(I):NEXT
12090 INPUT "██ARE YOU LOADING FROM TAPE
 (Y/N):";Q$:IF Q$="Y" THEN GOTO 11000
12100 INPUT "███INPUT NUMBER OF CURRENT M
ONTH:";MM:MM=MM-1:Y=MM+11
12110 GOSUB 18000:GOSUB 16000:GOTO11000
```

Initialisation module.

MODULE 6.3.4

```
18000 REM#********************************
18010 REM INCOME
18020 REM*********************************
18030 PRINT "████INPUT MAIN INCOME AS FOL
LOWS:"
18040 FOR I=MM TO Y:I1=I:IF I1>11 THEN I
1=I1-12
18050 PRINT MO$(I1);":":INPUT "█████████
█████";C1(H,I1):NEXT I
```

```
18060 PRINT "▮▮OTHER ANTICIPATED INCOME:
"
18070 FOR I=MM TO Y:I1=I:IF I1>11 THEN I
1=I1-12
18080 PRINT MO$(I1);":":INPUT "▮▮▮▮▮▮▮▮▮
▮▮▮▮";C2(H,I1):NEXT I
18090 GOSUB 14000:RETURN
```

This module accepts input of income under the headings of main income and supplementary income.

*Commentary*

Line 18040: Whereas data is stored in the arrays in the order January-December, the 12 month period which the program is capable of covering can begin in any calendar month. Accordingly, the variable I1 is used to ensure that when the twelfth month is completed, the loop moves on to address the first month of the calendar year.

Line 18050: Note the way in which the variable H is used to determine which side of the arrays is addressed.

*Testing Modules 6.3.2–6.3.4*

By inserting temporary RETURNs at 14000 and 16000, you should now be able to input income data for the 12 months from your chosen starting month. For the moment, stick to inputting to the real side of the arrays—all will be made clear later.

MODULE 6.3.5

```
16000 REM#*******************************
16010 REM INPUT OF PAYMENTS
16020 REM********************************
16030 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮INPUT OF BILLS
:"
16040 PRINT"▮PRECEDE NAME OF ITEM WITH A
 '*' IF YOU","DO NOT WANT IT BUDGETED."
16050 PRINT "▮▮('ZZZ' TO QUIT)"
16060 INPUT "▮▮HEADING FOR BILL:";Q$:IF
Q$="ZZZ" THEN GOSUB 14000:RETURN
16070 N(H)=N(H)+1
16080 IF N(H)=30 THEN N(H)=29:PRINT "NO
MORE ROOM":FOR I=1 TO 2000:NEXT:RETURN
16090 PA$(H,N(H)-1)=Q$:PRINT "▮▮PAYMENTS
 UNDER ";Q$;":▮"
```

```
16100 FOR I=MM TO Y:I1=I:IF I1>11 THEN I
1=I1-12
16110 PRINT MO$(I1):INPUT "[illegible]"
;PA(H,N(H)-1,I1):NEXT I
16120 GOTO 16000
```

This module accepts the input of bill headings and associated amounts.

*Commentary*

Line 16040: The program has the facility to calculate an average monthly figure which will cover the yearly total of payments under any payment headings. Preceding the payment name with a * excludes the particular payment from this process—ie it is treated as a one-off item.

Line 16070: The variable H is used to increment one or other side of the 2 element array N, which records the number of payments stored on each side of the array.

Lines 16090–16110: Having specified the payment title, input is requested for each of the 12 months in the period covered.

*Testing Module 6.3.5*

You should now be able to input a number of bills and find them stored in the zero side of arrays PA$ and PA—again sticking to the real side of the arrays. The temporary RETURN at 14000 should be retained for this test.

MODULE 6.3.6

```
14000 REM#******************************
14010 REM UPDATE BUDGET
14020 REM*******************************
14030 T(H)=0
14040 FOR I=0 TO N(H)-1:BU=0:IF LEFT$(PA
$(H,I),1)="*" THEN 14060
14050 FOR J=0 TO 11:BU=BU+PA(H,I,J):NEXT
:MO(H,I)=BU/12:T(H)=T(H)+MO(H,I)
14060 NEXT I
14070 TT=0:CU=0:FOR I=MM TO Y:I1=I+12*(I
>11):PT(H,I1)=0
14080 FOR J=0 TO N(H)-1:PT(H,I1)=PT(H,I1
)+PA(H,J,I1):NEXT J:TT=TT+PT(H,I1)
14090 FOR J=0 TO N(H)-1:IF LEFT$(PA$(H,J
),1)="*" THEN TT=TT-PA(H,J,I1):NEXT J
14100 BD(H,I1)=T(H)*(I-MM+1)-TT:CU=CU+C1
```

```
(H,I1)+C2(H,I1)-PT(H,I1):BA(H,I1)=CU
14110 NEXT I:RETURN
```

This module performs all the calculations necessary for the construction of the table of figures we are working towards.

*Commentary*

Lines 14040–14060: Monthly average budget figures are calculated and stored in the array MO. The cumulative total for these figures is stored in the array T. The process is not carried out for payment headings commencing with a *.

Line 14070: Note the use of the logical condition (I> 11) to calculate the value of I1. If I is less than or equal to 11 then this condition will have a value of zero and will make no difference to the value of I1. When I is greater than 11 the condition will take on the value minus one and can be used to reset I1.

Line 14080: The total of all the bills to be paid in a particular month are cumulated in the relevant line of the array PT. TT is used to hold the cumulative total of these monthly totals.

Line 14090: From TT are now subtracted the amounts associated with any items that are not to be included in the average budget calculations. TT now contains the cumulative total of items which are included in the average budget calculation.

Line 141000: The balance of the budgeted figure over actual payments is now stored in the array BD by multiplying the average monthly payment by the number of months and subtracting the actual payments on budgeted items up to the relevant month. The balance of the two forms of income over the total actual payments for the month is stored in the array BA.

*Testing Module 6.3.6*

It is difficult to fully test this module until the module which displays the table has been entered, but it is a good idea to enter some data since this will call up the module and check the syntax for you. If you are confident that the module is functioning correctly, then it is a good idea to save the data you have input on tape.

MODULE 6.3.7

