David Lawrence shows how the full power of the QL can be released in SuperBASIC. The QL introduces a new generation of high-powered home computers. With it is a need for a new approach to programming.

The areas covered by the programs in this book include home finance and tax, information storage and retrieval, household and diary management, creative graphics and effective display techniques, music, education and a collection of smaller programs which all perform useful functions but mostly show off the QL's immense abilities.

All the programs in this book are clearly explained and written in easily identifiable modules. The same techniques can be copied into your own programs. We also use the unique Sunshine Checksum Generator. This analyses your programs and ensures that errors can be avoided in entering them. This can save you hours of frustration.

David Lawrence is one of the most successful and popular computer authors. His books, for a wide range of home micros, have been best-sellers all over the world. He now divides his time between writing for micro owners and broadcasting. He is a regular contributor to Popular Computing Weekly.

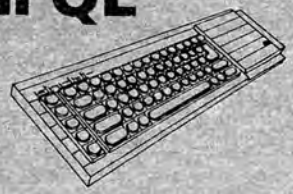
GB £ NET +006.95
ISBN 0-946408-46-7
00695
9 780946 408467

£6.95 net

SUNSHINE
ISBN 0 946408 46 7

DAVID LAWRENCE THE WORKING SINCLAIR QL SUNS 222 2634 4397

The Working Sinclair QL



A library of practical subroutines and programs

David Lawrence

The Working Sinclair QL

A library of practical subroutines and programs

David Lawrence

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street
London WC2R 3LD

British Library Cataloguing in Publication Data
Lawrence, David
The working Sinclair QL.
1. Sinclair QL (Computer) — Programming
2. Super BASIC (Computer program language)
I. Title
001.64'24 QA76.8.S625

ISBN 0–946408–46–7

Cover design by Grad Graphic Design Ltd.
Cover illustration by Stuart Hughes

CONTENTS

		Page
Program Notes		vii
Introduction		ix
1	Experiments with Time	1
	Anaclock	1
	Clock	14
	Timer	19
	Event	33
2	Son et Lumière	41
	Designer	41
	3-D Graph	56
	Screen	63
	Characters	71
	Sound Demo	87
	Music	90
3	Seriouser and Seriouser	97
	Unifile	98
	Nnumber	117
	MultiQ	129
4	Money Matters	145
	Banker	145
	Accountant	158
	Budget	171
APPENDIX: Instructions for Use of Checksum Generator Tables		193

Contents in detail

CHAPTER 1

Experiments with Time

Anaclock: runs a traditionally-faced clock in high resolution — defining a circle — Clock: provides a very different way of telling the time — Timer: provides you with 12 timers which run concurrently and are each capable of sounding an alarm and displaying a reminder message — Event: turns the QL into a stopwatch capable of giving a permanent record of times for a series of events.

CHAPTER 2

Son et Lumière

Designer: a tool which allows line drawings far larger than a single screen to be constructed, manipulated and displayed — 3-D Graph: using turtle graphics to produce a clear and attractive display for complex figures — Screen: copies the contents of the screen to a printer — the arrangement of screen memory on the QL — Characters: designs and stores your own customised character sets — character memory — Sound Demo: a simple routine to permit experiments with the sound parameters — Music: allows complex tunes to be input in a comprehensible form and played.

CHAPTER 3

Seriouser and Seriouser

Unifile: a powerful personal filing system capable of storing a wide variety of information for instant recall — Nnumber: a program which creates a dictionary of names and numbers for almost anything you wish, allowing you to create invoices, stock valuations or even a calorie count of the day's menu — MultiQ: a multiple choice test generator.

CHAPTER 4

Money Matters

Banker: allows the user to keep a clear and continuously updated record of a bank account over a 12-monthly period — Accountant: produces a set of traditionally laid out accounts — Budget: stores and processes large amounts of information about family finances and produces an analysis of the picture over a 12-month period, allowing 'what if' decisions to be

Program Notes

The programs in this book have all been word processed to improve their presentation in book form. The length of the programs has meant that it would be quite impractical to print them at a width of 37 characters, corresponding to the display generated by the QL on a standard television. In fact, the programs extend up to a maximum of 50 characters in width, with any carry over to the next line being indented to the same extent as the line start. Where a line does carry over, therefore, the spaces which make up the indentation of the second part of the line should be ignored.

Please note the section in the Introduction about how to use this book.

Debugging programs

Each program commentary includes test procedures which, if followed, should ensure that the completed program works as specified. Should you run into difficulties, however, there is always the Checksum Generator program and Checksum Tables, designed to show whether the program you have entered is the same as that listed in the book. Full details of the use of the program and tables are given in the Appendix.

Introduction

This book is part of one of the most successful series of microcomputer books ever published. In 1982, when the first 'Working Micro' book was issued, there were very few publishers who were prepared to believe that ordinary micro owners wanted to use and understand their machines, wanted to take *control* over them, learning to program by — programming. The majority of books were filled with games and trivia or consisted of yet another 'Beginners' guide to...'. The prospect of a book which set out to provide a collection of solid, useful programs and, at the same time, to give some insight into the methods employed in serious programming, didn't seem likely to set the world on fire.

But I was convinced, and the people at Sunshine Books were convinced, that the 'Working Micro' books were exactly what people were looking for, that here was a huge gap in the provision of books for the growing army of micro owners. And we were right.

Since that time, the 'Working Micro' books have followed microcomputers into almost every country where they have been sold. They have been, or are being, translated into 14 languages. *The Working Spectrum*, the first of the series, has been published in the United States and several European countries. As the book was written and as more and more people discovered the virtues of the Spectrum, it seemed that the approach of the book and the power of the machine were simply made for each other.

Now the QL heralds a new generation of Sinclair machines, which do indeed represent a Quantum Leap forwards in power, memory, and in terms of the SuperBASIC language it runs. No book can exhaust the possibilities of a machine whose capabilities would have been regarded as science fiction a few years ago, but I have enjoyed the challenge of putting it through at least some of its paces — so, I hope, will you.

How to use this book

You can use the book in a variety of different ways:

1) As a collection of useful programs which you can adapt and develop for your own purposes.
2) As a collection of subroutines out of which you can construct your own programs.

In terms of the format of the displays generated by the programs, the book is specifically aimed at those employing the QL in conjunction with a standard colour television. In deciding which combination of equipment to base a book on, I have always worked on the principle that it is easier to adapt a program to better equipment than to try and downgrade something which was written for a more sophisticated system. Owners of colour monitors will not find it hard to make adjustments to make full use of the extra screen capacity they are able to bring to bear.

However you decide to use this book, do remember that it was written *as a book* and not just as a random collection of programs in no particular order. I very often come across readers with problems because they have jumped into some of the more complex programs which fall towards the end of the book, without preparation. The earlier programs in the book, while useful and interesting in themselves, are also meant as an introduction to what comes later. Alongside the early programs goes a much higher level of explanation so that, by the time more complex programs are reached, the reader has a good grasp of some of the techniques being used.

CHAPTER 1
Experiments with Time

It is always difficult to know where to start a book like this. Too complex a program and readers may find themselves floundering before they have picked up some of the simple pointers which will make programs increasingly easy to understand as the book goes on. On the other hand, if the first programs are too trivial, many readers may not bother to discover that there are more substantial offerings to come.

As a result I have decided to stick, in this first chapter, to a set of four programs which deal with time, and the way it may be manipulated on the QL. The programs are relatively simple ones, but they introduce a wide range of concepts which will be used in later and more complex programs. In addition, in their use of calculation, sound and high resolution graphics, the programs will provide a good introduction to some of the outstanding abilities of your QL.

The programs in this chapter are:

ANACLOCK: Which runs a traditionally-faced clock in high resolution.

CLOCK: Which provides a very different way of telling the time.

TIMER: Which provides you with 12 timers which run concurrently and are each capable of sounding an alarm and displaying a reminder message.

EVENT: Which turns the QL into a stopwatch capable of giving of a permanent record of times for a series of events.

PROGRAM 1.1: ANACLOCK

Program function

The purpose of this program is produce a replica of a clock face on the screen, complete with hands which move to keep pace with the current time. In the course of following the program through, you will learn a great deal about the methods employed in this book, so it is recommended that

In terms of the format of the displays generated by the programs, the book is specifically aimed at those employing the QL in conjunction with a standard colour television. In deciding which combination of equipment to base a book on, I have always worked on the principle that it is easier to adapt a program to better equipment than to try and downgrade something which was written for a more sophisticated system. Owners of colour monitors will not find it hard to make adjustments to make full use of the extra screen capacity they are able to bring to bear.

However you decide to use this book, do remember that it was written *as a book* and not just as a random collection of programs in no particular order. I very often come across readers with problems because they have jumped into some of the more complex programs which fall towards the end of the book, without preparation. The earlier programs in the book, while useful and interesting in themselves, are also meant as an introduction to what comes later. Alongside the early programs goes a much higher level of explanation so that, by the time more complex programs are reached, the reader has a good grasp of some of the techniques being used.

CHAPTER 1

Experiments with Time

It is always difficult to know where to start a book like this. Too complex a program and readers may find themselves floundering before they have picked up some of the simple pointers which will make programs increasingly easy to understand as the book goes on. On the other hand, if the first programs are too trivial, many readers may not bother to discover that there are more substantial offerings to come.

As a result I have decided to stick, in this first chapter, to a set of four programs which deal with time, and the way it may be manipulated on the QL. The programs are relatively simple ones, but they introduce a wide range of concepts which will be used in later and more complex programs. In addition, in their use of calculation, sound and high resolution graphics, the programs will provide a good introduction to some of the outstanding abilities of your QL.

The programs in this chapter are:

ANACLOCK: Which runs a traditionally-faced clock in high resolution.

CLOCK: Which provides a very different way of telling the time.

TIMER: Which provides you with 12 timers which run concurrently and are each capable of sounding an alarm and displaying a reminder message.

EVENT: Which turns the QL into a stopwatch capable of giving of a permanent record of times for a series of events.

PROGRAM 1.1: ANACLOCK

Program function

The purpose of this program is produce a replica of a clock face on the screen, complete with hands which move to keep pace with the current time. In the course of following the program through, you will learn a great deal about the methods employed in this book, so it is recommended that you read the accompanying commentary with some care.

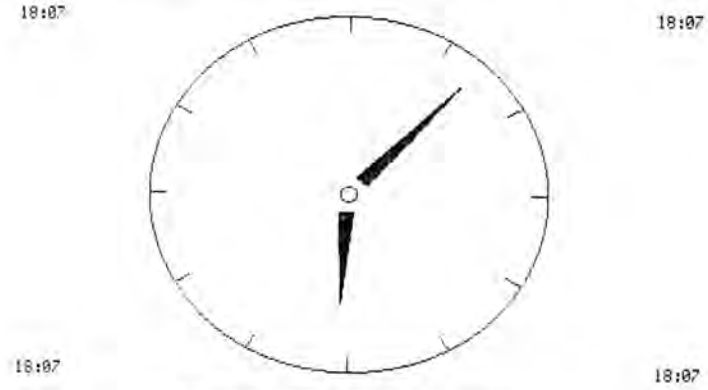


Figure 1.1: Screen Dump of Clock Face.

The ideas introduced during the course of the program include:

1) Saving programs during development.
2) Initialising variables.
3) Passing parameters to procedures.
4) Control modules.
5) The use of repeat loops.
6) Setting and reading the internal clock.
7) High resolution mode (mode 4).
8) Program flow and readability.
9) Modular programming.

Module 1.1.1: Saving the program

These four lines may seem a trivial place to start, but those who have worked with my books for earlier machines will know that this module can save an immense amount of heartache in the development of programs.

Most people learn only by bitter experience that programs *must* be saved regularly as they are developed. Sooner or later most of us reach a time when hours of work is thrown away because of a momentary surge in the power supply, a blown fuse or a knock to the micro or plug. Experienced users will have lost only some 15 minutes work because they will never have allowed more than 15 minutes to pass without SAVEing the program entered thus far.

The purpose of this module is to encourage you to make regular copies of the program you are working on by simply entering 'psave'.

The module precedes all my own programs but, since only the program name changes, it will not be included in the rest of the programs listed in the book — but you should remember to add it.

Module 1.1.1: Lines 1 – 3

```
1 DEFine PROCedure psave
2 DELETE MDV1_ANACLOCK : SAVE MDV1_ANACLOCK
3 DELETE MDV2_ANACLOCK : SAVE MDV2_ANACLOCK
4 END DEFine psave
```

Commentary

Lines 2 – 3: Deleting any previous version of the program before SAVEing ensures that the 'ALREADY EXISTS' message is not encountered. Some programmers adopt an even more cautious approach by numbering each version of the program, eg ANACLOCK01, ANACLOCK02, etc. The advantage of this is that, if something goes wrong during the SAVEing of the program so that the program is lost from memory and not properly SAVEd on the microdrive, a previous version will still be recoverable. When the program is finished, the development versions can be erased. Line 3 can be omitted by those who are prepared to rely on one microdrive to store their program, but I would recommend caution since even on the most reliable system it can be costly to have only one copy of your work.

Testing

Ensure that you have a cartridge in the drive. Enter

psave[ENTER]

and drive 1 should start up. After a moment, the light should go out on drive 1 and drive 2 should start. Finally, the flashing cursor should return to the bottom of the screen and drive 2 stop. You can now delete the three lines in memory and reload the program from the disk by entering

new[ENTER] (erases the current program)
load mdv1_anaclock[ENTER]

When the loading process has finished, list the program and the module should have been reinstated.

Module 1.1.2: Initialising the variables

Every program worth the name uses variables and constants, that is to say, labels whose values can be changed during the course of the program or at least from program to program. Very few variables absolutely *must* have their values declared when the program is first run, and people often leave it until the middle of a program, when a variable is absolutely vital, to define their values. This can be mistaken policy because, as the program is developed, it becomes increasingly difficult to see what the value of the important variables is when the program first commences. In general, it is good practice to declare at the very beginning of the program the value of

the major variables, and this process is known as 'initialisation', though exceptions are often made when memory is limited — hardly a problem on the QL. Some variables *must* be defined before they are used. For instance, you can enter a line such as:

1050 T = 50

in the middle of a program, without having given T any previous value, but if you try:

1050 A = T*2

then the QL will stop with an ERROR IN EXPRESSION message — it does not recognise the variable 'T'.

Another class of variables which must always be declared before they are used are arrays, eg T(10,10) or T$(10,10). No use whatsoever can be made of an array before a DIM statement has set up the space for it in the memory. Thus even a line such as:

T(1,1) = 50

which would work for a normal variable, will stop the program since the QL has not been told the details of the array.

Module 1.1.2: Lines 2000 – 2070

```
2000 REMark ************************
2010 DEFine PROCedure initialise
2020 REMark ************************
2030   minute2=0
2040   x1=0 : x2=0 : x3=0 : y1=0 : y2=0 : y3=0
2050   m_angle=0 : h_angle=0
2060   e_flag=0
2070 END DEFine initialise
```

Commentary

Line 2010: At the risk of boring those who are already familiar with SuperBASIC, it is probably better to mention 'procedures' briefly before we go any further, since all the programs which follow rely entirely on them.

Those of you who have programmed extensively on other machines which do not possess a structured BASIC like the QL's will know that competent programming relies on the use of subroutines to break down a program into manageable parts. It is the job of the programmer to know where each subroutine is within the program and to be able to call each up with a GOSUB command which specifies the start line number of the subroutine. SuperBASIC provides a far superior tool for programming known as the 'procedure'. Once entered into memory with a start (DEFine PROCedure) and an end (END DEFine), the *name* of the procedure becomes effectively a keyword in SuperBASIC, and entering the name will

cause the QL to carry out the procedure. In the programs which follow you will find no subroutines, and in the testing procedures you will mostly be asked to test your routines by entering their procedure names — a far more elegant way than entering meaningless GOSUBs.

The uses of the variables defined in this module will be described during the course of the commentary on the main part of the program.

Testing

Before going on, enter:

initialise[ENTER]

and the flashing cursor should return almost immediately. If you have entered any syntax errors, these will be shown up. If for any reason you clear the memory by using RUN or CLEAR, or by storing the half-finished program on microdrive and switching off, remember to initialise again *before* trying any of the tests for subsequent modules. If you do not, the modules will come across variables which have not been declared and the program will stop with an error.

Module 1.1.3: Setting the time

Before we can embark on creating a clock, we must have some means to set the time. The purpose of this module is to allow the user to input the current time in hours and minutes and have it stored in the QL's powerful internal clock. In the course of the module we shall be introduced to the use of REPEAT loops and the system variable DATE$, which records the current time and date.

Module 1.1.3: Lines 3000 – 3160

```
3000 REMark ************************
3010 DEFine PROCedure set_time
3020 REMark ************************
3030   AT 1,12 : PRINT "CLOCK SETTING"
3040   REPeat hour
3050     AT 5,1 : INPUT "HOUR (0-23):";h_temp
3060     IF h_temp >=0 AND h_temp <=23 THEN EXIT
         hour
3070   END REPeat hour
3080   REPeat minute
3090     AT 7,1 : INPUT "MINUTE (0-59):";m_temp
3100     IF m_temp>=0 AND m_temp<=59 THEN EXIT
         minute
3110   END REPeat minute
3120   temp$=DATE$
3130   time=3600*temp$(13 TO 14)+60*temp$(16 TO
       17)+temp$(19 TO 20)
```

```
3140    new_time=3600*h_temp+60*m_temp
3150    ADATE new_time-time
3160 END DEFine set_time
```

Commentary

Lines 3040 – 3070 and **3080 – 3110:** The purpose of these loops is to continue asking for the hour or minute to be input until a sensible value is received — that is a value between zero and 23 for an hour, zero and 59 for a minute. Lines 3060 and 3100 have the function of jumping out of the loop when a valid entry is made. Note that this is not an infallible method of proofing the program against operator error — simply pressing RETURN without a number will stop the program on the QL. In later programs you will see how this can be overcome but, if it does happen, simply type RETRY and the prompt will be repeated.

Lines 3120 – 3150: The actual setting of the time on the QL is quite simple, and can be done on two levels, either setting the whole date or simply the time within a particular day. When the QL is first switched on, its internal clock is set to midnight on the 1st January 1961 and it begins to count in seconds. You can see the current result of this count by entering PRINT DATE, because DATE is a variable set up by the QL itself to contain the current time in seconds. Printing out DATE, however, will illustrate that human beings do not find the time in seconds very meaningful. The QL therefore provides a second means of reading the time, DATE$. The format of DATE$ is:

YYYY MMM DD HH: MM: SS

with the month being specified in three-letter shorthand, rather than as a number.

Not only does DATE$ make reading the time possible, the system provides two methods for the user to input a new time, after which the QL will immediately reset its internal clock to the specified time.

We shall start with the shorter of the two methods, which involves adjusting the internal clock by means of the ADATE command. The function of ADATE (Adjust DATE) is to add or subtract a specified number of seconds from the current time. In order to accomplish this, line 3130 first copies the current time into the variable TEMP$. The hours, minutes and seconds digits are then multiplied and added together in order to find the total number of seconds, which are stored in the variable TIME. Note that, in doing this, we can simply multiply the strings characters containing the values, TEMP$(13 to 14) for the hours, TEMP$(16 to 17) for the minutes and TEMP$(19 to 20) for the seconds. 'Coercion', or the QL's ability to treat a string as a number when a program line requires it, takes care of the rest. Line 3140 sets the variable NEW__TIME to the value in seconds of the time just input by the user. Since, as we have already mentioned, ADATE

works by adjusting the existing time, all we have to do is to use ADATE and specify the difference between TIME and NEW__TIME. This sets the hours, minutes and seconds elements of the internal timer to what has been input by the user — it makes no difference to the year, month and day elements.

Testing

Clear the screen, then type:

set__time[ENTER]

and you should be prompted to give the time in hours and minutes. Inputting an invalid figure for either should result in the prompt being repeated. After a time has been accepted, the program will stop. You can now test what you have entered further by typing:

print date$[ENTER]

The hours, minutes and seconds should be what you have input, plus any delay in printing them out.

Module 1.1.4: Setting up the clock face

Having input the time, we come to the drawing of the face of the clock, on to which later modules will place the hands.

Module 1.1.4: Lines 4000 – 4120

```
4000 REMark ************************
4010 DEFine PROCedure clock_face
4020 REMark ************************
4030   CIRCLE 80,50,48
4040   CIRCLE 80,50,2
4050   FOR i=0 TO 330 STEP 30
4060     x1=80+COS(RAD(i))*44
4070     y1=50+SIN(RAD(i))*44
4080     x2=80+COS(RAD(i))*48
4090     y2=50+SIN(RAD(i))*48
4100     LINE x1,y1 TO x2,y2
4110   NEXT i
4120 END DEFine clock_face
```

Commentary

Defining a circle

The basis of this module and the next is the technique needed to pinpoint positions around a circle. This is based on the fact that any point on the circumference of a circle can be determined if the following pieces of information are known:

a) The radius of the circle. (RADIUS)
b) The angle that has to be travelled clockwise from the three o'clock position to arrive at the specified point. (ANGLE)
c) The coordinates of the centre of the circle. (CENTRE X and CENTRE Y)

Given these three, the position will be expressed by two formulae:

X coordinate = RADIUS*COSINE(ANGLE/180*PI)+CENTRE X
Y coordinate = RADIUS*SINE(ANGLE/180*PI)+CENTRE Y

Space does not permit us to analyse here why this should be, but any good introductory book on trigonometry will lay out the logic in full. If you want to test the technique, enter the following on your machine:

```
10 CLS
20 FOR i=0 TO 359
30 x=80+COS(RAD(i))*50
40 y=50+SIN(RAD(i))*50
50 POINT x,y
60 NEXT i
```

and you will find that a very presentable circle is drawn, with its centre at 80(x), 50(y). Here, the 80 and 50 figures represent the centre of the circle (80 across, 50 up). The only thing you may not recognise is the RAD function. In order for the QL to be able to recognise an angle it must be presented in the form of units called radians (one radian is equal to 180/PI). The RAD function performs the task of producing a result in radians when it is given a figure in degrees to work on.

Lines 4030 – 4040: One large circle to define the outside of the clock, one small one at the centre of the face, around which the hands will turn.

Lines 4050 – 4110: The circle techniques described above are used to draw in markings every 30 degrees around the circle — ie every five minutes on the clock face. The two points calculated for every repetition of the loop are first of all a point on the circumference of the large circle, and then one four pixels in towards the centre. Drawing a line between them makes a series of neat markers around the face.

Testing

Enter

clock_face [ENTER]

and you should see the clock face drawn.

Module 1.1.5: Calculating minutes and hours

In this module we get down to the real work of the program, which begins with the extraction and examination of the value for the current hour and minute from the system variable DATE$.

Module 1.1.5: Lines 5000 – 5130

```
5000 REMark ************************
5010 DEFine PROCedure time_values
5020 REMark ************************
5030   REPeat minute_test
5040     LET temp$=DATE$
5050     hour=temp$(13 TO 14)
5060     minute1=temp$(16 TO 17)
5070     IF minute1<>minute2 THEN EXIT minute_test
5080     IF INKEY$=" " THEN e_flag=1 : EXIT
         minute_test
5090   END REPeat minute_test
5100   minute2=minute1
5105   m_angle2=m_angle : h_angle2=h_angle
5110   m_angle=90-(6*minute1)
5120   h_angle=90-(30*hour+5*minute1/12)
5130 END DEFine time_values
```

Commentary

Lines 5030 – 5090: What this loop does is continually to extract the time from DATE$ and store it in the variable TEMP$. It is, of course, quite possible to work directly on DATE$ but this could result in errors if the process of slicing up the string began a fraction of a second before the hour was about to change. If the hour changed in between lines of the program being carried out, it would be possible to end up with a value of, say, 11.00, when the real time was 12.00. Having extracted the value for the hours and minutes, the loop tests the current minute against the minute value for the last time the clock hands were moved. If the current minute is different, then line 5070 jumps out of the loop. In line 5080, provision is made so that if the user presses the space bar a flag is set (the variable E_FLAG) and the loop terminates. In the normal course of the program, this loop will simply wait until the minute changes.

Line 5100: The new minute is stored until the next time this procedure is called, when it will be used as the next benchmark for a change of minute.

Line 5110: The same is done for the previous angles of the hour and minute hands. These need to be recorded in order that the existing hands can be erased before new ones are drawn.

Lines 5120 – 5130: The angles of the hour and minute hands. The QL measures circles from a point that we would call three o'clock and moves around anti-clockwise. Twelve o'clock is therefore – 90 and the minute

value must be subtracted from that — six degrees per minute. The hour angles consists of 30 degrees for each hour plus five degrees for each 12 minutes.

Testing

Enter

time_values [ENTER]

and you should almost instantaneously find that execution stops and the flashing cursor returns. This is because the value of MINUTE2 should be zero, and the loop will immediately stop when it samples the real time. MINUTE2 has now been reset to the current minute, so follow the procedure again and, unless you are unlucky enough to just catch the minute change, you should find that the QL will wait for a period before giving you back the cursor — the wait was for the new minute to arrive. If you like, you can print out the values of MINUTE1 and HOUR1, then print DATE$, illustrating that the two values have been extracted from the internal clock.

Module 1.1.6: Drawing the hands

Having calculated all the necessary figures to arrive at the time, we can now proceed to employ the QL's graphics capabilities to draw the hands of the clock. If you do not remember the general introduction to the mathematics of drawing a circle given earlier, you would be wise to go back and take a quick look at it.

Module 1.1.6: Lines 6000 – 6200

```
6000 REMark ************************
6010 DEFine PROCedure hand_draw(angle,size1,
     size2,colour)
6020 REMark ************************
6030   angle1=RAD(angle)
6040   angle2=RAD(angle+20)
6050   angle3=RAD(angle-20)
6060   y1=50+size1*SIN(angle1)
6070   x1=80+size1*COS(angle1)
6080   y2=50+size2*SIN(angle2)
6090   x2=80+size2*COS(angle2)
6100   y3=50+size2*SIN(angle3)
6110   x3=80+size2*COS(angle3)
6120   INK colour
6130   FILL 1
6140   LINE x1,y1 TO x2,y2 TO x3,y3 TO x1,y1
6150   FILL 0
6160   AT 0,0 : PRINT temp$(13 TO 17)
6170   AT 0,69 : PRINT temp$(13 TO 17)
```



```
6180   AT 19,0 : PRINT temp$(13 TO 17)
6190   AT 19,69 : PRINT temp$(13 TO 17)
6200 END DEFine hand_points
```

Commentary

Line 6010: Our first example of a procedure using parameters passed from the rest of the program. The four variable names appended to the procedure name allow the program to use the same procedure for different purposes. The four variables will be used by the procedure — in this case to determine where to draw the clock hand, how large it should be and what colour. In this way, exactly the same procedure can be used to draw hour and minute hands.

Lines 6030 – 6050: These lines define three angles. Their purpose is to allow the program to pinpoint three spots on the screen which will be the corners of a triangular clock hand. ANGLE1 is the angle of the tip of the hand, relative to the centre of the face. ANGLE2 and ANGLE3, which are respectively slightly ahead and behind ANGLE1, point to the two ends of the base of the triangle. If you are not clear on this, don't worry, all will become plain when the hands are drawn on the screen.

Lines 6060 – 6110: If you tried the example circle program earlier, you will recognise these lines. They plot points which are SIZE1 or SIZE2 pixels out from the centre of the circle, in the direction indicated by ANGLE1, ANGLE2 or ANGLE3. SIZE1 represents the distance to the tip of the hand, SIZE2 the shorter distance to the broad base of the hand.

Lines 6120 – 6150: These lines draw the hand. Its colour is passed to the procedure in the form of the variable COLOUR. The FILL command ensures that the completed triangle will be FILLed with the prevailing ink colour.

Lines 6160 – 6190: As an added touch, the time is displayed in digital format at the four corners of the screen — note that the coordinates refer to the high resolution screen which will be set up by the next module.

Testing

To set up this module for testing would require a number of temporary lines to be entered. Since the complete program requires only a few lines to be added, it is recommended that you leave testing until the program is completed.

Module 1.1.7: Making it all work together

At this stage, you may well be wondering why the program is written as it is. Could not all of the functions we have described have been put together

and made to run with the use of a few GOTOs. Unfortunately, that is how many published programs are constructed. In this book you will find that all of the programs are constructed out of clearly identifiable modules, most of them procedures in their own right.

The reason for this is that programs written in modules can be more easily read, they can be more easily debugged, they can be changed by substituting modules which work more efficiently if you learn new methods, they can be added to by patching in more modules. There is much to be learned from the programs in this book, but probably the most valuable lesson of all for your future programming will be the technique of modular programming.

The current module is the key to the technique for, when all the working modules have been entered and tested, we need one more to control the flow of the program. In a sense, the module you are about to enter *is* the program — everything else is merely an extension of it.

Module 1.1.7: Lines 1000 – 1180

```
1000 REMark **********************
1010 REMark control loop
1020 REMark **********************
1030   PAPER 2 : INK 7 : BORDER 0
1040   CLS : CLS#0
1050   initialise
1060   set_time
1070   INK 7 : PAPER 0 : MODE 4 : CLS
1080   clock_face
1090   REPeat control
1100     time_values
1110     IF e_flag THEN EXIT control
1120     hand_draw m_angle2,40,5,0
1130     hand_draw h_angle2,30,5,0
1140     hand_draw m_angle,40,5,7
1150     hand_draw h_angle,30,5,2
1160   END REPeat control
1170   MODE 8
1180   STOP
```

Commentary

Line 1040: It is always a good idea, at the beginning of a program, to clear not only the main screen but the command lines at the bottom (CLS #0).

Lines 1050 – 1060: Note how naturally the control structure develops using the names of the modules — the control structure becomes in itself a brief description of the program's work.

Line 1070: The program is designed to run in high resolution mode, mode 4. In this mode, in return for halving the number of possible colours on the screen at one time, we shall get a sharper definition of the clock face.