```
22000 REM#****************************
22010 REM FUNCTIONAL SUBROUTINES
22020 REM*****************************
22030 MY=INT(ABS(MY)+10000):MY$=MID$(STR
$(MY),3):IF MY>=20000 THEN MY$="####"
22040 RETURN
```

A formatting routine which returns a four digit number with leading zeros in necessary. If the figure being processed is greater than £9999 it is displayed as '####' to show that it is outside the range that can be accurately displayed by the program. The program calculations are unaffected by this.

MODULE 6.3.8

```
13000 REM#****************************
13010 REM DISPLAY FIGURES
13020 REM*****************************
13030 PRINT "▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓WORKSHEET"
13040 INPUT"NUMBER OF MONTH TO START:";M
1:IF M1<1 OR M1>11 THEN 13040
13050 M1=M1-1:IF MM-M1-12*(M1>MM-1)<4 TH
EN M1=MM-4-12*(MM<5)
13060 PRINT "▓▓▓ ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒"
13070 PRINT" ▓▓▓MONTH        ";
13080 FORJ=M1 TO M1+3:PRINT "▓▓▓";LEFT$
(MO$(J+12*(J>11)),3);
13090 NEXT J:PRINT "▓▓ ▓▓ ▓▓"
13100 PRINT "▓ ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒"
13110 FOR I=0 TO N(H)-1:IF I<>15 THEN 13
140
13120 INPUT "▓▓RETURN▓ TO CLEAR SCREEN A
ND CONTINUE:";Q$:PRINT "▓▓▓▓"
13125 FOR J=1 TO 20
13130 PRINT "
               ":NEXT J:PRINT "▓▓▓▓"
13140 PRINT " ▓▓▓";LEFT$(PA$(H,I),12):PR
INT "▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓";
13150 FOR J=M1 TO M1+3:PRINT "▓▓▓";:MY=I
NT(PA(H,I,J+12*(J>11))):GOSUB 22030
```

```
13160 PRINT MY$;:NEXT J:PRINT "▮▒▮";
13170 MY=INT(MO(H,I)):GOSUB 22030:PRINT
MY$:NEXT I
13180 PRINT " ▮▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒"
13190 INPUT "▮PRESS ▮RETURN▮ TO DISPLAY
ANALYSIS:";Q$
13200 PRINT"▮▮▮▮":FOR I=1 TO 20:PRINT "
                                        "
13210 NEXT I:RESTORE:FOR J=1 TO 12:READ
A$:NEXT:PRINT "▮▮▮▮▮"
13220 DATA TOTAL,BUDGET,BUDGET BAL.,MAIN
 INCOME, SUPP. INCOME,TOTAL INCOME
13230 DATA CASH BALANCE,CUM. BALANCE
13240 FOR I=1 TO 8:READ A$:PRINT " ▮▒▮";
A$
13250 PRINT "▮ ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒":NEXT I
13260 FO$="▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮"
13270 FOR I=M1 TO M1+3:I1=I+12*(I>11):PR
INT "▮▮▮▮"
13280 PRINT FO$;"▮▒";:MY=PT(H,I1):PRINT
"▮";:IF MY<0 THEN PRINT "▮";
13290 GOSUB 22030:PRINT MY$:PRINT
13300 PRINT FO$;"▮▒";:MY=T(H):PRINT "▮";
:IF MY<0 THEN PRINT "▮";
13310 GOSUB 22030:PRINT MY$:PRINT
13320 PRINT FO$;"▮▒";:MY=BD(H,I1):PRINT
"▮";:IF MY<0 THEN PRINT "▮";
13330 GOSUB 22030:PRINT MY$:PRINT
13340 PRINT FO$;"▮▒";:MY=C1(H,I1):PRINT
"▮";:IF MY<0 THEN PRINT "▮";
13350 GOSUB 22030:PRINT MY$:PRINT
13360 PRINT FO$;"▮▒";:MY=C2(H,I1):PRINT
"▮";:IF MY<0 THEN PRINT "▮";
13370 GOSUB 22030:PRINT MY$:PRINT
13380 PRINT FO$;"▮▒";:MY=MY+C1(H,I1)-100
00:PRINT "▮";:IF MY<0 THEN PRINT "▮";
13390 GOSUB 22030:PRINT MY$:PRINT
13400 PRINT FO$;"▮▒";:MY=MY-PT(H,I1)-100
00:PRINT "▮";:IF MY<0 THEN PRINT "▮";
13410 GOSUB 22030:PRINT MY$:PRINT
13420 PRINT FO$;"▮▒";:MY=BA(H,I1):PRINT
"▮";:IF MY<0 THEN PRINT "▮";
```

```
13430 GOSUB 22030:PRINT MY$:PRINT
13440 FO$=FO$+"█████":NEXT I
13450 INPUT "█DO YOU WISH TO REVIEW FIGU
RES (Y/N):";Q$
13460 IF Q$="Y" THEN 13060
13470 RETURN
```

In the last program we noted that display modules are often the most complex of a program whose task is to present a table of data, and this one is certainly no exception. Having said that, it should be noted that beneath the superficial complexity this is a relatively simple module which picks up figures which have already been calculated and places them on the screen. It looks complex only because of the sheer number of figures which are to be displayed.

*Commentary*

Lines 13040–13050: The table displays the figures for four months from the month specified by the user. However, running over the end of the current 12 month period would make a nonsense of the table so, if a figure less than four months from the end of the period is input, the start month is reset.

Lines 13060–13100: The heading of the table is printed, consisting of the first three letters of the relevant months and a heading for the 'average budget' column.

Lines 13100–13170: Payment names are obtained from the array PA$ and printed in the left-hand column. Following the name, the figures for the four months and the average budget figure for the item are printed across the screen, separated by graphics characters into columns, with the previous module being used to format the amounts with leading zeros if necessary. Fifteen lines are printed, with provision to clear the screen and print another 15 if that is not sufficient. The program can handle up to 30 payment headings.

Lines 13210–13240: The titles for the figures given in the second part of the table are read from the DATA statements and printed down the left hand side of the screen—the table heading remaining undisturbed (the budget column heading is now redundant but is not erased).

Lines 13260–13440: Despite its length, a simple routine which, using the string FO$ to determine the position of the column, prints the relevant figures for each month down the screen opposite their headings. At the end of each month's column, five cursor right characters are added to FO$

and the process is repeated in a fresh column for the next month. Note the use of the red and black control characters to show whether an item is positive or negative. Note also, in Lines 13380 and 13400, the temporary variable MY, from the formatting module, is used to add a figure to one previously printed. To do this the 10000 which was added in the formatting process must first be subtracted.

*Testing Modules 6.3.7–6.3.8*

If you have some data stored you should now be able to display it on the screen. To check the table (apart from the fact that it is properly displayed) you must understand what the various figures mean:

| | |
|---|---|
| TOTAL | This is the total of all payments to be made in the month. |
| BUDGET | The same for each month, this is the average sum that will have to be set aside in order to cover all the non-excluded bills in the 12 month period. An average budget will not necessarily cover all the payments up to any particular month (eg if all the payments were made in the first month). This figure records whether the amount set aside in the average budget is ahead or behind the actual payments it is meant to cover. At the end of the 12 month period it will be zero. |
| MAIN INCOME/SUPP. INCOME/TOTAL INCOME | These are self explanatory. |
| CASH BALANCE | The difference between income and outgoings for the relevant month. |
| CUM. BALANCE | The difference between total income and total payments since the beginning of the 12 month period. |

Note that there will be small discrepancies since only integer figures are displayed, while the actual calculations are performed on the full figures. Thus the monthly budget for a payment of £47 will be displayed as £3 but this will not affect the proper calculation of the total monthly budget.

MODULE 6.3.9