Lines 1090 – 1160: This loop will continue sampling the time using TIME_VALUES and drawing the hands. Note how economical the process is. The four lines from 1120 to 1150 first tell the HAND_DRAW procedure to draw the current hands in black (ie erase them), then draw the minute hand in white and a slightly smaller hour hand in red.

If at any point the program returns from TIME_VALUES with the value of the variable E_FLAG set to 1 rather than 0 (indicating that the user has pressed the space bar), the loop is terminated by line 1110. This use of IF followed by a variable name is a common one which depends on the fact that the IF statement will be carried out if the variable has any other value than zero.

Line 1170: The final act of the program, before terminating, is to return the screen to the lower resolution mode, mode 8.

Testing

The program should now be fully functioning. Run it, enter the time and you should see the clock face displayed with the hands in the correct position.

General comment

One feature of the program can really only be noticed once it has been fully entered. If you examine the program you will notice that it contains not a single GOTO instruction, a fact which may surprise you if you have learned your computing on a less capable machine than the QL. This is quite deliberate. The whole development of the BASIC language over recent years has been in the direction of eliminating the need for GOTO.

While it is possible to carry this practice to absurd lengths (and waste considerable amounts of memory in doing so), there *are* good reasons for trying to reduce reliance on GOTO as a part of your programs. The problem with GOTO is that it is so arbitrary, an instruction to jump which does not, when the program is read later, explain itself. Using GOTO, a program can quickly become a mass of arbitrary jumps which are difficult to plan or explain and, what is worse, encourage you to patch a messy program together with GOTOs when your time would be far better spent redesigning it. The aim of programming on a machine which provides REPEAT loops is to write a program which flows from beginning to end without arbitrary jumps. It is not an aim to become neurotic about — some programmers talk as if a few GOTOs in a program are a sign of brain damage — but you will be surprised how much more satisfying the planning and execution of a program is if you can learn to make it work on the basis of things happening until an EXIT condition is met, rather than jumping about all over the place.

PROGRAM 1.2: CLOCK

Program function

One of the most enjoyable things about computers with graphic displays as good as the QL's, is that they allow you to play about at displaying things in new and imaginative ways. Having just entered a fairly standard clock, this next program gives rather a different view of time. In Clock, hours and minutes are represented by two lines which sweep from left to right and top to bottom, dividing the screen into four rectangles of different colours. Much of the material in Clock is similar to that in Anaclock, so the explanations can be accordingly shortened.

Module 1.2.1: Initialisation

A standard initialisation module. The uses of the variables described will be outlined in the course of the commentary on the program.

Module 1.2.1: Lines 2000 – 2070

```
2000 REMark ************************
2010 DEFine PROCedure initialise
2020 REMark ************************
2030   minute=10 : hour=10
2040   PAPER 0 : INK 7 : CLS : CLS #0
2050   x1=20 : x2=140 : y1=86 : y2=15
2060   e_flag=0
2070 END DEFine initialise
```

Module 1.2.2: Time input

The same module as was included in Anaclock.

Module 1.2.2: Lines 3000 – 3180

```
3000 REMark ************************
3010 DEFine PROCedure set_time
3020 REMark ************************
3030   CLS
3040   AT 1,12 : PRINT "CLOCK SETTING"
3050   REPeat hour
3060     AT 5,1 : INPUT "HOUR (0-23):";h_temp
3070     IF h_temp >=0 AND h_temp <=23 THEN EXIT
         hour
3080   END REPeat hour
3090   REPeat minute
3100     AT 7,1 : INPUT "MINUTE (0-59):";m_temp
3110     IF m_temp>=0 AND m_temp<=59 THEN EXIT
         minute
3120   END REPeat minute
3130   temp$=DATE$
```



```
3140   time=3600*temp$(13 TO 14)+60*temp$(16 TO
       17)+temp$(19 TO 20)
3150   new_time=3600*h_temp+60*m_temp
3160   ADATE new_time-time
3170   CLS
3180 END DEFine set_time
```

Module 1.2.3: Setting up the screen border

This module prints an hour and minute grid along the lefthand side and top of the screen.

Module 1.2.3: Lines 4000 – 4110

```
4000 REMark ************************
4010 DEFine PROCedure draw_framework
4020 REMark ************************
4030   FOR i=1 TO 60
4040     yt=88 : IF i/5=INT(i/5) THEN yt=90
4050     LINE x1+i*2-1,yt TO x1+i*2-1,y1
4060   NEXT i
4070   FOR i=1 TO 12
4080     xt=18 : IF i/3=INT(i/3) THEN xt=16
4090     LINE xt,y1-i*6+1 TO x1,y1-i*6+1
4100   NEXT i
4110 END DEFine draw_framework
```

Commentary

Lines 4030 – 4060: The minute values are spaced out along the top of the screen in white. The loop draws 60 small vertical lines at regular intervals across the screen. These begin at X1 pixels across the screen (20) and Y1 pixels up (86). The X coordinates increase with the loop variable I and every fifth line, when INT(I) will equal I, the line is made slightly longer by adding 2 to the Y coordinate of the top of the line (YT). The effect of this is that the markers for five minutes stand out.

Lines 4070 – 4100: The same principle except that here it is the Y coordinate which increases with the loop variable I, so that the lines move down the screen, representing the hours. Every third hour line is emphasised by drawing it slightly longer.

Testing

Type:

initialise [ENTER]
draw_framework [ENTER]

and you should see the grid drawn on the screen. You will probably note that the tiny minute markers appear not to be perfectly spaced. The reason for this is that in low resolution mode (mode 8), which we are using to

obtain all the colours we want — spacing lines at small intervals leads to some of them falling 'in-between' the actual pixel positions on the screen, so that they are in fact moved one place to the side.

Module 1.2.4: Calculating hours and minutes

This module is parallel to the time calculation module in the last program, though it is slightly simpler because angles do not have to be calculated.

Module 1.2.4: Lines 5000 – 5120

```
5000 REMark ************************
5010 DEFine PROCedure time_values
5020 REMark ************************
5030   minute_temp=minute
5040   REPeat new_minute
5050     LET temp$=DATE$
5060     hour=temp$(13 TO 14)
5070     IF hour>11 THEN hour=hour-12
5080     minute=temp$(16 TO 17)
5090     IF INKEY$=" " THEN e_flag=1 : EXIT
         new_minute
5100     IF minute<>minute_temp THEN EXIT
         new_minute
5110   END REPeat new_minute
5120 END DEFine time_values
```

Commentary

Lines 5030 – 5110: The only differences between this module and Anaclock are that here the value of the last minute acted on is stored at the beginning of the module rather than when the loop is ended, and that the value HOUR is never allowed to exceed 12. Anaclock, because its values describe positions on a circle, can afford to work in 24-hour time — values over 12 simply cause the angle of the hand to 'wrap around' the dial once. Clock, since it works in straight lines, would quickly move off the screen if hours with a value over 12 were generated.

Testing

As for the equivalent module in Anaclock, having initialised the program, call the procedure once to set up the variables and then a second time to ensure that it does wait for the minute to change.

Module 1.2.5: Displaying the time

In this module we get down to the task of displaying on the screen the rectangles which will depict the time. What the module is intended to achieve is the effect of a line sweeping across the screen to depict the

minutes, and another descending, which records the hours. This is accomplished by dividing the screen into four rectangular sections, red, purple, blue and yellow, the edges of these rectangles representing the lines for hours and minutes, as in **Figure 1.2**.

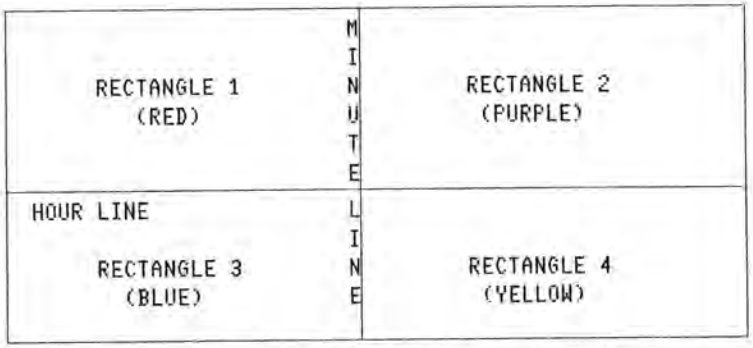


Figure 1.2: The Screen Divided into 4 Rectangular Sections.

Using the variables calculated by the previous module, the LINE command makes it a simple matter to place these rectangles exactly where we want them on the screen.

Module 1.2.5: Lines 6000 – 6250

```
6000 REMark ************************
6010 DEFine PROCedure draw_line(hour,minute)
6020 REMark ************************
6030   hy=y1-hour*6 : IF hour=0 THEN hy=85
6040   mx=x1+minute*2 : IF minute=0 THEN mx=21
6050   INK 0
6060   FILL 1
6070   LINE x1,y1 TO x2,y1 TO x2,y2 TO x1,y2 TO
       x1,y1
6080   FILL 0
6090   FILL 1
6100   INK 2
6110   LINE x1,y1 TO mx-1,y1 TO mx-1,hy+1 TO
       x1,hy+1 TO x1,y1
6120   FILL 0
6130   INK 3
6140   FILL 1
6150   LINE mx,y1 TO x2,y1 TO x2,hy+1 TO mx,hy+1
       TO mx,y1
6160   FILL 0
6170   INK 1
6180   FILL 1
6190   LINE x1,hy TO mx-1,hy TO mx-1,y2 TO x1,y2
```

```
        TO x1,hy
6200    FILL 0
6210    INK 6
6220    FILL 1
6230    LINE mx,hy TO x2,hy TO x2,y2 TO mx,y2 TO
        mx,hy
6240    FILL 0
6250 END DEFine draw_line
```

Commentary

Lines 6030 – 6040: These lines set up the distances across the screen and down which will represent the hours and minutes. The hour line will move down six pixels for every hour which passes, and the minute line across two pixels for every minute.

Lines 6050 – 6080: One of the reasons that this program is slightly simpler than Anaclock is that we have not troubled to remember the positions at which things were drawn for the last minute and hour. The reason for this is that all we need to do in order to erase a previous clock face is to black out the whole rectangle. This is accomplished by drawing a line, in black, around the whole clock face, with FILL set. The function of FILL is to record the coordinates between which lines are drawn and, whenever they make up a closed area, to paint that area with the current INK colour. Drawing a line around the face with FILL set, quickly and effectively erases the previous face. When this has been accomplished it is important to switch off FILL. If this is not done, the QL will become confused as to the coordinates of the shape it is being asked to FILL when further rectangles are drawn — no coordinates are forgotten until FILL is set to zero again.

Lines 6090 – 6120: The first of four sets of lines which draw the colour rectangles which will picture the hours and minutes (see Figure 1.2). These lines draw the upper lefthand rectangle. Lines are drawn from the top lefthand corner, across to the position of the minute line, down to the position of the hour line, back across to the lefthand side of the rectangle and back to the top lefthand corner. Once the rectangle is complete, the FILL command paints it red.

Lines 6130 – 6240: The remaining three rectangles are drawn in the same manner. Note that in order to draw all four rectangles, we only need six variables. X1 and Y1 represent the top lefthand corner of the clock face, X2 and Y2 represent the bottom righthand corner. The position of the minute line is held in MX and that of the hour line in HY. MX,HY is a point inside the clock face where the corners of all the four rectangles meet. In drawing the rectangles, a slight gap (MX – 1 to MX and HY + 1 to HY) is left so that the minute and hour lines will be etched in background black.

Testing

Type:

draw_line 6,30[ENTER]

You should see the four rectangles of roughly equal size displayed on the screen.

Module 1.2.6: Putting it all together

Having entered all the working elements of the program, we can now construct a control module to execute them in the correct order. The use of procedures makes the module completely self-explanatory.

Module 1.2.6: Lines 1000 – 1110

```
1000 REMark ************************
1010 REMark control loop
1020 REMark ************************
1030   initialise
1040   set_time
1050   draw_framework
1060   REPeat control
1070     time_values
1080     IF e_flag THEN EXIT control
1090     draw_line hour,minute
1100   END REPeat control
1110   STOP
```

Testing

You are now in a position to run the full program, input the time and see it displayed.

PROGRAM 1.3: TIMER

Program function

Timer provides you with 12 flexible count-down timers, each of which can be separately programmed to sound an alarm after a specified period and display a short message indicating what the particular occurrence of the alarm is for.

New techniques covered in this program:

1) The 'menu' module.
2) The simple use of BEEP.
3) Adjusting the time with SDATE.

```
                TIMERS
               18:35:02
 1>  07:00:00 WAKE UP!
 2>  08:15:00 LEAVE FOR TRAIN
 3>  00:00:00
 4>  16:45:00 WATCH DANGER-MOUSE
 5>  00:00:00
 6>  00:00:00
 7>  00:00:00
 8>  17:30:00 PHONE CYRIL
 9>  00:00:00
10>  00:00:00
11>  00:00:00
12>  00:00:00
WAITING
```

Figure 1.3: Typical Display Taken from Timer.

Module 1.3.1: Initialisation

The main purpose of this module in the current program is to dimension the two arrays TIMER and TIMER$. TIMER(11) will hold the alarm times of up to 12 separate timers (numbering of arrays begins at zero, remember), while TIMER$(11,20) will hold up to 12 optional messages which may be tagged on to a particular alarm call. Note that in defining the length of the strings in a string array, numbering does *not* begin at zero, so that the maximum message length which TIMER$ can contain is 20 characters, not 21 as the numbering of arrays in other respects might suggest.

Module 1.3.1: Lines 2000 – 2070

```
2000 REMark ************************
2010 DEFine PROCedure initialise
2020 REMark ************************
2030   PAPER 0 : INK 7 : CLS : CLS#0
2040   DIM timer(11), timer$(11,20)
2050   message$=""
2060   sounded=0
2070 END DEFine initialise
```

Module 1.3.2: Formatting the time

Throughout most of this program, the current time will be displayed on the screen. This module uses the QL's flexible time-handling to produce a string containing hours, minutes and seconds, in the format HH:MM:SS, for any given number of seconds.

Module 1.3.2: Lines 6000 – 6050

```
6000 REMark ************************
6010 DEFine PROCedure extract_time (time)
```



```
6020 REMark ************************
6030   temp_time$=DATE$(time)
6040   time$=temp_time$(13 TO)
6050 END DEFine extract_time
```

Commentary

Line 6030: DATE$ has two functions. Used on its own, it produces a string containing the year, month, etc. Used in conjunction with an argument (a figure in brackets following it), it returns the date and time which would represent the number of seconds contained in the argument. Thus, if you enter PRINT DATE$(1), you will see:

1961 Jan 01 00:00:01

or the first second of the period the QL is capable of dealing with. This facility can be used to bypass all kinds of calculations and discover the time represented by a number of seconds.

Line 6040: The time in hours, minutes and seconds is sliced out of the string which was extracted by means of DATE$.

Testing

Enter:

extract_time (1)[ENTER]
print time$

and you should see:

00:00:01

Module 1.3.3: Setting the time

A different module to the one used in the previous two programs. Rather than employing ADATE to adjust the hours, minutes and seconds of DATE$ to the current time, this module makes use of SDATE, which specifies the complete set of figures for DATE$, ie year, month, day, hour, minute, second. The module consists of a series of loops used to ensure sensible figures are input, followed by an SDATE command. Note that the time is actually set when the user presses 'Y' to confirm the new time, so it is wise to enter the time a minute in advance and wait for the precise moment to confirm it.

If you feel it important to update the whole of the date, rather than simply the time, this module can be used to replace the time-setting modules in the two earlier programs.

Module 1.3.3: Lines 10000 – 10340

```
10000 REMark ************************
10010 DEFine PROCedure set_time
10020 REMark ************************
10030   REPeat date_set
10040     CLS
10050     AT 1,12 : PRINT "CLOCK SETTING"
10060     REPeat year
10070       AT 5,1 : INPUT "YEAR (1984-1999):";
            year
10080       IF year>=1984 AND year<=1999 THEN
            EXIT year
10090     END REPeat year
10100     REPeat month
10110       AT 7,1 : INPUT "MONTH (1-12):";month
10120       IF month>=1 AND month<=12 THEN EXIT
            month
10130     END REPeat month
10140     REPeat day
10150       AT 9,1 : INPUT "DAY (1-31):";day
10160       IF day>=0 AND day<=31 THEN EXIT day
10170     END REPeat day
10180     REPeat hour
10190       AT 11,1 : INPUT "HOUR (0-23):";hour
10200       IF hour >=0 AND hour <=23 THEN EXIT
            hour
10210     END REPeat hour
10220     REPeat minute
10230       AT 13,1 : INPUT "MINUTE (0-59):";
            minute
10240       IF minute>=0 AND minute<=59 THEN EXIT
            minute
10250     END REPeat minute
10260     AT 15,1 : INPUT "ARE THESE CORRECT
          (Y/N):";Q$
10270     IF Q$="y" THEN EXIT date_set
10280   END REPeat date_set
10290   SDATE year,month,day,hour,minute,0
10300   AT 17,1 : PRINT "DATE IS NOW: ";DATE$
10310   AT 19,1 : PRINT "press any key to
        continue"
10320   PRINT INKEY$(-1)
10330   CLS
10340 END DEFine set_time
```

Testing

Enter:

set_time[ENTER]

and respond correctly to the various prompts. The module tests itself in that the date printed out at the bottom of the screen represents the time you have set.

Module 1.3.4: Sampling the timers

This small module is an integral part of the program in that it allows the program to enter a waiting state during which the user can make an input, yet at the same time the program is constantly carrying on the work of sampling the 12 timers to see if the alarm should be sounded for any of them. We shall enter the module now, even though it will not be fully used until several later modules have been entered, because it is an essential subroutine for the menu module which follows.

Module 1.3.4: Lines 4000 – 4160

```
4000 REMark ************************
4010 DEFine PROCedure waiting
4020 REMark ************************
4030   REPeat test
4040     FOR count=0 TO 11
4050       IF timer(count)<DATE AND timer(count)
           <>0
4060         current (timer(count))
4070         EXIT test
4080       END IF
4090       extract_time (DATE)
4100       AT 3,14
4110       PRINT time$
4120       t$=INKEY$
4130       IF t$<>"" AND t$<>CHR$(10) THEN EXIT
           test
4140     NEXT count
4150   END REPeat test
4160 END DEFine waiting
```

Commentary

Lines 4030 – 4150: The object of this module is simply to wait — either for a timer to sound or for a key to be pressed. These two exit conditions can be found at lines 4070 and 4130.

Lines 4040 – 4140: This loop shuttles through the 12 timers repeatedly.

Lines 4050 – 4080: These lines are activated if a timer has been set and the time for an alarm to be sounded has arrived. Whether the time has come is easily discernible from the fact that the contents of the timer, held in one element of the array TIMER, are less than the contents of the system variable DATE, which holds the current time in seconds. The procedure CURRENT will be entered later — its purpose is to sound an alarm.

Lines 4090 – 4110: The current time is extracted from DATE$ and printed on the centre of a line towards the top of the screen. Other displays in the program will be built around this.

Lines 4120 – 4130: These lines have the effect of sampling the keyboard and leaving the main loop of the module if a key is pressed. INKEY$ with-

out a parameter attached does not interrupt program flow, it merely catches any key which is being pressed at the moment it is executed. Later on we shall use other forms of INKEY$ to force programs to pause. Note that the WAITING procedure only ends if the key pressed is not CHR$(10) —the ENTER key. The reason for this is that the program relies largely on one key entry for menus and other choices, using INKEY$ rather than INPUT. This is necessary so that other processes, like sampling the timers, can go on while the program is waiting for the user — something that cannot happen with INPUT, which locks up the program until ENTER is pressed. Some people, however, find it difficult to remember that all they have to do is press a single key and press ENTER anyway. The filtering out of the ENTER key here helps to reduce the annoyance of choosing an option from the main program menu and then returning straight to that menu when the unnecessary ENTER is pressed.

Testing

Provided that you have initialised the program, enter:

waiting [ENTER]

and the QL should do just that, displaying the current time towards the top of the screen. The waiting state should carry on until you next press a key other than ENTER.

Module 1.3.5: The program menu

We now come to a new technique which will play a large part in the programs within this book — the 'menu'. In the programs which have led up to this one, the control of the program has been left to the program itself. Once run, a control module has taken over and dictated the flow of program execution until the user signifies that the program is to be terminated. This program, and many of those which follow, is different in that there is no single direction of program flow. The program presents a variety of possibilities to the user and it must be the user who, to a large extent, dictates what happens. This is done by means of a module known as the program menu, which presents the user with a list of the choices which the program provides and allows the user to specify which is to be acted upon. More complex programs later in the book will make use of several menus, each reflecting the variety of choices under one main heading. For the moment, however, we shall stick to the single menu required by this program.

Module 1.3.5: Lines 3000 – 3340

```
3000 REMark ************************
3010 DEFine PROCedure menu
```



```
3020 REMark ************************
3030   REPeat choice
3040     CLS
3050     AT 1,11 : PRINT "TIMER MAIN MENU"
3060     AT 5,1 : PRINT "commands available:"
3070     PRINT\\,"1) DISPLAY TIMERS"
3080     PRINT,"2) SET TIMER"
3090     PRINT,"3) BLANK SCREEN WAIT"
3100     PRINT,"4) STOP"
3110     REPeat response
3120       AT 13,1 : PRINT "WHICH DO YOU
           REQUIRE:";
3130       FLASH 1
3140       PRINT "WAITING"
3150       FLASH 0
3160       waiting
3170       IF sounded=1
3180         sounded=0
3190         choice=0
3200         EXIT response
3210       END IF
3220       choice="0" & t$
3230       IF choice>=1 AND choice<=4
3240         EXIT response
3250       END IF
3260     END REPeat response
3270     SELect ON choice
3280       ON choice=1 : display
3290       ON choice=2 : set_timer
3300       ON choice=3 : blank
3310       ON choice=4 : RETurn
3320     END SELect
3330   END REPeat choice
3340 END DEFine menu
```

Commentary

Lines 3030 and 3330: This loop will continue to display the menu every time execution returns to this module, until the user inputs the number 4. The program will then terminate.

Lines 3060 – 3100: The list of the program options which will become available as subsequent modules are entered.

Lines 3110 – 3260: This loop will repeat the prompt at line 3120 until the user makes a valid input which the program can deal with.

Lines 3120 – 3160: The WAITING procedure that you have already entered is used to accept a one-key input from the user. To help remind the user that only a single keypress is needed, the word 'WAITING' is printed on the screen with FLASH set.

Lines 3170 – 3210: These lines refer to action which will be taken if an alarm has been sounded, as indicated by the value of the variable

SOUNDED. The variable is reset so that it can be used again, the value of CHOICE (the user's input) set to 0 so that the program will do nothing, and the whole menu printed again, since it has been erased by the alarm call.

Lines 3220 – 3250: T$, which is the key pressed by the user during the WAITING routine, has '0' added to the front. The reason for this is that we are about to treat T$ as a number, under the name CHOICE. If the user has inadvertently pressed a non-numeric key, this process could either crash the program or, if the key pressed is the name of a single letter variable, result in a spurious value being stored in CHOICE. Adding zero to the beginning of T$ makes no difference if the input was a number, but results in the return of zero if the key pressed was anything else. The loop is terminated if the CHOICE entered was a valid one, ie in the range 1 – 4.

Lines 3270 – 3340: SELECT is used to allocate work amongst the program according to the selection made by the user.

Testing

If you now enter:

menu [ENTER]

you should see the menu displayed. It should not accept any invalid inputs, but the only one of the options available to you is 'stop', which terminates the program.

Module 1.3.6: Displaying the current timer settings

The purpose of this module is to print out in an orderly way the times for which the 12 timers are currently set. At the moment, since we have not entered the module which allows us to set the timer values, each timer will be displayed as zero.

Module 1.3.6: Lines 11000 – 11160

```
11000 REMark ************************
11010 DEFine PROCedure display
11020 REMark ************************
11030   CLS
11040   AT 1,15
11050   PRINT "TIMERS"
11060   AT 5,0
11070   FOR count=0 TO 11
11080     extract_time (timer(count))
11090     IF count<9 THEN PRINT " ";
11100     PRINT count+1;")"!time$!timer$(count)
11110   NEXT count
11120   FLASH 1
```



```
11130   PRINT "WAITING"
11140   FLASH 0
11150   waiting
11160 END DEFine display
```

Commentary

Lines 11070 – 11110: This loop prints out the contents of the array TIMER and TIMER$. The contents of TIMER, which gives the time for each timer in seconds, is processed by the EXTRACT_TIME module to extract the time in hours, minutes and seconds and it is in this format that it is printed, together with any message stored in TIMER$.

Lines 11120 – 11150: Like the menu, this module relies on WAITING to put it into a waiting state until a key is pressed, following which the program returns to the menu.

Testing

Type:

display [ENTER]

You should see the times for the 12 timers displayed, though the time for each will be '00-00-00' and there will be no messages alongside each one since none have been entered. The flashing 'WAITING' will be displayed at the bottom of the screen and the current time underneath the program title.

Module 1.3.7: Calculating timer settings

Now that we can display the timers we shall enter the two modules which allow the user to set them. The current module is primarily designed to accept a time in hours and minutes which will be used by the next module to set a timer.

Module 1.3.7: Lines 7000 – 7210

```
7000 REMark ************************
7010 DEFine PROCedure create_time
7020 REMark ************************
7030   REPeat countdown
7040     INPUT "COUNTDOWN (y/n):";down$
7050     IF down$="y" OR down$="Y" OR down$="n"
         OR down$="N" THEN EXIT countdown
7060   END REPeat countdown
7070   REPeat hour
7080     AT 10,0
7090     INPUT "HOUR:";hour
7100     IF down$="n" OR down$="N"
7110       IF hour>=0 AND hour<=47 THEN EXIT hour
```

```
7120      END IF
7130      IF down$="y" OR down$="Y" THEN EXIT hour
7140    END REPeat hour
7150    REPeat minute
7160      AT 12,0
7170      INPUT "MINUTE:";minute
7180      IF minute>=0 AND minute<=59 THEN EXIT
          minute
7190    END REPeat minute
7200    time=hour*3600+minute*60
7210 END DEFine create_time
```

Commentary

Lines 7030 – 7060: Setting of timers can be done in two different ways. Either a time can be input at which the timer will go off (eg input 12.00 and the timer goes off at twelve o'clock) *or* a period can be specified after which the timer will sound (eg input 1 hour and the timer will sound after one hour). These lines ask the user to specify whether the time to be input is to be used for a count-down or simply as a time of day.

Lines 7070 – 7140: If what is being input is a time of day, then the hour figure can only be in the range zero to 47. This may sound like a rather strange range to impose, but it is a simple way of allowing the user to specify whether the alarm is to go off today or tomorrow. If the alarm is intended for 12.00 and the time is already 19.00, there is no point in entering 12 hours, no minutes as the time, since the alarm would go off immediately. For dates on the following day, simply add 24 hours to the time expressed in terms of the 24-hour clock, so that 12.00 the following day would be entered as 36. If the figure is to be the basis for a count-down timer, no limit is imposed.

Line 7200: The time input is translated into seconds.

Testing

Type:

create_time [ENTER]

In answer to the prompt, specify that you do not want a count-down timer and you should then be prompted for the hours and minutes. If you input 10 hours and 11 minutes, the resulting value of the variable TIME should be 36660, and you can print out TIME to check this.

Module 1.3.8: Setting the timers

Now that we can display the state of the timers and translate an input into a time value expressed in seconds, we can proceed to the module which allows the user to specify the settings for the 12 timers. Note that this

module uses INPUT to obtain the three items of data it requires, so while the module is operating the timers themselves are not being sampled and any timer due to give an alarm will not sound until this module has been completed.

Module 1.3.8: Lines 8000 – 8310

```
8000 REMark ************************
8010 DEFine PROCedure set_timer
8020 REMark ************************
8030   CLS
8040   AT 1,13
8050   PRINT "SET TIMER"
8060   REPeat response
8070     AT 5,0
8080     INPUT "which timer (1-12):";tnum
8090     IF tnum>=1 AND tnum<=12 THEN EXIT
         response
8100   END REPeat response
8110   tnum=tnum-1
8120   AT 8,0
8130   create_time
8140   IF down$="Y" OR down$="y"
8150     time=DATE+time
8160   ELSE
8170     days=3600*24*(INT(DATE/(3600*24)))
8180     time=days+time
8190   END IF
8200   AT 14,0
8210   INPUT "message:";message$
8220   REPeat response
8230     AT 16,0
8240     INPUT "Is this correct (Y/N):";Q$
8250     IF Q$="Y" OR Q$="y" OR Q$="N" OR Q$="n"
         THEN EXIT response
8260   END REPeat response
8270   IF Q$="Y" OR Q$="y"
8280     timer(tnum)=time
8290     timer$(tnum)=message$
8300   END IF
8310 END DEFine set_timer
```

Commentary

Lines 8060 – 8110: The user is invited to input a timer number in the range 1 – 12. Note that since the array runs from zero to 11, this value has to be reduced by one.

Lines 8140 – 8190: If the user has specified a count-down timer, all that needs to be done is to add the time input by the user (stored in seconds in the variable TIME) to the contents of the system variable DATE (the current time and date expressed in seconds). If the time input is intended simply as a straight time, these lines take the value of DATE and slice off the part

which refers to hours and minutes, so that the number of seconds up to the start of the current day is stored in DAYS. To this is then added the number of seconds which make up the time input by the user. Thus what is eventually stored is not simply a figure representing hours and minutes but a record of the total number of seconds since January 1st 1961 up to the specified time on the current day or the day following.

Lines 8210 – 8300: A short message (up to 20 characters) can be attached to any timer when it is set. These lines accept the message, confirm all the details and then store time and message in the arrays TIMER and TIMER$. The place in the array is dictated by the user's choice, recorded in the variable TNUM.