```
19000 REM#*******************************
19010 REM CHANGES
19020 REM********************************
19030 PRINT "[illegible]CHANG
ES"
19040 PRINT "[illegible]COMMANDS AVAILABLE:"
```

```
19050 PRINT " ■1)CHANGE BUDGET HEAD"
19060 PRINT " 2)CHANGE MAIN INVOME"
19070 PRINT " 3)CHANGE ADDITIONAL INCOME
"
19080 INPUT "■■WHICH DO YOU REQUIRE:";QQ
:ON QQ GOSUB 19100,19190,19190
19090 GOSUB 14000:RETURN
19100 INPUT "■■NAME OF BUGET HEAD TO BE
CHANGED:";Q$
19110 FOR I=0 TO N(H)-1:IF Q$<>PA$(H,I)
THEN 19130
19120 PP=I:GOTO 19140
19130 NEXT I:PRINT "■ITEM NOT FOUND":FOR
 I=1 TO 2000:NEXT I:RETURN
19140 PRINT "■■";PA$(H,PP):PRINT "■■INPU
T NEW AMOUNT OR '*' TO LEAVE:■"
19150 FOR I=MM TO Y:I1=I+12*(I>11)
19160 PRINT MO$(I1):PRINT "■■■■■■■■■■■■■
:";PA(H,PP,I1);:INPUT Q$
19170 IF Q$<>"*" THEN PA(H,PP,I1)=VAL(Q$
)
19180 NEXT I:RETURN
19190 IF QQ=2 THEN PRINT "■■■MAIN INCOME
:";
19200 IF QQ=3 THEN PRINT "■■■ADDITIONAL
INCOME:";
19205 PRINT " ('*' LEAVES UNCHANGED)"
19210 FOR I=MM TO Y:I1=I+12*(I>11)
19220 PRINT MO$(I1):PRINT "■■■■■■■■■■■■■
:";
19230 IF QQ=2 THEN PRINT C1(H,I1);
19240 IF QQ=3 THEN PRINT C2(H,I1);
19250 INPUT Q$:IF Q$<>"*" AND QQ=2 THEN
C1(H,I1)=VAL(Q$)
19260 IF Q$<>"*" AND QQ=3 THEN C2(H,I1)=
VAL(Q$)
19270 NEXT I:RETURN
```

If a change to an item already entered is required this module which allows the user to specify whether the item to be changed is a payment heading, main or supplementary income. The relevant figures are then displayed and the user can either confirm each figure by entering a * or entering a new value.

MODULE 6.3.10

```
20000 REM#******************************
20010 REM DELETE BUDGET HEAD
20020 REM*******************************
20030 INPUT "NAME OF BUDGET HEAD TO DE
LETE:";Q$
20050 FOR I=0 TO N(H)-1:IF Q$=PA$(H,I) T
HEN 20080
20070 NEXT I:PRINT "ITEM NOT FOUND":FO
R J=1 TO 2000:NEXT J:RETURN
20080 N(H)=N(H)-1:FOR J=I TO N(H)-1:PA$(
H,J)=PA$(H,J+1)
20090 FOR K=0 TO 11:PA(H,J,K)=PA(H,J+1,K
):NEXT K,J:GOSUB 14000:RETURN
```

This module allows any budget heading to be deleted.

MODULE 6.3.11

```
17000 REM#******************************
17010 REM REGISTER MONTH
17020 REM*******************************
17030 PRINT "UPDATE MONTH"
17040 INPUT "INPUT NUMBER OF CURRENT M
ONTH:";M2:IF M2<0 OR M2>12 THEN 17040
17050 M2=M2-1:IF M2=MM THEN RETURN
17060 IF M2<MM THEN M2=M2+12
17070 FOR I=MM TO M2:I1=I+12*(I>11)
17080 PRINT "UPDATE MONTH"
17090 PRINT "INPUT IN FULL AMOUNTS FOR
 NEXT ";MO$(I1);":"
17100 FOR J=0 TO N(0)-1:PRINT PA$(0,J);"
 (";PA(0,J,I1);"):";:INPUT PA(0,J,I1)
17110 NEXT J
17120 INPUT "MAIN INCOME:";C1(0,I1)
17130 INPUT "ADDITIONAL INCOME:";C2(0,
I1):NEXT I
17140 MM=M2+12*(M2>11):Y=MM+11:H=0:GOSUB
 14000:GOSUB 15000:RETURN
```

The purpose of this module is to allow for changes of month. When the user specifies that the current month has changed then new figures are requested for each payment heading and the income types, for each of the months which have passed and are now to be tagged onto the end of the 12 month period. Thus if the old period began with May and the new one begins with July, then the user will be requested to input figures for May and June only, since these now become the last two months of the 12 month period.

*Testing Module 6.3.11*

You should now be able to change the figures for payment headings or income, to delete payment headings and to change the period which the program is set to cover. To test the last module you will need to insert a temporary RETURN at line 15000.

MODULE 6.3.12

```
15000 REM#****************************
15010 REM SET UP SHADOW ARRAYS
15020 REM*****************************
15030 T(1)=T(0)
15040 FOR I=0 TO N(0)-1:PA$(1,I)=PA$(0,I
):MO(1,I)=MO(0,I)
15050 FOR J=0 TO 11:PA(1,I,J)=PA(0,I,J):
NEXT J,I
15060 FOR J=0 TO 11:PT(1,J)=PT(0,J):BD(1
,J)=BD(0,J):C1(1,J)=C1(0,J)
15070 BA(1,J)=BA(0,J):NEXT J:N(1)=N(0):R
ETURN
```

This simple module is one of the most important in the program. What it does is to copy the data you have input to the real side of the arrays, into the hypothetical side. One of the main points of this program is that you can choose to input data to the hypothetical side of the arrays to test the effects of a financial decision, and this will have no effect whatsoever on the real data.

All the operations of the program can be performed on hypothetical data and, when you are satisfied, all you have to do is to call up this module and the data in the hypothetical side is instantly reset to the real data. This module is automatically called up when the month is reset, otherwise the two sides of the tables would be working on different periods.

*Testing Module 6.3.12*

In fact you can now test the hypothetical sides of all the functions simply by specifying hypothetical data when the functions are called up. Add and subtract items from both sides, then use the table display to check that neither side is affecting the other. Then use this module to copy the real data into the hypothetical side. Note that the hypothetical side is empty on first initialising the program.

If the hypothetical side functions work properly then the program is ready for use.

## *Summary*

This long program is powerful, properly used, although it takes practice to get the most out of it. Taken seriously it can give you some surprising information about the state of your finances throughout the year—when things will be tight and when there might be a little to spread around,how payments might be re-arranged to ensure a little more at Christmas or for holidays, what might be the overall effect of a new commitment or of increased income.

Remember, however, that this book is intended to set your 64 to work for you. If you have successfully overcome the problems of debugging this program then there is no reason why you should not go on to adapt it to other uses which require flexible input and manipulation of data, together with clear presentation in the form of tables and the possibilities of running two sets of data at the same time. The program can be looked upon as a foundation for putting your 64 and your new found confidence to work.

## *Going further*

1) The program might be more useful if you had the facility to copy the hypothetical arrays into the real ones, once you decide to go ahead with something you have assessed. This should only involve a tiny change to one module.
2) Savings in the length of the program could result from cutting down the number of arrays by packing the same amount of data into fewer but more complex arrays. You might then be able to print the data with a small number of loops.
3) If you wish to change only a single value for a payment or for income, you have to work through all twelve payments. Try adding the facility to jump into the middle of the period and to escape from the series when you have completed the change you want to make.

# CHAPTER 7
## Music

One of the joys of the 64 is the way in which the quality and sheer cleverness of the sound capabilities open up a whole new world of possibilities for home micro- owners. In the not too distant future whole books will no doubt be written on the uses of the 64's Sound Interface Device (SID) chip.

The sheer complexity of the SID chip's capabilities means that no one program can do full justice to them and one chapter of a general work cannot serve as more than an introduction to the almost infinite combinations of sounds available. Having said that, however, the program presented here is one which provides a firm foundation for future experimentation and creation. The purpose of the program is not simply to allow the user to input and play tunes (which it does) but to allow every part of the SID to be directly available to the user. Most things that the SID is capable of achieving can be done quite simply using the program as a tool.

The first thing to remember is that a normal musical note is not simply a vibration of a certain frequency, it is in fact a combination of different frequencies, high and low. To create a note therefore requires the input of two separate frequencies, one high and one low. Each of the SID's three voices has provision for these two inputs for each note that is played. The program must be capable of accepting notes in a way comprehensible to the user and then translating the notes into a form usable by the SID.

Secondly, the intensity of any particular note varies in a complex way as the note is played:

a) The first phase of the note is known as the ATTACK. This is the speed with which the sound rises from nothing to its peak. The shorter the period of the attack, the more twangy the quality of the note.
b) The second phase is called DECAY and during this phase the note falls away from the original peak.
c) After this first falling away, the note enters the SUSTAIN phase, which determines the length of the main body of the note.
d) Lastly, the note fades away in the RELEASE phase which, like the ATTACK, can be sharp or gradual.

Different musical instruments have different qualities of tone, quite independent from the notes they play and the shape of those notes. These differences depend upon the waveform of the sound produced by the instrument.

The SID permits each of its three voices to produce any one of three musical waveforms and another white noise waveform which is useful in the creation of sound effects.

Of the three musical waveforms one, the pulse waveform, is itself capable of a considerable degree of variation by changing the length of the pulses which go to make up the waveform.

Having finally produced the desired frequency, tone and shape of note, the SID chip allows the notes to be filtered. This means that different frequencies within the note can be reduced in loudness, while others are left untouched.

**Music: Table of variables**

| | |
|---|---|
| FI%(3): | Filter characteristics for the three voices. |
| HF%(2, 1000): | High frequency values for each note in tune to be played. |
| IN: | Initialisation pointer. |
| LF%(2, 1000): | Low frequency values for each note in tune to be played. |
| R$: | Separator for data files. |
| NL: | Length of note. |
| NO%(1,95): | High and low frequency values for the 96 notes available. |
| NT: | Note value taken from Appendix M of user manual. |
| VO%(2,6): | User-defined values to be POKEd into the SID chip to determine sound characteristics of the three voices. |
| VS(: | Address of the start of a voice in the SID. |
| WF%(2, 1000): | Waveform values for each note to be played. |
| WW: | Waveform value for each individual note. |

MODULE 7.1.1

```
11000 REM#*******************************
11010 REM MENU
11020 REM********************************
11030 POKE 53281,15:PRINT "[illegible]
[illegible]MUSIC"
11040 PRINT "[illegible]COMMANDS AVAILABLE:"
11050 PRINT "[illegible]1)SET VOICE"
11060 PRINT "[illegible]2)PLAY PRESENT TUNE"
11070 PRINT "[illegible]3)COMPILE TUNE"
11080 PRINT "[illegible]4)DATA FILES"
11090 PRINT "[illegible]5)INITIALISE"
11100 PRINT "[illegible]6)STOP
```