Testing

Provided that you have previously initialised and set the time, you should at this stage be able to test the module by typing:

menu [ENTER]

When you have the menu up, specify option 2 and input:

1, Y, 0, 1 *and* TEST

in response to the five prompts — ie timer 1 to be set, (Y)es to a count-down timer, period no hours and one minute, with the message 'TEST' attached. The program should now return to the menu. Now specify option 1 to display the timers and you should find that timer 0 has been given a value somewhere less than a minute ahead of the current time (how far ahead will depend on how quick you have been) plus the label 'TEST'. Don't bother waiting for the alarm to sound as we have not yet entered the module to achieve that.

Module 1.3.9: Sounding the alarm

We now have all the working elements of the program with the exception that the timers cannot yet announce themselves with an alarm call. This module and the next add that final touch, with the current module producing the sound and the next one connecting the alarm to the rest of the system.

Module 1.3.9: Lines 5000 – 5270

```
5000 REMark ************************
5010 DEFine PROCedure current (time)
5020 REMark ************************
5030   CLS
5040   now=DATE
5050   extract_time (time)
```



```
5060   FLASH 1
5070   AT 5,14 : PRINT time$
5080   FLASH 0
5090   AT 8,18-LEN(timer$(count))/2
5100   PRINT timer$(count)
5110   AT 19,1
5120   PRINT "press any key to continue"
5130   REPeat sound
5140     BEEP 0,10
5150     FOR delay=1 TO 500
5160     NEXT delay
5170     BEEP 0,20
5180     FOR delay=1 TO 500
5190     NEXT delay
5200     BEEP
5210     t$=INKEY$
5220     IF DATE>now+60 THEN EXIT sound
5230     IF t$<>"" THEN EXIT sound
5240   END REPeat sound
5250   timer(count)=0
5260   timer$(count)=""
5265   sounded=1
5270 END DEFine current
```

Commentary

Line 5040: The current time is stored in the variable NOW. This will be used later in the module to determine whether the alarm has been sounded for one minute.

Lines 5050 – 5080: This module is called by one you have already entered, WAITING. That module has already extracted a time from one of the timers and sent it to this module in the form of the parameter TIME. These lines calls up EXTRACT_TIME to translate the time into a string. The result is placed on the screen in flashing letters.

Lines 5090 – 5100: If there is a message in the corresponding line of the array TIMER$, it is printed in the centre of the screen.

Lines 5130 – 5240: This loop sounds two notes, rather like a film version of the siren on a French police car, for a period of 60 seconds. The lines from 5140 to 5200 sound two notes with a short delay between them, then turn the sound off. Tests are then made to see whether a key is being pressed or whether it is more than 60 seconds since the variable NOW was set. If either of these tests is positive, the loop which sounds the alarm ends.

Lines 5250 – 5270: Having finished with the alarm, the value of the timer is set to 0 and any associated message cleared. The variable SOUNDED is set to 1 as an indication to the menu module that the screen has been cleared while it was waiting for an input.

Testing

Type:

menu [ENTER]

and specify a count-down timer with a period of a minute. The result should be that at the end of the minute the alarm should sound for a period of approximately one more minute. If you wish, you can set another timer and this time cut off the alarm by pressing a key — the alarm may not stop instantly due to the fact that the notes and the timing loop have to finish. Finally, you might like to check that the alarm still sounds when you have the 12 timers displayed on the screen using menu option 1.

Module 1.3.10: Blanking the screen

The whole purpose of this program is that it should be left running in the background so that it can act as a timer system. Leaving the QL switched on to achieve this will do it no harm at all, but leaving the television set switched on and displaying the same screen for long periods of time may eventually cause some of the brighter parts of what is displayed to become slightly etched upon the screen so that they appear as permanent ghosts over whatever the screen is showing. (Don't panic about this, it's not something that's going to happen just because you happen to leave the QL and TV switched on for a few hours by mistake — we are talking about continual use of the same screen design over a long period).

To overcome this possible problem, the program has built into it a screen protection module which blanks the screen and then displays a short message in flashing letters. Since flashing letters are not permanently displayed on the screen, they will not cause etching on the screen in the same way as normal lettering.

While the screen is blanked, the timers are constantly being sampled by means of the previous module. Any alarms which arise will be sounded in the normal way.

Module 1.3.10: Lines 9000 – 9120

```
9000 REMark ************************
9010 DEFine PROCedure blank
9020 REMark ************************
9030   REPeat blank_loop
9040     CLS
9050     FLASH 1
9060     AT 9,15
9070     PRINT "TIMER"
9080     waiting
9090     FLASH 0
9100     IF t$<>"" THEN EXIT blank_loop
9110   END REPeat blank_loop
9120 END DEFine blank
```


Testing

Set up one or two timers for short periods and then call up the blank screen state from the menu. Though you cannot see the state of the timers until they expire, you should find that alarms are sounded in the normal way.

Module 1.3.11: The control module

Finally, the control module, which ties together what you have entered.

Module 1.3.11: Lines 1000 – 1070

```
1000 REMark ************************
1010 REMark control loop
1020 REMark ************************
1030   initialise
1040   set_time
1050   menu
1060   CLS
1070   STOP
```

Testing

All the functions which you have already entered should now be available when you run the program.

PROGRAM 1.4: EVENT

Program function

The final program in this chapter of experiments with time is fairly unusual in that it seeks to turn the QL into a stopwatch. Of course the accuracy of a microcomputer is not to be compared with that of specialist watches, but the computer does have some advantages in that it can keep a record of the times it has produced, perform calculations on them, and so on. The current program is designed to time one of a series of events, to display the series of times, with or without an identifying message attached and, for each event, to calculate the period since the previous one. It can also, on request, print out the list of times so that a hard copy can be kept.

New topics introduced in the course of this chapter:

1) The use of the SER1 port for printed output from programs.

Module 1.4.1: Initialisation

A standard initialisation module. The variables will be discussed during the course of the commentary on the program.

```
18:47:46  (00:00:01)
18:47:47  (00:00:01)
18:47:49  (00:00:02)  LORRY
18:47:53  (00:00:04)
18:47:54  (00:00:01)
18:47:56  (00:00:02)
18:47:59  (00:00:03)  BUS
18:48:06  (00:00:07)
18:48:07  (00:00:01)
18:48:11  (00:00:04)  LORRY
18:48:14  (00:00:03)
18:48:15  (00:00:01)  LORRY
18:48:21  (00:00:06)
18:48:22  (00:00:01)
18:48:24  (00:00:02)
18:48:24  (00:00:00)
18:48:24  (00:00:00)
18:48:26  (00:00:02)  TANK
```

Figure 1.4: Output of Event being used to Time Traffic, with Heavy Vehicles Marked.

Module 1.4.1: Lines 2000 – 2060

```
2000 REMark ************************
2010 DEFine PROCedure initialise
2020 REMark ************************
2030   PAPER 0 : INK 7 : CLS : CLS#0
2040   time1=DATE
2050   printer=0
2060 END DEFine initialise
```

Module 1.4.2: Setting the time

The last in the series of time-setting modules. This one is the longest but, paradoxically, the easiest to use. To the module in the last program has been added the option to choose whether or not to reset the existing time. Once the time has been set, a simple 'N' saves all the trouble of entering everything again.

Module 1.4.2: Lines 3000 – 3370

```
3000 REMark ************************
3010 DEFine PROCedure set_time
3020 REMark ************************
3030   REPeat date_set
3040     CLS
3050     AT 1,1 : PRINT "DATE IS NOW: ";DATE$
3060     AT 3,1 : INPUT "Do you wish to reset
         date (Y/N):";Q$
3070     IF Q$<>"y" AND Q$<>"Y" THEN RETurn
```



```
3080     CLS
3090     AT 1,12 : PRINT "CLOCK SETTING"
3100     REPeat year
3110       AT 5,1 : INPUT "YEAR (1984-1999):";year
3120       IF year>=1984 AND year<=1999 THEN EXIT
           year
3130     END REPeat year
3140     REPeat month
3150       AT 7,1 : INPUT "MONTH (1-12):";month
3160       IF month>=1 AND month<=12 THEN EXIT
           month
3170     END REPeat month
3180     REPeat day
3190       AT 9,1 : INPUT "DAY (1-31):";day
3200       IF day>=0 AND day<=31 THEN EXIT day
3210     END REPeat day
3220     REPeat hour
3230       AT 11,1 : INPUT "HOUR (0-23):";hour
3240       IF hour >=0 AND hour <=23 THEN EXIT
           hour
3250     END REPeat hour
3260     REPeat minute
3270       AT 13,1 : INPUT "MINUTE (0-59):";minute
3280       IF minute>=0 AND minute<=59 THEN EXIT
           minute
3290     END REPeat minute
3300     AT 15,1 : INPUT "ARE THESE CORRECT
         (Y/N):";Q$
3310     IF Q$="y" THEN EXIT date_set
3320   END REPeat date_set
3330   SDATE year,month,day,hour,minute,0
3340   AT 17,1 : PRINT "DATE IS NOW: ";DATE$
3350   AT 19,1 : PRINT "press any key to continue"
3360   PRINT INKEY$(-1)
3370 END DEFine set_time
```

Module 1.4.3: Waiting for input

This module is equivalent to the WAIT procedure in the previous program in that it creates a waiting state. In addition, it is capable of recognising if the user is asking for the printer to be connected or for the program to be terminated.

Module 1.4.3: Lines 4000 – 4200

```
4000 REMark ************************
4010 DEFine PROCedure wait
4020 REMark ************************
4030   message$=""
4040   REPeat waiting
4050     t$=INKEY$
4060     IF t$<>""
4070       time_values
4080       IF t$=CHR$(10)
```

```
4090          INPUT#0, "MESSAGE (<=15 CHARS):";
              message$
4100          IF LEN(message$)>15 THEN message$=
              message$(1 TO 15)
4110            CLS #0
4120        END IF
4130        IF t$="p" OR t$="P"
4140          printer=ABS(printer-1)
4150        ELSE
4160          EXIT waiting
4170        END IF
4180      END IF
4190    END REPeat waiting
4200 END DEFine wait
```

Commentary

Line 4030: This string will be used to store any message associated with a particular event timing.

Lines 4070 – 4110: If the key pressed is ENTER, then the user is prompted on the command lines at the bottom of the screen to input a short message to accompany the event time, the command lines then being cleared again.

Lines 4130 – 4140: If the key pressed is the letter P, then the value of the variable PRINTER is toggled between zero and one. It's worth taking note of this simple routine as it represents the classic way of shuttling a variable between two values. If you have a variable X and you want to shuttle it backwards and forwards between values V1 and V2 (where V1 is the lower), the format is:

X = V1 + V2 – X

but, since our lower value is zero, it is simplified to:

X = V2 – X

Lines 4150 – 4170: To arrive at this point, the key pressed must have been anything but the letter P. The result is that the time values are updated by the next module and the waiting state terminated.

Testing

Ensure that the program is initialised, then type:

wait [ENTER]

and the QL should go into the waiting state. If you press the P key, nothing should visibly happen. If you press [ENTER], you should be prompted for a message and the wait terminated. Any other key simply terminates the procedure.

Module 1.4.4: Extracting time values

This module obtains all the information necessary for printing out the current time and the period between events.

Module 1.4.4: Lines 5000 – 5090

```
5000 REMark ************************
5010 DEFine PROCedure time_values
5020 REMark ************************
5030   time2=time1
5040   time1=DATE
5050   now$=DATE$
5060   period=time1-time2
5070   time$=DATE$(period)
5080   time$=time$(13 TO)
5090 END DEFine time_values
```

Commentary

Lines 5030 – 5040: The last time recorded is shifted over to the variable TIME2 before the current time is taken into TIME1.

Line 5050: The string NOW$ will be used for printing out the current time.

Lines 5060 – 5070: The difference between TIME1 and TIME2 is obtained and translated into a time format by use of DATE$.

Testing

Provided that you have initialised the program, type:

time_values [ENTER]
print time$,now$[ENTER]

You should find that NOW$ roughly represents the current time, while TIME$ will represent the period since the program was last initialised and TIME1 set equal to DATE.

Module 1.4.5: Printing the results

This module prints out the results of the previous calculations, either on the screen alone, or to the screen and printer if this has been specified.

Module 1.4.5: Lines 6000 – 6090

```
6000 REMark ************************
6010 DEFine PROCedure print_values
6020 REMark ************************
6030   PRINT now$(13 TO 20)!!"(";time$;")"!!
       message$
6040   IF printer=1
6050     OPEN#9,ser1
```

```
6060      PRINT#9,now$(13 TO 20)!!"(";time$;")"!!
          message$
6070      CLOSE#9
6080    END IF
6090 END DEFine print_values
```

Commentary

Line 6030: The time at which the key was pressed, the period since the last event and any associated message are printed on the same line.

Lines 6040 – 6080: These lines open a channel of communication with the port on the back of the QL labelled SER1. This port is designed so that the QL can communicate with a printer which operates according to what is known as the RS232C protocol — and make sure your printer *does* work with the QL before you buy it! Once the channel has been opened, items to be printed which are sent along that channel by including the channel number after the PRINT command will end up being output to the printer rather than to the screen. In the case of this module the channel is opened and closed for each item to ensure that it is properly closed when the program terminates. It could equally well be opened at the beginning of the program and closed at the end provided that no other use is being made of the SER1 port. Note that if you don't have a printer connected to the SER1 port and you call up the printer, the program will lock up.

Testing

If you performed the test on the previous module, just type:

print_values [ENTER]

and you should see what you previously printed out as NOW$ and TIME$, neatly formatted. If the variable PRINTER was equal to 1, and a printer connected, the output will also go to the printer.

Module 1.4.6: The control module

The final touch, as always, is the control module, which ties the whole program together.

Module 1.4.6: Lines 1000 – 1120

```
1000 REMark ************************
1010 REMark control loop
1020 REMark ************************
1030   initialise
1040   set_time
1050   CLS
1060   REPeat control
1070     wait
```



```
1080     IF t$=CHR$(27) THEN EXIT control
1090     print_values
1100   END REPeat control
1110   PRINT #0,"Program terminated"
1120   STOP
```

Commentary

Lines 1080 and 1110: One extra touch which isn't apparent from the rest of the program is that, if you pressed the ESCape key while in the waiting loop, the program will stop, printing the message 'Program terminated' on the command lines at the bottom of the screen.

Conclusion

There are a number of lessons to be learned from the programs in this chapter, not least that the uses of your QL are only limited by your imagination. But perhaps the most important lesson you can learn, if you look back over the programs, is just how easy the design of programs becomes when everything is contained in neat modules which can be transferred from one program to the next. With its easy-to-use MERGE command and the ability to save parts of programs, the QL cries out to be used in this way. Once you have built up a sufficient library of useful routines you will find that much of your programming becomes like fitting together a jigsaw puzzle, except that the completed puzzle will, at least sometimes, be more useful.

CHAPTER 2

Son et Lumière

In this chapter we turn to a series of programs which will allow you to put some of the QL's graphical abilities to use and, in addition, to make the most of the rather confusing sound commands.

The programs included in this chapter are:

DESIGNER: A tool which allows line drawings far larger than a single screen to be constructed, manipulated and displayed.

3-D GRAPH: Using turtle graphics to produce a clear and attractive display for complex figures.

SCREEN: Copies the contents of the screen to a printer.

CHARACTERS: Designs and stores your own customised character sets.