```
11110 INPUT "■■WHICH DO YOU REQUIRE:";Z:
PRINT "□";
11120 IFIN=0AND(Z<5)THENPRINT"■NOTINITIA
LISED!":FORI=1TO2000:NEXT:GOTO 11000
11130 ON Z GOSUB 15000,14000,13000,17000
,12000,11140:GOTO 11000
11140 PRINT "▯▯▯▯▯▯▯▯▯▯▯▯▯■■■■■■■■■■■■■■
■■MUSIC TERMINATED":END
```

A standard menu module.

MODULE 7.1.2

```
18000 REM********************************
18010 REM DATA FOR NOTE TABLE
18020 REM********************************
18030 REM NOTE FREQUENCIES
18040 DATA 268,284,301,318,337,358,379,4
01,425,451,477,506
18050 DATA 536,568,602,637,675,716,758,8
03,851,902,955,1012
18060 DATA 1072,1136,1204,1275,1351,1432
,1517,1607,1703,1804,1911,2025
18070 DATA 2145,2273,2408,2551,2703,2864
,3034,3215,3406,3608,3823,4050
18080 DATA 4291,4557,4817,5103,5407,5728
,6069,6430,6812,7217,7647,8101
18090 DATA 8583,9094,9634,10207,10814,11
457,12139,12860,13625,14435,15294,16203
18100 DATA 17167,18188,19269,20415,21629
,22915,24278,25721,27251,28871,30588
18110 DATA 32407
18120 DATA 34334,36376,38539,40830,43258
,45830,48556,51443,54502,57743,61176
18130 DATA 64814
```

The data in this table is simply a shorthand way of entering the high and low frequency note values. Each number represents 256 times the high frequency note value plus the low frequency value.

MODULE 7.1.3

```
12000 REM#****************************
12010 REM SET UP TABLES
12020 REM*****************************
12030 CLR:DIM NO%(1,95):FOR I=0 TO 95:RE
AD NN:NO%(0,I)=INT(NN/256)
12040 NO%(1,I)=NN-256*INT(NN/256):NEXT
12050 DIM VO%(2,6),FI%(3),LF%(2,1000),HF
%(2,1000),WF%(2,1000)
12070 IN=1:R$=CHR$(13)
12080 GOTO 11000
```

The use of the main variables defined here is explained in the table of variables.

*Commentary*

Lines 12030–12040: High and low frequency values for each of the 95 notes are read and decoded. They cannot be stored in single number form in the array, since it is an integer array and can only hold numbers up to 32767. The note values are placed into NO%(0) for high and NO%(1) for low.

*Testing Module 7.1.3*

After calling up this module, you should be able to read from the table high and low frequency values approximately the same as those in Appendix M of the User's Manual. Note that they will not be exactly the same since the values used here are taken from the *Programmers Reference Manual* whose table differs slightly.

MODULE 7.1.4

```
15000 REM#****************************
15010 REM VOICE SETTINGS
15020 REM*****************************
15030 INPUT "■■■VOICE NUMBER (1-3):";V:I
F V<1 OR V>3 THEN 15030
15040 VS=54272+7*(V-1)
15050 PRINT "■■■■■■■■■■■■■■VOICE";V
15060 REM*****************************
15070 T1$="PULSE W/F WIDTH (LOW:0-255):"
15080 PRINT "■";T1$;VO%(V-1,2);:Q$=""
15090 INPUT Q$:IF Q$<>"" THEN VO%(V-1,2)
=VAL(Q$)
15100 REM*****************************
15110 T1$="PULSE W/F WIDTH (HIGH:0-15):"
15120 PRINT T1$;VO%(V-1,3) AND 15;:Q$=""
```

```
15130 INPUT Q$:IF Q$<>"" THEN VO%(V-1,3)
=VAL(Q$)
15140 REM********************************
15150 T1$="RANDOM NOISE W/F (1=ON/0=OFF)
:"
15160 PRINT T1$;(VO%(V-1,4) AND 128)/128
;:Q$=""
15170 INPUT Q$:IF Q$<>"" THEN VO%(V-1,4)
=(VO%(V-1,4) AND 126) OR (VAL(Q$)*129)
15180 REM********************************
15190 T1$="PULSE W/F (1=ON/0=OFF):"
15200 PRINT T1$;(VO%(V-1,4) AND 64)/64;:
Q$=""
15210 INPUT Q$:IF Q$<>"" THEN VO%(V-1,4)
=(VO%(V-1,4) AND 190) OR (VAL(Q$)*65)
15220 REM********************************
15230 T1$="SAWTOOTH W/F (1=ON/0=OFF):"
15240 PRINT T1$;(VO%(V-1,4) AND 32)/32;:
Q$=""
15250 INPUT Q$:IF Q$<>"" THEN VO%(V-1,4)
=(VO%(V-1,4) AND 222) OR VAL(Q$)*33
15260 REM********************************
15270 T1$="TRIANGLE W/F (1=ON/0=OFF):"
15280 PRINT T1$;(VO%(V-1,4) AND 16)/16;:
Q$=""
15290 INPUT Q$:IF Q$<>"" THEN VO%(V-1,4)
=(VO%(V-1,4) AND 238) OR VAL(Q$)*17
15300 REM********************************
15305 T1$="DISABLE THIS VOICE (1=YES/0=N
O):"
15310 PRINT T1$;(VO%(V-1,4) AND 8)/8;:Q$
=""
15320 INPUT Q$:IF Q$<>"" THEN VO%(V-1,4)
=(VO%(V-1,4) AND 246) OR VAL(Q$)*9
15330 REM********************************
15340 T1$="RING MOD."+STR$(V)+" WITH"+ST
R$(V-1-3*(V=1))+"(1=ON/0=OFF):"
15350 PRINT T1$;(VO%(V-1,4) AND 4)/4;:Q$
=""
15360 INPUT Q$:IF Q$<>"" THEN VO%(V-1,4)
=(VO%(V-1,4) AND 250) OR VAL(Q$)*5
15370 REM********************************
15380 T1$="SYNCHRONISE"+STR$(V)+" WITH"+
STR$(V-1-3*(V=1))+"(1=ON/0=OFF):"
```