SOUND DEMO: A simple routine to permit experiments with the sound parameters.

MUSIC: Allows complex tunes to be input in a comprehensible form and played.

PROGRAM 2.1: DESIGNER

Program function

No doubt we have all seen the impressive displays created by what is known as CAD (computer aided design) software. With deft touches, the engineer adds lines and shapes to a complex design, or erases those which already exist. In a limited kind of way, Designer is intended to mimic that kind of capability. While clearly not as sophisticated, it will allow you to create complex designs which are far larger than a single screen, using the television screen as a moving window to examine parts or shrinking the whole area down so that it can be viewed in its entirety. Lines, circles and boxes can be added at will or deleted, and the whole design stored on microdrive for later use.

New concepts introduced in this program include:

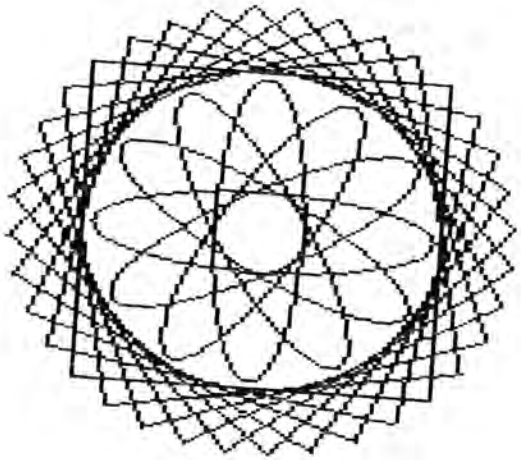


Figure 2.1: Screen Dump from Designer.

1) Graphics commands LINE, CIRCLE and SCALE.
2) Starting a program without clearing variables.
3) A user-defined flashing cursor in hi-res.
4) Storing and retrieving data on microdrives.

Module 2.1.1: Initialisation

A standard initialisation module.

Module 2.1.1: Lines 2000 – 2090

```
2000 REMark ************************
2010 DEFine PROCedure initialise
2020 REMark ************************
2030   INK 7
2040   DIM a(1000,8)
2050   size=2 : screen=100
2060   cx=83 : cy=50
2070   item=0
2080   SCALE 100,0,0
2090 END DEFine initialise
```

Commentary

Line 2040: The array A will be used to store details of the lines and shapes to be drawn.

Line 2050: The variable SIZE will record the size of the flashing cursor. This will vary with the scale at which the design is being viewed.

Line 2060: The original coordinates of the cursor (cursor X and cursor Y).

Line 2070: The number of lines or shapes contained within the drawing.

Line 2080: Later program sections will allow the design to be viewed at different degrees of enlargement and in different sections. To begin with we use the normal scaling of the machine, with 100 units on the horizontal axis and the picture starting at position 0,0.

Module 2.1.2: A flashing cursor

In any design program, one of the first necessities is that the user should know where the current drawing position is. This is usually achieved by means of a cursor of some kind, marking the current position on the screen. The cursor for this program consists of a small flashing cross which can be moved over the design using the cursor control keys, without affecting the contents of the screen.

Module 2.1.2: Lines 3000 – 3150

```
3000 REMark ************************
3010 DEFine PROCedure cross
3020 REMark ************************
3030   OVER -1
3040   REPeat draw
3050     LINE cx-size,cy TO cx+size,cy
3060     LINE cx,cy-size TO cx,cy+size
3070     FOR delay=1 TO 10 : NEXT delay
3080     LINE cx-size,cy TO cx+size,cy
3090     LINE cx,cy-size TO cx,cy+size
3100     FOR delay=1 TO 10 : NEXT delay
3110     t$=INKEY$
3120     IF t$<>"" THEN EXIT draw
3130   END REPeat draw
3140   OVER 0
3150 END DEFine cross
```

Commentary

Lines 3040 – 3130: These lines draw two short lines, to a length of SIZE, at right angles, crossing at the current drawing position. The cursor is drawn with OVER set in such a way that it reverses any pixels over which it passes. Drawing the cursor twice erases it and reinstates the screen to the position which existed before it was drawn. The two DELAY loops ensure that the timing is such that the cursor appears to be flashing rather than merely flickering.

Lines 3110 – 3120: These lines ensure that the cursor will continue to flash until a key is pressed. Dealing with the input is the job of the next module.

Testing

Type:

initialise[ENTER]
cross[ENTER]

The screen should clear and you should see the small flashing cursor appear in the centre. Pressing any key should make the cross disappear and terminate the module.

Module 2.1.3: Input of commands

Having given ourselves a flashing cursor and the ability to detect the user pressing a key, the current module allocates work among the various parts of the program when a key is pressed. Its correct functioning can only be tested when the subsequent module, the short control module, is added. Most of the single commands mentioned in the commentary will not, of course, become operative until later modules are added.

Module 2.1.3: Lines 4000 – 4270

```
4000 REMark ************************
4010 DEFine PROCedure analyse_key
4020 REMark ************************
4030   IF t$=CHR$(192) THEN cx=cx-1
4040   IF t$=CHR$(196) THEN cx=cx-10*screen/100
4050   IF t$=CHR$(200) THEN cx=cx+1
4060   IF t$=CHR$(204) THEN cx=cx+10*screen/100
4070   IF t$=CHR$(208) THEN cy=cy+1
4080   IF t$=CHR$(212) THEN cy=cy+10*screen/100
4090   IF t$=CHR$(216) THEN cy=cy-1
4100   IF t$=CHR$(220) THEN cy=cy-10*screen/100
4110   IF t$="1"
4120     x1=cx
4130     y1=cy
4140   END IF
4150   IF t$="!"
4160     cx=x1
4170     cy=y1
4180   END IF
4190   IF t$="l" OR t$="L" THEN line_draw
4200   IF t$="c" OR t$="C" THEN circle_draw
4210   IF t$="b" OR t$="B" THEN box
4220   IF t$="d" OR t$="D" THEN remove
4230   IF t$="s" OR t$="S" THEN scaling
4240   IF t$="m" OR t$="M" THEN store
4250   IF t$=CHR$(27) THEN STOP
4260   CLS #0
4270 END DEFine analyse_key
```


Commentary

Lines 4030 – 4100: The character codes given here are those of the cursor keys. Pressing one of the cursor arrows alters the value of CX or CY, thus moving the cursor. Using the shifted cursor keys moves the cursor 10 screen positions, regardless of the scale at which the design is being viewed.

Lines 4110 – 4140: Input of '1' defines the end of a line or shape to be drawn.

Lines 4150 – 4180: Input of '!' moves the cursor immediately to the position defined by pressing 1.

Line 4190: Input of 'L' draws a line between the points defined by the pressing of 1 and the current position of the cursor.

Line 4200: Input of 'C' results in the drawing of a circle whose circumference will touch points 1 and the current cursor position and whose centre will lie halfway between them.

Line 4210: Input of 'B' draws a box whose upper lefthand corner is defined by 1 and bottom righthand corner by the current cursor position.

Line 4220: Input of 'D' puts the program into the mode where individual lines can be deleted.

Line 4230: Input of 'S' allows the user to change the scale or position from which the design is viewed.

Line 4240: Input of 'M' stores the design on microdrive.

Line 4250: Pressing ESCape stops the program.

Module 2.1.4: The control module

The short control module needs to be entered at this point so that the cursor move function can be tested and later modules tested as they are entered. Note that the module makes provision for the recall of data from microdrives — this will be discussed in full when the relevant modules are entered. In addition, provision is made for the user to specify whether the program is to clear its variables or not.

Module 2.1.4: Lines 1000 – 1140

```
1000 REMark ************************
1010 DEFine PROCedure  do (fresh)
1020 REMark ************************
1030   CLS : CLS #0 : OVER 0
1040   IF fresh THEN initialise
1050   CLS #0
1060   INPUT #0,"Load from Microdrive (y/n): ";q$
```

```
1070    IF q$="y" OR q$="Y"
1080      recall
1090    END IF
1100    REPeat x
1110      cross
1120      analyse_key
1130    END REPeat x
1140 END DEFine do
```

Commentary

Lines 1010 and 1040: The argument attached to DO is used to determine whether the initialisation module will be called up. If the command to start the program is DO 1, the variables will be cleared, while if it is DO 0, they will be left intact and any existing design may be retrieved.

Testing

Start the program with DO 1 and you should find that you are able to move the flashing cursor around by means of the cursor control keys. None of the other program functions will yet be available.

Module 2.1.5: Drawing a line

The simplest kind of addition which can be made to the design is the drawing of a line between points 1 and 2. Like all the drawing modules, this one draws the line with OVER set, then gives the user the opportunity to confirm the addition before it is finalised. In a crowded design, adding the line with OVER set will erase any inked pixels over which the line passes. This is rectified when the line is confirmed or when it is erased.

Module 2.1.5: Lines 5000 – 5130

```
5000 REMark ************************
5010 DEFine PROCedure line_draw
5020 REMark ************************
5030   OVER -1
5040   LINE x1,y1 TO cx,cy
5050   INPUT #0,"CONFIRM (Y/N):";q$
5060   IF q$="Y" OR q$="y"
5070     OVER 0
5080     type=1
5090     record
5100   END IF
5110   LINE x1,y1 TO cx,cy
5120   OVER 0
5130 END DEFine line_draw
```

Testing

Enter the following lines which will form part of a later module:

```
8000 DEF PROCrecord
8350 END DEFrecord
```


Now run the program and immediately press 1 to define one end of a line. Now move the cursor and press 'L' (or 'l'). You should see a line drawn between the two points and be asked to confirm it. If you do confirm it, the flashing cursor will return. If you do not confirm it, the line should be erased before the cursor returns.

Module 2.1.6: Drawing a circle

This module is, in principle, no different from the last, with a shape being drawn and confirmed by the user. Apart from the two defined points, however, one further piece of information is required, namely the rotation of the circle to be drawn — since it may not be a pure circle but an ellipse of some kind.

Module 2.1.6: Lines 6000 – 6180

```
6000 REMark ************************
6010 DEFine PROCedure circle_draw
6020 REMark ************************
6030   OVER -1
6040   ox=(x1+cx)/2
6050   oy=(y1+cy)/2
6060   ra=ABS(cy-y1)/2
6070   e=ABS(cx-x1)/(ABS(cy-y1)+1)
6080   INPUT #0,"ROTATION (0-360): ";ro
6090   CIRCLE ox,oy,ra,e,RAD(ro)
6100   INPUT #0,"CONFIRM (Y/N):";q$
6110   IF q$="Y" OR q$="y"
6120     OVER 0
6130     type=2
6140     record
6150   END IF
6160   CIRCLE ox,oy,ra,e,RAD(ro)
6170   OVER 0
6180 END DEFine circle_draw
```

Commentary

Lines 6040 – 6050: The origin of the circle is discovered by adding the X and Y coordinates of points 1 and 2 and dividing by two, thus finding a point halfway in between.

Lines 6060 – 6070: The QL works, for the radius of its CIRCLE command, on the Y axis, so half the distance between the two Y coordinates is calculated. Any difference between the distance between the Y coordinates and the X coordinates will be used to supply the 'eccentricity' of the circle, the degree to which it will appear stretched or squashed horizontally.

Line 6080: Having defined the shape, the circle command also requires a figure for the rotation of that shape. This is input in degrees and translated into radians using RAD.

Testing

You should now be able to call up the circle command with 'C' while the program is running.

Module 2.1.7: Drawing a box

The final shape which the program is capable of drawing is a simple rectangle, with its opposite corners at points 1 and 2. This module is slightly more complex than the previous ones since there is no built-in command for the shape and, in addition, I have included provision for it to be rotated.

Module 2.1.7: Lines 7000 – 7260

```
7000 REMark ************************
7010 DEFine PROCedure box
7020 REMark ************************
7030   OVER -1
7040   ox=(x1+cx)/2
7050   oy=(y1+cy)/2
7060   INPUT #0,"ROTATION (0-360): ";ro
7070   ro=RAD(ro)
7080   rx1=ox+(x1-ox)*COS(ro)+(y1-oy)*SIN(ro)
7090   ry1=oy+(y1-oy)*COS(ro)-(x1-ox)*SIN(ro)
7100   rx2=ox+(cx-ox)*COS(ro)+(y1-oy)*SIN(ro)
7110   ry2=oy+(y1-oy)*COS(ro)-(cx-ox)*SIN(ro)
7120   rx3=ox+(cx-ox)*COS(ro)+(cy-oy)*SIN(ro)
7130   ry3=oy+(cy-oy)*COS(ro)-(cx-ox)*SIN(ro)
7140   rx4=ox+(x1-ox)*COS(ro)+(cy-oy)*SIN(ro)
7150   ry4=oy+(cy-oy)*COS(ro)-(x1-ox)*SIN(ro)
7160   LINE rx1,ry1 TO rx2,ry2 TO rx3,ry3 TO
       rx4,ry4 TO rx1,ry1
7170   INPUT #0,"CONFIRM (Y/N):";q$
7180   IF q$="Y" OR q$="y"
7190     OVER 0
7200     type=3
7210     record
7220   END IF
7230   LINE rx1,ry1 TO rx2,ry2 TO rx3,ry3 TO
       rx4,ry4 TO rx1,ry1
7240   OVER 0
7250   CLS #0
7260 END DEFine box
```

Commentary

Lines 7040 – 7050: Because we are going to supply the option of rotating the rectangle, its centre has to be found in the same way as for the circle.

Lines 7080 – 7150: The formulae for rotating one point around another are as follows:

X2 = XO + XD * COS ANGLE + YD * SIN ANGLE
Y2 = YO + YD * COS ANGLE – XD * SIN ANGLE

X2 AND Y2 are the X and Y coordinates which result from rotating the point X,Y.
XO and YO are the coordinates around which the point is being rotated.
XD and YD are the distances of X and Y from XO and YO respectively.
ANGLE is the angle through which the point defined by X and Y is being rotated.

The basic rectangle, unrotated, will have corners of X1/Y1, X2/Y1, X2/Y2 and X1/Y2. Looking at the lines and comparing them with the formulae above, you should be able to see that they provide the rotated coordinates for the four corner points.

Testing

You should now be able to press B, define a rectangle and rotate it at will, in the same way as a circle.

Module 2.1.8: Recording a design

A program like this one is little use for anything but some passing fun unless the design being worked on can be recorded in some way. In our case, recording the design in an array will later allow us both to store the data on microdrive and to delete individual lines. Lines and shapes are all recorded by calling this module as they are entered, which is why we earlier had to enter the two lines which begin and end this module before tests could be made.