```
15390 PRINT T1$;(VO%(V-1,4) AND 2)/2;:Q$
=""
15400 INPUT Q$:IF Q$<>"" THEN VO%(V-1,4)
=(VO%(V-1,4) AND 253) OR VAL(Q$)*2
15410 REM********************************
15420 T1$="ATTACK CYCLE (0-15):"
15430 PRINT T1$;(VO%(V-1,5) AND 240)/16;
:Q$=""
15440 INPUT Q$:IF Q$<>"" THEN VO%(V-1,5)
=(VO%(V-1,5) AND 15) OR VAL(Q$)*16
15450 REM********************************
15460 T1$="DECAY CYCLE (0-15):"
15470 PRINT T1$;VO%(V-1,5) AND 15;:Q$=""
15480 INPUT Q$:IF Q$<>"" THEN VO%(V-1,5)
=(VO%(V-1,5) AND 240) OR VAL(Q$)
15490 REM********************************
15500 T1$="SUSTAIN CYCLE (0-15):"
15510 PRINT T1$;(VO%(V-1,6) AND 240)/16;
:Q$=""
15520 INPUT Q$:IF Q$<>"" THEN VO%(V-1,6)
=(VO%(V-1,6) AND 15) OR VAL(Q$)*16
15530 REM********************************
15540 T1$="RELEASE CYCLE (0-15):"
15550 PRINT T1$;VO%(V-1,6) AND 15;:Q$=""
15560 INPUT Q$:IF Q$<>"" THEN VO%(V-1,6)
=(VO%(V-1,6) AND 240) OR VAL(Q$)
15570 REM********************************
15580 T1$="FILTER LOW CUTOFF (0-7):"
15590 PRINT T1$;FI%(0) AND 7;:Q$=""
15600 INPUT Q$:IF Q$<>"" THEN FI%(0)=(FI
%(0) AND 248) OR VAL(Q$)
15610 REM********************************
15620 T1$="FILTER HIGH CUTOFF (0-255):"
15630 PRINT T1$;FI%(1);:Q$=""
15640 INPUT Q$:IF Q$<>"" THEN FI%(1)=VAL
(Q$)
15650 REM********************************
15660 T1$="FILTER RESONANCE (0-15):"
15670 PRINT T1$;(FI%(2) AND 240)/16;:Q$=
""
15680 INPUT Q$:IF Q$<>"" THEN FI%(2)=(FI
%(2) AND 15) OR VAL(Q$)*16
15690 REM********************************
15700 T1$="FILTER THIS VOICE (1=YES/0=NO
):"
15710 PRINT T1$;(FI%(2) AND 2↑(V-1))/2↑(
V-1);:Q$=""
15720 INPUTQ$:IFQ$<>"" THEN FI%(2)=(FI%(
2) AND (255-2↑(V-1))) OR VAL(Q$)*2↑(V-1)
15730 REM********************************
15740 IF V<> 3 THEN 15780
15750 T1$="CUT-OFF VOICE 3 (1=YES/0=NO):
"
15760 PRINT T1$;(FI%(3) AND 128)/128;:Q$
=""
15770 INPUT Q$:IF Q$<>"" THEN FI%(3)=(FI
%(3) AND 127) OR VAL(Q$)*128
15780 REM********************************
15790 T1$="HIGH-PASS FILTER (1=ON/0=OFF)
:"
15800 PRINT T1$;(FI%(3) AND 64)/64;:Q$="
"
15810 INPUT Q$:IF Q$<>"" THEN FI%(3)=(FI
%(3) AND 191) OR VAL(Q$)*64
15820 REM********************************
15830 T1$="BAND-PASS FILTER (1=ON/0=OFF)
:"
15840 PRINT T1$;(FI%(3) AND 32)/32;:Q$="
"
15850 INPUT Q$:IF Q$<>"" THEN FI%(3)=(FI
%(3) AND 223) OR VAL(Q$)*32
15860 REM********************************
15870 T1$="LOW-PASS FILTER (1=ON/0=OFF):
"
15880 PRINT T1$;(FI%(3) AND 16)/16;:Q$="
"
15890 INPUT Q$:IF Q$<>"" THEN FI%(3)=(FI
%(3) AND 239) OR VAL(Q$)*16
15900 REM********************************
15910 T1$="VOLUME SETTING (0-15):"
15920 PRINT T1$;FI%(3) AND 15;:Q$=""
15930 INPUT Q$:IF Q$<>"" THEN FI%(3)=FI%
(3) OR VAL(Q$)
15940 FOR I=0 TO 6:IF VO%(V-1,I)>255 THE
N GOTO 15970:NEXT
15950 FOR I=0 TO 3:IF FI%(I)>255 THEN GO
TO 15970:NEXT
15960 RETURN
```

```
15970 PRINT "SORRY! THERE'S AN ERROR IN
YOUR INPUT"
15980 PRINT "PLEASE GO THROUGH THIS VOIC
E AGAIN."
15990 FOR I=1 TO 2000:NEXT:GOTO 15000
```

Though this module looks dauntingly long, it is in fact extremely simple. Its purpose is to allow the user to address all the relevant functions in the SID chip separately. The values are then stored in the arrays VO% and FI% until such time as a tune is to be played.

*Commentary*

Lines 15030–15040: V is set equal to the voice number which the user desires to set. The address 54272 is the start of the SID chip, with the main parts of each voice taking 7 bytes of data, so that the start position of the relevant voice is calculated by 15040.

Lines 15060–15130: These two routines set the pulse width if the user wishes to use the pulse waveform. The current value of each is displayed and is left unchanged if RETURN is pressed.

Lines 15140–15170: This routine sets the random noise waveform for the voice. Notice that here we are addressing, not the whole of a byte in the computer, but one bit (there are eight bits, or on-off switches, in each byte). In order to do this we make use of the AND and OR functions. To show whether a particular bit is set we print the value in the array ANDed with 2 to the power of the number of the relevant bit—the bits being numbered from 0 to 7 in increasing value from right to left. If the bit is set (on) then the same value is returned, if it is not set then the value zero is returned. In order to render the value returned either a zero or a one, it is divided by 2 to the power of the bit number. To change the value of the desired bit requires that the value of the whole byte is ANDed with 255-2↑bit number.: this results in the desired bit being set to zero, and all others being unchanged. The individual bit is now ORed with the 1 or 0 input, thus setting its value to either 0 or 1 as desired.

Line 15170: Note that though the bit we desire to set is number 7 (value 128), we actually OR the byte with 129, thus setting bit 7 and bit 0. This is because the waveform values do not actually produce a tone unless bit 0 is set.

Lines 15180–15290: These three routines perform the same function for bits 6-4, the three other waveforms.

Lines 15300–15400: These two routines allow the user to modulate the output of this voice with the waveform and note shape of another voice with often surprising results. The other voice need not be set to actually play, but it must have waveform and note shape values entered. The use of these two functions will only be discovered by experimentation.

Lines 15410–15560: These four routines allow ATTACK, DECAY, SUSTAIN and RELEASE to be set. Note that in these routines, instead of setting individual bits, we set groups of four bits. ANDing the byte value with 240, then ORing it with a value from 0-15, acts upon bits 0-3. ANDing the byte with 15, then ORing it with a value (0-15)*16 acts upon bits 4-7.

Lines 15570–15680: The frequencies at which the SID's filters operate can be set by the user using these three routines. These values will then apply to all voices for which filters are set.

Lines 15690–15890: The remaining sections allow the user to set the three types of filter available either to on or off. The high pass filter passes unchanged frequencies above the previously set value. The low pass filter performs the same function for low frequencies. The bandpass filter allows through a band of middle range frequencies. If all three filters are set then the volume of the whole note will be reduced. The user has the choice of whether any particular voice will be filtered or not.

Lines 15730–15770: In the case of voice 3 there is a special bit which allows the output of the voice to be cut off.

Lines 15910–15930: The volume at which notes are played is set for all voices simultaneously.

Lines 15940–15990: Since there are no error checks on the input of values up to this point, the contents of the arrays are checked in order to determine that there are no values greater than 255, since trying to POKE such a value into a single byte would result in the program stopping.

*Testing Module 7.1.4*

It is not possible, at this point, to fully check the module, since there is no routine to actually play a tune. However a reasonable check can be made by carefully noting the inputs made and then, with the help of the listing, printing out the values in the arrays VO% and FI% to check that they correspond with what has been input. For instance, if the maximum value is input for each prompt, with the voice being set to 1, then VO% (0,2-6) should contain 255.

MODULE 7.1.5

```
13000 REM********************************
13010 REM SET UP TUNE IN ARRAY
13020 REM********************************
13030 RESTORE:FOR I=0 TO 95:READ A:NEXT
13040 TL=0:FOR I=0 TO 2:VL=1
13050 READ NT,NL:IF NT=0 THEN 13140
13060 WW=VO%(I,4):IF NT<0 THEN NT=-NT:WW
=1
13065 NT=12*INT(NT/16)+NT-16*INT(NT/16)
13070 IF NL<>1 THEN 13090
13080 HF%(I,VL)=NO%(0,NT):LF%(I,VL)=NO%(
1,NT):WF%(I,VL)=WW:VL=VL+1:GOTO 13050
13090 FOR J=1 TO NL-1:HF%(I,VL)=NO%(0,NT
):LF%(I,VL)=NO%(1,NT):WF%(I,VL)=WW
13100 VL=VL+1:NEXT J
13110 HF%(I,VL)=NO%(0,NT):LF%(I,VL)=NO%(
1,NT):WF%(I,VL)=WW-1:VL=VL+1
13120 IF WF%(I,VL-1)<0 THEN WF%(I,VL-1)=
1
13130 GOTO 13050
13140 IF VL>TL THEN TL=VL
13150 NEXT I
13160 RETURN
```

This module takes the tune specified by the user in the form of data statements, and compiles it into a form which is playable by the SID. The reason that this is necessary is that it allows the program to cope with different note lengths. The actual notes played by the program are all the same length, dictated by a timing loop. Longer notes are played by running together a series of notes to the required length-no division between the individual parts of the notes is discernable. Note lengths can vary from voice to voice but obviously must be such that all the voices being used are co-ordinated.

To set up the data for a tune, all that is necessary is to record, for each note, its number in Appendix M of the User's Manual, and its length. The units in which length is recorded will depend upon the shortest note it is desired to play. By shortening the timing loop at Line 14180 it will be possible to play shorter notes, but this means that longer notes will have to be made up of more of the shorter units. The drawback to this is that each individual unit of a note, no matter how short or long the timing loop makes it, takes up a space in each of the arrays LF%, HF% and WF% so that as the timing loop shortens, so the memory required to hold a tune of a given length increases proportionately.

*Commentary*

Line 13030: As the voice settings and note values are changed during the development of the tune, it will be necessary to read the tune data several times. Since the table of note values is placed before the actual tune data, it is necessary to reset the DATA pointer using RESTORE and to READ through the note table to the beginning of the tune data each time. The DATA pointer cannot be set to any particular desired point in the program, it can only be reset to the beginning of the data or left pointing to the item of data following the last one read.

Line 13040: This loop will ensure that the tune data for each voice is compiled. The variable VL is used to store the length of the tune for each particular voice.

Line 13050: Note value and note length (NL) are read from the tune data. If the note value is zero, then the program takes that voice's data to be complete and moves on to the next.

Line 13060: The value for the waveform of the current voice is read from the array VO%. If the note value is a minus number, then the waveform value is reduced by 1, thus turning off bit zero of the waveform value, which will give a silence of length NL rather than a sounded note.

Lines 13070–13080: If note length is 1, then the high frenquency and waveform data are stored in the relevant arrays and the voice length is increased by 1.

Lines 13090–13130: If the note length is greater than 1, then NL-1 successive spaces in the array are filled with the frequency and waveform values. On the last unit of the note, the waveform value is reduced by 1, thus allowing the note to fade away naturally. If no data other than 0,0 are entered for a voice (ie you do not wish to use that voice) then it is possible for the waveform to have a value of -1, which would stop the program if it were attempted to POKE it in. Line 13120 checks that this does not occur.

Line 13140: The length of the tune for the current voice (VL) is compared with TL, which contains at first zero and, subsequently, the length of the longest voice/tune. This ensures that, when the tune is played, it does not stop before the full content of each voice/tune is exhausted.

*Testing Module 7.1.5*

Once again, this module cannot be fully tested until the tune is actually played. However, if some tune data such as that given in Module 8 is entered, with a waveform set for at least voice 1, calling up this module should place the correct high and low frequency values into HF% and LF% and the waveform values into WF%.

MODULE 7.1.6