Module 2.1.8: Lines 8000 – 8350

```
8000 REMark ************************
8010 DEFine PROCedure record
8020 REMark ************************
8030   IF item=1001
8040     PRINT #0, "NO ROOM FOR MORE LINES"
8050     t$=INKEY$(-1)
8060     RETurn
8070   END IF
8080   IF type=1
8090     a(item,0)=type
8100     a(item,1)=x1
8110     a(item,2)=y1
8120     a(item,3)=cx
8130     a(item,4)=cy
8140   END IF
8150   IF type=2
8160     a(item,0)=type
8170     a(item,1)=ox
8180     a(item,2)=oy
8190     a(item,3)=ra
8200     a(item,4)=e
```

```
8210      a(item,5)=ro
8220    END IF
8230    IF type=3
8240      a(item,0)=type
8250      a(item,1)=rx1
8260      a(item,2)=ry1
8270      a(item,3)=rx2
8280      a(item,4)=ry2
8290      a(item,5)=rx3
8300      a(item,6)=ry3
8310      a(item,7)=rx4
8320      a(item,8)=ry4
8330    END IF
8340    item=item+1
8350 END DEFine record
```

Commentary

Lines 8080 – 8140: If the new item is a simple line, all that needs to be done is to record its type (1) and the start and finish coordinates.

Lines 8150 – 8220: In the case of a circle, the coordinates of the centre, the radius, the eccentricity and the rotation are recorded.

Lines 8230 – 8330: For a box, the X and Y coordinates of all the four corners needs to be recorded if the rotation is not to be recalculated when the shape is redrawn.

Module 2.1.9: Redrawing a shape

Having given ourselves the ability to record a shape in an array, we now turn to the question of retrieving a shape from an array and placing it back on the screen.

Module 2.1.9: Lines 10000 – 10120

```
10000 REMark ************************
10010 DEFine PROCedure global_draw (i)
10020 REMark ************************
10030   IF a(i,0)=1
10040     LINE a(i,1),a(i,2) TO a(i,3),a(i,4)
10050   END IF
10060   IF a(i,0)=2
10070     CIRCLE a(i,1),a(i,2),a(i,3),a(i,4),
          RAD (a(i,5))
10080   END IF
10090   IF a(i,0)=3
10100     LINE a(i,1),a(i,2) TO a(i,3),a(i,4) TO
          a(i,5),a(i,6) TO a(i,7),a(i,8) TO
          a(i,1),a(i,2)
10110   END IF
10120 END DEFine global_draw
```


Testing

Run the program and draw a few shapes, stop the program with ESCape and then clear the screen. Call up the drawing module by typing:

global_draw (0)[ENTER]

and you should see your first shape redrawn. According to the number of shapes you originally drew, you can call up GLOBAL_DRAW with arguments other than zero.

Module 2.1.10: Redrawing the whole design

With a module installed which is capable of redrawing a single line or shape, it becomes a trivial matter to redraw the whole design.

Module 2.1.10: Lines 9000 – 9090

```
9000 REMark ************************
9010 DEFine PROCedure redraw
9020 REMark ************************
9030   IF item>0
9040     CLS
9050     FOR i=0 TO item-1
9060       global_draw (i)
9070     NEXT i
9080   END IF
9090 END DEFine recall
```

Testing

When you have entered a number of lines and shapes, stop the program, clear the screen and type:

redraw[ENTER]

The entire design should be recreated on the screen.

Module 2.1.11: Deleting lines and shapes

Since the design is not stored as a whole but in the form of individual lines and shapes, it is also a simple matter to give the user the option of deleting individual items. This module displays the whole design, line by line, with the option of deleting any line. It can also be used to redraw the design if the screen has been cleared by stopping the program.

Module 2.1.11: Lines 11000 – 11400

```
11000 REMark ************************
11010 DEFine PROCedure remove
11020 REMark ************************
```

```
11030   CLS
11040   i=0
11050   PRINT #0, "DELETE (Y/N): ?"
11060   PRINT #0,\"ESCAPE TO TERMINATE DELETE
        FUNCTION"
11070   REPeat loop
11080     REPeat flashing
11090       OVER -1
11100       global_draw (i)
11110       global_draw (i)
11120       t1$=INKEY$
11130       IF t1$="y" OR t1$="Y"
11140         FOR j=i TO item-1
11150           FOR k=0 TO 8
11160             a(j,k)=a(j+1,k)
11170           NEXT k
11180         NEXT j
11190         item=item-1
11200         EXIT flashing
11210       END IF
11220       IF t1$=CHR$(27)
11230         OVER 0
11240         FOR j=i TO item-1
11250           global_draw (j)
11260         NEXT j
11270         RETurn
11280       END IF
11290       IF t1$<>""
11300         EXIT flashing
11310       END IF
11320     END REPeat flashing
11330     IF t1$<>"y" AND t1$<>"Y"
11340       OVER 0
11350       global_draw (i)
11360       i=i+1
11370     END IF
11380     IF i=item THEN EXIT loop
11390   END REPeat loop
11400 END DEFine remove
```

Commentary

Lines 11080 – 11320: This loop continually draws and redraws one line or shape with OVER set to minus one, so that the item flashes on the screen.

Lines 11130 – 11210: If the user responds with 'Y' to the prompt asking if the item is to be deleted, the rest of the contents of the array above the current item are copied down one place, wiping that item out.

Lines 11220 – 11280: If the ESCape key is pressed, the rest of the design is drawn and the module terminates. This facility is useful for redrawing the design if the program has been restarted with DO 0.

Lines 11290 – 11370: Any key other than 'Y' or 'y' simply sends the module on to the next item in the design, having drawn the current shape on the screen.

Testing

Provided that you have entered some shapes, you should now be able to press 'D' from the main part of the program to obtain this module. Confirm that you can page through the design you have entered, deleting items or leaving them untouched. Pressing ESCape at the beginning of the design should result in no change being made.

Module 2.1.12: Windows and scales

We have so far not touched at all on one of the most useful abilities of Designer — to move the screen like a window over a large design or to shrink a design so that it can be seen on a single screen. Normally, this is a fairly complex mathematical manouevre but the simplicity of this module is an indication of the power that SuperBASIC brings to bear in its graphics commands. The lines of calculations necessary on most other machines are reduced on the QL to a single, simple SCALE command.

Module 2.1.12: Lines 12000 – 12120

```
12000 REMark ************************
12010 DEFine PROCedure scaling
12020 REMark ************************
12030   INPUT #0,"SCALE (50-10000): ";screen
12040   INPUT #0,"ORIGIN X (0-9950): ";org_x
12050   INPUT #0,"ORIGIN Y (0-9950): ";org_y
12060   size=2*screen/100
12070   IF size<2 THEN size=2
12080   SCALE screen,org_x,org_y
12090   cx=org_x+1.6*screen/2
12100   cy=org_y+screen/2
12110   redraw
12120 END DEFine scaling
```

Commentary

Line 12030: On start-up, the design is viewed on a scale where 100 units would represent the height of the screen. This can be altered so that the design can be viewed at twice the size (50 units screen height) or one hundredth the scale (10,000 units screen height).

Line 12040 – 12050: We noted at the beginning that the program allows for designs of up to 10,000 by 10,000 pixels. To begin with, the program presents you with a view of the bottom lefthand corner of the design, beginning at coordinates 0,0. Using the scale command, however, it is per-

fectly possible to move the coordinates represented by the bottom lefthand corner of the screen, so that a different part of the design is seen.

Line 12060: The size of the cursor must vary according to the scale at which the design is being viewed. If the design has been been shrunk by a factor of a hundred, a cursor which is 4/100 pixels across will not be much use.

Lines 12090 – 12100: Having rescaled the screen, the cursor coordinates are recalculated so that they lie in the middle of the screen — note that the screen is always 1.6 times as wide as it is high. Note that moving the screen in relation to the design is the fast way to move the cursor from one point to another.

Testing

With a design entered, you should now be able to move the screen over the design and enlarge or shrink it.

Module 2.1.13: Storing data on microdrive

We now turn to a very important module from the point of view of this program and, indeed, of most of the programs in the book. This module and the next form a unit which allows data generated by a program to be stored on microdrive and then retrieved by the program at a later date.

Module 2.1.13: Lines 13000 – 13170

```
13000 REMark ************************
13010 DEFine PROCedure store
13020 REMark ************************
13030   CLS
13040   AT 1,14 : PRINT "SAVE DATA"
13050   INPUT\\" Name of data file:";file$
13060   tfile$="mdv1_" & file$
13070   DELETE tfile$
13080   OPEN_NEW #8,"mdv1_" & file$
13090   PRINT#8,item
13100   FOR i=0 TO item-1
13110     FOR j=0 TO 8
13120       PRINT #8,a(i,j)
13130     NEXT j
13140   NEXT i
13150   CLOSE#8
13160   redraw
13170 END DEFine store
```

Commentary

Lines 13050 – 13060: The module gives the user the chance to define the name under which the data is to be stored. This is only possible because the QL allows us to use a string variable as a filename when talking to to the microdrives.

Line 13070: Since one object of the program is to be able to work on an existing file, modify it and store it again, we must be careful to remove any files of the same name before attempting to store some data, otherwise the program will stop with an ALREADY EXISTS error message.

Line 13080: The microdrive is instructed to create a new file (OPEN_NEW), which will be accessed along '#8', or channel 8, by the program. The file is to be on microdrive 1 and will appear in the directory under the name recorded in FILE$.

Lines 13090 – 13140: These lines simply print out the data stored in array, not to the screen but to channel 8, which we have already informed the operating system is to be directly associated with the data file we have just opened. Note that in storing on microdrive it is important to store the number of items to be recorded as the first item. This allows the next module, which recalls the data, to know how many items are to be picked up. There are other methods, such as detecting the end of file marker, but they are none of them as fast as knowing exactly how many items are to be obtained.

Line 13150: In order to create a valid file, we must ensure that it is properly terminated. This is done using the CLOSE command, which also frees channel 8 for other uses.

Line 13160: Having corrupted the screen in storing the data, the design is recreated using REDRAW.

Module 2.1.14: Recalling the data

This is a mirror image of the previous module, in allowing data on the microdrive to be picked up and the design which it defines to be recreated.

Module 2.1.14: Lines 14000 – 14160

```
14000 REMark ************************
14010 DEFine PROCedure recall
14020 REMark ************************
14030   CLS
14040   AT 1,14 : PRINT "RECALL DATA"
14050   DIR mdv1_
14060   INPUT\\" Name of data file:";file$
14070   OPEN_IN #8,"mdv1_" & file$
14080   INPUT#8,item
14090   FOR i=0 TO item-1
14100     FOR j=0 TO 8
14110       INPUT #8,a(i,j)
14120     NEXT j
14130   NEXT i
14140   CLOSE#8
14150   redraw
14160 END DEFine recall
```

Commentary

Lines 14050 – 14060: The directory for the cartridge currently in drive 1 is displayed and the user asked to specify which file is to be used.

Line 14070: The name input by the user is used to open an existing file (OPEN_IN).

Lines 14090 – 14140: The information which was stored by the last module is pulled off the drive in exactly the same order.

Line 14150: The design whose details have been obtained is placed on to the screen using REDRAW.

Testing

Ensure that there is a properly formatted microdrive cartridge in drive 1, then create a simple design and use 'M' to store it. Stop the program, then start it again with DO 1 to clear the memory. Answer 'Y' to the prompt asking you if you wish to load from the microdrive. Give the name of the file under which the design was stored and you should see the drive activate and then the design recreated on the screen.

If this test is successful, the program is ready for use.

PROGRAM 2.2: 3-D GRAPH

Program function

This program is intended as a timely warning against over-reliance on the increasing tendency of major manufacturers to make their machines the basis of a much broader package including software. Later in the book you will find several examples of applications which could no doubt be simulated using your Psion software, raising the question of whether you wish to rely solely on what you are given or retain the ability to design for yourself.

The current program is hardly a complex one, but that is precisely the point. In relatively few lines, it succeeds in creating a visual display of certain types of numeric information which is far more striking than anything which the excellent Easel package can produce. A full three-dimensional graph effect is created by the elementary use of the turtle graphics commands, giving a taste of the power they hold for more complex applications.

New concepts introduced during the course of the program include:

1) Flexible use of DATA statements.
2) The turtle graphics commands.

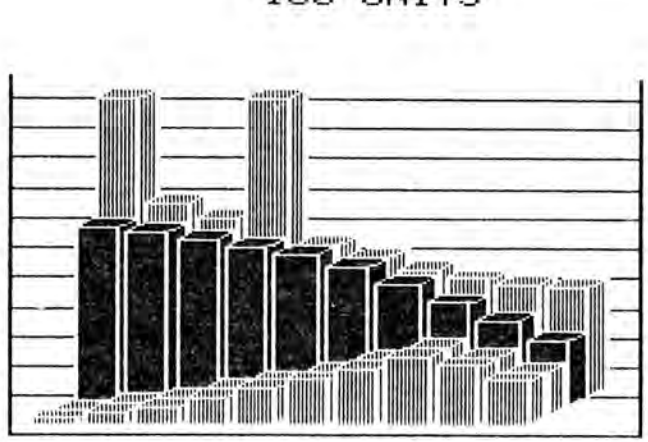


Figure 2.2: Part of Display Generated by 3-D Graph.

Module 2.2.1: Simple line drawing

The beauty of the QL's turtle graphics is the sheer simplicity which they bring to the drawing of shapes, as indicated by this two-line module. Its effect is to accept an instruction to draw a line in a certain direction for a certain distance.

Module 2.2.1: Lines 3000 – 3050

```
3000 REMark **************************
3010 DEFine PROCedure turtle (angle,distance)
3020 REMark **************************
3030   TURNTO angle
3040   MOVE distance
3050 END DEFine turtle
```

Module 2.2.2: Describing a shape

Once it is possible to have a line drawn, it becomes a relatively simple matter to describe most shapes, whether two or three dimensional. The current module lays down a three-dimensional block on the screen once a number of variables have been defined.

Module 2.2.2: Lines 2000 – 2150

```
2000 REMark **************************
2010 DEFine PROCedure blocks
```

```
2020 REMark *************************
2030   LINE start_x,start_y
2040   PENDOWN
2050   turtle 180,wide
2060   turtle 90,high
2070   turtle 0,wide
2080   turtle 180,wide
2090   turtle 30,deep
2100   turtle 0,wide
2110   turtle 210,deep
2120   turtle 270,high
2130   turtle 30,deep
2140   turtle 90,high
2150 END DEFine blocks
```

Commentary

Line 2030: Using the line command with a single set of coordinates moves the graphics cursor, thus defining the start point for the shape to be drawn.

Lines 2050 – 2140: The outline of the block. It is far easier to wait until it is displayed on the screen than to attempt to describe what is happening here.

Testing

A number of variables have to be defined and, in addition, it will be helpful to modify the previous module temporarily in order to allow the shape being drawn to be analysed line by line.

Enter a temporary line:

3045 tt$ = inkey$(– 1)

then type the following in direct mode:

high = 20[ENTER]
wide = 20[ENTER]
deep = 20[ENTER]
ink 7 : paper 0 : cls[ENTER]
start_x = 40[ENTER]
start_y = 40[ENTER]
blocks[ENTER]

You should see a single line drawn on the screen. Press a key and a second line will be drawn. As you continue to press a key each time a line is drawn, what you are seeing is the individual TURTLE instructions being carried out by the previous module. When you have finished the test, don't forget to remove the temporary program line at 3045.

Module 2.2.3:The data for the program

Most of the programs in this book, as you will discover, are interactive — that is to say, as they run they request information from the user and act upon it. This is the ideal method for most purposes, but when limited amounts of data are being processed it is quite possible that the amount of programming involved in providing the interactive routines gets out of proportion to the possible benefits.