```
14000 REM#******************************
14010 REM PLAY TUNE
14020 REM*******************************
14030 FOR I=54272 TO 54296:POKE I,0:NEXT
14040 FOR I=0 TO 2:VS=54272+7*I
14050 POKE VS+2,VO%(I,2)
14060 POKE VS+3,VO%(I,3)
14070 POKE VS+5,VO%(I,5)
14080 POKE VS+6,VO%(I,6)
14090 NEXT I
14100 POKE 54293,FI%(0)
14110 POKE 54294,FI%(1)
14120 POKE 54295,FI%(2)
14130 POKE 54296,FI%(3)
14140 FOR I=1 TO TL
14150 POKE 54272,LF%(0,I):POKE 54279,LF%
(1,I):POKE 54286,LF%(2,I)
14160 POKE 54273,HF%(0,I):POKE 54280,HF%
(1,I):POKE 54287,HF%(2,I)
14170 POKE 54276,WF%(0,I):POKE 54283,WF%
(1,I):POKE 54290,WF%(2,I)
14180 FOR TT=1 TO 80:NEXT TT,I
14190 FOR TT=1 TO 200:NEXT:POKE 54296,0
14200 POKE 54276,1:POKE 54283,1:POKE 542
90,1
14210 RETURN
```

This module POKEs the voice characteristics set in Module 4 into the SID, then successively POKEs in the high and low frequencies, together with the desired waveform to produce the notes that make up the tune.

*Commentary*

Line 14030: The SID chip is initialised by POKEing zero into each of its locations.

Lines 14040–14090: The voice settings specified in Module 4 are POKEd into the SID.

Lines 14100–14130: Filter settings are shared by each voice, they are POKEd in only once.

Lines 14140–14180: For each note, up to the tune length (TL), the high and low frequency values are placed into the first two bytes of each voice location of the SID, followed by the waveform into the fifth location of each voice. This activates the desired note, which is played for as long as the loop at 14180 lasts.

Line 14190: At the conclusion of the tune a slightly longer loop is used to allow the sound to die away, then the three waveforms are set to silence.

*Testing Module 7.1.6*

Quite simple. If you have entered some tune data, initialised the program, set at least one voice and then compiled the tune, you should now be able to play your creation. The specimen data given in Module 8 plays a scale of C with the first two voices.

MODULE 7.1.7

```
17000 REM#********************************
17010 REM DATA FILES
17020 REM*********************************
17030 INPUT "██POSITION TAPE THEN █RETUR
N█--";Q$
17040 PRINT "██1)SAVE":PRINT "2)LOAD"
17050 INPUT "█WHICH DO YOU REQUIRE:";Q
17060 ON Q GOTO 17070,17130:RETURN
17070 OPEN 1,1,2,"MUSIC":PRINT#1,TL
17080 FOR I=0 TO 1:FOR J=0 TO 95:PRINT#1
,NO%(I,J):NEXT J,I
17090 FOR I=0 TO 2:FOR J=0 TO 6:PRINT#1,
VO%(I,J):NEXT J,I
17100 FOR I=0 TO 3:PRINT#1,FI%(I):NEXT
17110 FOR I=0 TO 2:FORJ=0TOTL:PRINT#1,LF
%(I,J);R$;HF%(I,J);R$;WF%(I,J):NEXTJ,I
17120 CLOSE1:RETURN
17130 OPEN 1,1,0,"MUSIC":INPUT#1,TL
17140 FOR I=0 TO 1:FOR J=0 TO 95:INPUT#1
,NO%(I,J):NEXT J,I
17150 FOR I=0 TO 2:FOR J=0 TO 6:INPUT#1,
VO%(I,J):NEXT J,I
17160 FOR I=0 TO 3:INPUT#1,FI%(I):NEXT
17170 FOR I=0 TO 2:FORJ=0TOTL:INPUT#1,LF
%(I,J),HF%(I,J),WF%(I,J):NEXTJ,I
17180 CLOSE1:RETURN
```

This is a standard data-file module which allows the tune data to be stored on tape.

MODULE 7.1.8

```
19000 REM#********************************
19010 REM DATA FOR VOICE 1
19020 REM*********************************
19030 DATA 34,2,34,2,39,2,41,2,43,2,39,1
,38,1,36,2,48,1,48,1,0,0
20000 REM#********************************
20010 REM DATA FOR VOICE 2
20020 REM*********************************
20030 DATA 36,2,38,2,40,2,41,2,43,2,45,2
,47,2,48,2,0,0
21000 REM#********************************
21010 REM DATA FOR VOICE 3
21020 REM*********************************
21030 DATA 0,0
```

*Commentary*

Lines 19000–21030: Specimen data to play a scale of C. Note that if you wish to use a voice you must still enter 0,0 for it.

***Summary***

This program will only be the beginning of your adventures with the 64's sound capabilities. It is a workhorse which will allow you to develop your own music, which can then be transferred to other programs, using only the arrays in which it is stored and Module 6, which actually plays the tune.

***Going further***

Provided that you do not run into too many limitations of memory, there are several ways in which this program could be extended:

1) Why not give yourself the ability to enter the waveform for each note, rather than simply for each voice? It is simply a matter of including a third value for each note to be played.
2) If you want to play longer tunes, why not adapt the program so that it uses a variable length loop to dictate the length of the note,thus saving massively on the amount of array space needed. All that would need to be stored in the final array would be the note value and its length, the latter being used to dictate the duration of the timing loop. Unfortunately this can only be done for one voice, since the timing loop dictates the length of the note for all three voices.
3) If you want longer tunes with more than one voice, what about giving the program the capability to compile parts of a tune which can be picked up later by a playing program which has no need for the memory consuming DATA statements.

**The Working Commodore 64** is based on a collection of solid, sophisticated programs in areas such as data storage, finance, graphics, household management, education and games of skill. The programs have been designed to make the most of the CBM 64's special features.

Some of the more advanced programs include a word processor and text editor, a music and sound synthesiser program, a sprite editor and a program which allows the user to enter high resolution graphics mode. This is not available in the standard Basic.

Each of the programs is explained in detail, line by line. And each of the programs is built up out of general purpose subroutines and modules which, once understood, can form the basis of any other programs you need to write.

Advanced programming skills spring out of the discussion explaining each subroutine. The collection also leaves you with a wide range of practical applications programs which might otherwise only be available on cassette.

The author, David Lawrence, is the author of several books on home computing and is a regular contributor to *Popular Computing Weekly*.

**Sunshine Books**

**£5.95 net**

ISBN 0 946408 02 5