One simple solution is to make use of the often neglected DATA statement, which allows information to be recorded within the program itself and recaptured by the program using the READ statement. The advantage of this is that, once information has been recorded in DATA statements, it can easily be saved with a program, or saved as a separate section and later merged with the core of the program. It can easily be displayed by use of nothing more complex than LIST and, of course, changes to the data can be made by use of EDIT. The current module is an example of how a DATA module can be laid out so that the information it contains is easily recognisable.

Module 2.2.3: Lines 5000 – 5380

```
5000 REMark *************************
5010 REMark data
5020 REMark *************************
5030 :
5040   REMark NUMBER OF COLUMNS (1-10)
5050   DATA 10
5060 :
5070   REMark NUMBER OF BANKS (1-4)
5080   DATA 4
5090 :
5100   REMark NAMES FOR BANKS
5110   DATA "Bank 1"
5120   DATA "Bank 2"
5130   DATA "Bank 3"
5140   DATA "Bank 4"
5150 :
5160   REMark NAME FOR VERTICAL AXIS
5170   DATA "VERTICAL"
5180 :
5190   REMark MAXIMUM VALUE VERTICALLY
5200   DATA 100
5210 :
5220   REMark NO. OF MARKERS VERTICALLY
5230   DATA 10
5240 :
5250   REMark NAME FOR HORIZONTAL AXIS
5260   DATA "Horizontal"
5270 :
5280   REMark DATA FOR BANK 1
5290   DATA 100,65,60,100,51,47,43,40,38,37
5300 :
5310   REMark DATA FOR BANK 2
5320   DATA 60,58,56,53,51,47,41,35,29,22
```

```
5330 :
5340   REMark DATA FOR BANK 3
5350   DATA 2,4,6,9,12,15,19,23,20,15
5360 :
5370   REMark DATA FOR BANK 4
5380   DATA 2,4,6,9,12,15,19,23,20,15
```

Commentary

Lines 5040 – 5050: The program as currently written is intended to deal with between one and four rows of three-dimensional columns. According to the number of columns in a row, the width of the individual columns will be adjusted so that the full width of the screen is used.

Lines 5070 – 5080: One to four rows can be displayed, each being drawn in front of the previous row. For this reason, the program is most suited to applications where there is a fair guarantee that one set of figures will contain values which are less than those of a previous set. An example of this might be the turnover and profit of a company over a number of years, where profit is always going to be less than turnover. If the front rows of a particular display are going to be consistently higher than the back rows, the back rows will be effectively hidden and the display of very little use.

Lines 5100 – 5140: The names of the individual rows, or banks, will eventually be displayed at the bottom of the screen in colours appropriate to the four banks.

Lines 5160 – 5230: The name to be displayed against the vertical axis and the maximum number of units it is intended to represent. Figures input for the height of the individual columns will be expressed as a proportion of this maximum figure. The user may specify the number of units into which the axis is to be divided.

Lines 5250 – 5260: The name to be displayed against the horizontal axis of the graph.

Lines 5280 – 5380: The data for the 4*10 columns specified for the graph.

Module 2.2.4: Reading in the data

Having entered the data on which the graph will be based, we now turn to the module which will convert that data into variables which will be used by the drawing routines.

Module 2.2.4: Lines 1000 – 1280

```
1000 REMark **************************
1010 REMark  control
1020 REMark **************************
1030   PAPER 0 : CLS : CLS#0
1040   RESTORE 5000
```



```
1050   READ columns,banks
1060   DIM bank$(banks-1,20)
1070   FOR i=0 TO banks-1
1080     READ bank$(i)
1090   NEXT i
1100   RESTORE 5160
1110   READ v_axis$,v_max,v_marks
1120   READ h_axis$
1130   wide=INT(100/columns)
1140   deep=5
1150   grid
1160   RESTORE 5290
1170   FOR bank=1 TO banks
1180     start_x=25-deep*bank+wide
1190     start_y=10-deep*(bank-1)/2
1200     FOR i=1 TO columns
1210       READ high
1220       high=high*80/v_max
1230       FILL 1 : INK bank+1 : blocks
1240       FILL 0 : INK 0 : blocks
1250       start_x=start_x+wide+2
1260     NEXT i
1270   NEXT bank
1280 STOP
```

Commentary

Lines 1050 – 1090: The number of columns and banks is read from the DATA section and the names for the four banks stored in the array BANK$.

Lines 1100 – 1150: The DATA relating to the vertical axis and the name of the horizontal axis is read. The width of the individual columns is calculated so that one row of columns will span one hundred graphics locations across the screen. The variable DEEP records the amount by which an individual column will appear to go back 'into' the screen. Finally the call to GRID, the next module, will produce the framework within which the eventual graph will be displayed. Note the RESTORE statement which sets the READ instruction to the appropriate line for data. It is important, when setting up your own graphs, that you do stick to the same sections within the data module as shown in the current program. If less than four banks are to be used, do not delete the spare lines and renumber or you may confuse the acquisition of data.

Lines 1160 – 1270: These lines read the figures from the last section of the data module, which represent the height of the individual columns. The two loops ensure that the correct number of columns are created for each bank in turn. The variables START_X and START_Y are set for each bank so that each row of columns will start lower down and to the left of its predecessor. As each column is completed, START_X is incremented so that the row will be drawn from left to right.

Lines 1210 – 1240: The figures for the height of each column are read and recalculated so that they conform to the maximum value laid down for the vertical axis. The line drawing routines are now called up twice. On the first call the columns are drawn with FILL set and with a different colour for each bank — the result is a rather featureless shape. The second call switches off FILL and sets the INK colour to black. The result is that the edges of the column are picked out in black, giving a three-dimensional effect. As each column is completed, START_X is incremented so that the row is drawn from right to left.

Testing

The module can only be tested if the reference to GRID is deleted in line 1150, since that module has yet to be entered, or if the first three lines and the last line of the next module are entered so that there is a dummy procedure called 'grid' for the current module to call up.

Once that has been done, simply run the program and you should see the four banks of columns drawn on a black screen. In comparing the height of the columns with the figures contained in the data module, remember that the 3-D effect means that each bank is slightly lower than the previous one. To find the true height of a column, you need to follow back its top surface to the back row, since it is the front top edge of columns on the back row which accurately represents the height of the column. This will be clearer when the next module has been entered, giving a grid against which to measure the columns.

Module 2.2.5: Drawing the grid for the graph

A simple module which draws lines across the screen at the intervals specified by the user, and which labels the various axes.

Module 2.2.5: Lines 4000 – 4200

```
4000 REMark ************************
4010 DEFine PROCedure grid
4020 REMark ************************
4030   INK 7
4040   v_unit=80/v_marks
4050   LINE 0,96 TO 0,0 TO 150,0 TO 150,96
4060   FOR i=0 TO v_marks
4070     LINE 0,10+i*v_unit TO 150,10+i*v_unit
4080   NEXT i
4090   AT 0,12 : PRINT v_max;" UNITS"
4100   v_axis$=v_axis$ & " " & INT(v_max/v_marks)
       & "*" & v_marks
4110   FOR i=1 TO LEN(v_axis$)
4120     AT i-1,35 : PRINT v_axis$(i)
```



```
4130   NEXT i
4140   PRINT #0,h_axis$
4150   FOR i=0 TO 3
4160     AT #0,i,20 : INK #0,i+2
4170     PRINT #0,bank$(i)
4180   NEXT i
4190   AT #0,1,0
4200 END DEFine grid
```

Testing

Running the program should now result in the same display of columns, but in this case properly labelled.

PROGRAM 2.3: SCREEN

Program function

This program is, in its original form at least, a triumph of technique over commonsense. Its purpose is to provide a screen dump, or copy to a printer, of the contents of the screen. This is not an over-complex task, though the details can be a little fiddly. What it *is*, however, in BASIC, is very time-consuming. The screen covers a lot of memory within the QL (32K), and every byte — indeed every bit — has to be analysed before the full screen dump is achieved.

One of the major problems is that micros and printers work in different directions. The QL, like most other micros, records the contents of the screen in the form of 8-bit bytes, with each bit in a particular byte representing one in a line of eight pixels across the screen (in the QL it isn't quite this simple, but we'll go into that later). Printers, or at least the dot-matrix variety, are usually capable of receiving a byte and interpreting it as an instruction to print eight dots in a line — but vertically.

The result of all this is that, while the screen memory of the micro is held in straightforward bytes and while the printer can be sent straightforward bytes, the whole thing falls apart because, in order to print exactly what is on the screen, the bytes in the micro's memory have to be sliced apart and combined with parts of others. Consider the 64-pixel square in **Figure 2.3**.

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
60 61 62 63 64 65 66 67
70 71 72 73 74 75 76 77
```

Figure 2.3: 64-Pixel Square.

To a typical micro, the small block represented would be stored in the form of eight bytes, the first recording pixels 00 – 07, the second 10 – 17, etc. For a typical printer to print the area represented, the information would have to be supplied in the form of eight bytes, the first recording 00 – 70, the second recording 01 – 71 etc. In the case of the QL the situation is even worse, since, rather than recording the information in easily-addressable blocks of eight bytes, the QL's screen memory is arranged in single-pixel thickness lines right the way across the screen.

Even so, with proper analysis of the screen memory bytes, and time-consuming translations, any printer capable of bit-mapped graphics can be made to print out a black and white copy of the screen. The only problem is that, for most conceivable purposes, all the work of translation is entirely unnecessary. My co-author on several other books, Mark England, pointed out that if, in terms of the 8*8 square of pixels printed above, the printer were sent eight bytes containing 70 – 77,60 – 67,50 – 57, etc, it would print out the square perfectly, except that the piece of paper would need to be turned through 90 degrees. The resulting program takes one-fifth the time to do exactly the same job, except that on some printers, where the horizontal spacing between the pixels is different from the vertical, the image may be squashed one way or another. For the sake of completeness, both methods are included within the program, though you are free to regard the first four modules as nothing more than an interesting example of how to analyse the contents of an area of screen memory, and accordingly omit them.

The arrangement of screen memory on the QL

Before we go any further, it would be wise to take a quick look at the way in which screen memory on the QL is arranged, since it is, like previous Sinclair machines, a little eccentric and has a major effect on the way the program has been written.

In essence, the QL's screen memory consists of 32K of memory (32768 bytes) which begins at address 131072. The screen which the user sees (on a domestic TV) is 1024 pixels across, and a single line across of 1024 pixels is represented by 128 bytes of consecutive memory. Thus a picture of screen memory might look like this:

```
TOP LEFT                                        TOP RIGHT
131072 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 131199
131200 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 131327
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
163584 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 163711
163712 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 163839
BOTTOM LEFT                                  BOTTOM RIGHT
```


This, of course, is the whole screen, including the areas given over to the border and to the #0 command lines at the bottom. The part normally used by the program is 896 pixels wide and 200 pixels high, and its start and finish addresses are as follows:

```
TOP LEFT                                        TOP RIGHT
133128 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 133239
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
158600 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 158711
BOTTOM LEFT                                  BOTTOM RIGHT
```

But even this does not actually define the situation for there is no simple one to one relationship between the bytes and the screen pixels, rather the relationship varies according to the screen mode being employed.

In mode 8, the lower resolution mode, each group of eight pixels horizontally has two bytes controlling it. The two bytes dictate four characteristics about each pixel in the group — whether their colour includes green, red, and blue, and whether they are flashing. Elementary maths leads to an obvious conclusion here. Two bytes contain 16 bits, so, using two bytes, there is no way in which four facts can be recorded about eight pixels, ie 32 facts. The solution to this is that in the lower resolution mode the pixels are grouped in horizontal pairs, so that each characteristic applies to two pixels in a group of four double pixels. Each two memory bytes are arranged in the following manner:

```
                    BYTE 1                          BYTE 2
PIXEL PAIR  1/2   3/4   5/6   7/8     1/2   3/4   5/6   7/8
           [GF]  [GF]  [GF]  [GF]    [RB]  [RB]  [RB]  [RB]
```

where F records whether FLASH is set for the pixel pair and G, R and B record whether green, red and blue are included in its colour.

In high resolution mode, where the four available colours can all be produced from two primary colours, and the flash characteristic is not available, there are accordingly only two pieces of information to be recorded about each pixel, and two bytes can deal with eight individual pixels, rather than grouping them into pairs:

```
                    BYTE 1                          BYTE 2
PIXEL No.   1  2  3  4  5  6  7  8     1  2  3  4  5  6  7  8
           [G][G][G][G][G][G][G][G]   [R][R][R][R][R][R][R][R]
```

In the program which follows, we take advantage of this two-byte structure to exclude certain colours from the screen dump, thus allowing a greater

variety of background colours. The ever-present problem with screen dumps is how to choose the basis on which a pixel is going to be regarded as 'set' and therefore printed — if all the colours are regarded as setting the pixel, then only areas of the screen which are black (ie have no colour whatsoever) will be left blank in the screen dump. For the sake of simplicity I have chosen to look only at the second of the two bytes when choosing pixels which will be printed. This means that in both low and high resolution, backgrounds (or lettering) of either black or green will be left blank on the paper.

Module 2.3.1: Control

In use, this module would be altered substantially. In its present form, its job is to fill the screen with a range of characters before calling up the modules which carry out the screen dump.

Module 2.3.1: Lines 12000 – 12190

```
12000 REMark ************************
12010 DEFine PROCedure screen
12020 REMark ************************
12030   PAPER 0 : INK 7 : CLS : CLS#0
12040   FOR i=1 TO 20
12050     PRINT "*";FILL$(CHR$(47+i),35);"*"
12060   NEXT i
12070   CIRCLE 80,50,20
12080   CIRCLE 80,50,40
12090   REPeat check
12100     INPUT #0,"QUICK OR SLOW (Q/S): ";Q$
12110     IF Q$="q" OR Q$="Q" OR Q$="s" OR Q$="S"
          THEN EXIT check
12120   END REPeat check
12130   CLS#0
12140   OPEN #8,ser1
12150   PRINT #8,CHR$(27);"A";CHR$(8)
12160   IF Q$="s" OR Q$="S" THEN line_start
12170   IF Q$="q" OR Q$="Q" THEN line_start_2
12180   CLOSE #8
12190 END DEFine screen
```

Commentary

Lines 12090 – 12120: These lines allow the user to specify which of the two methods of dumping the screen is to be employed — see the introduction to the program for further explanation.

Line 12150: Any screen dump program must be aimed at a specific printer or printers and be capable of generating the specific commands needed by that particular machine. The current program is intended for an Epson FX-80 dot-matrix printer. This particular sequence of characters sets the spacing between each line of characters printed at 8/72 of an inch, removing the small gap which is normally left between each line. If you are using a

different printer, you will have to consult your manual as to the correct method of achieving this or, indeed, any of the special printer commands contained within the program. If your printer is not capable of altering the distance between lines, you will probably find that, when solid blocks extend over two character lines, a thin white line will be seen.

Testing

There is no effective way of testing any of these modules until the whole screen dump routine is entered. The program is not a long one, so this should not represent a major inconvenience.

Module 2.3.2: Stepping down the screen

The method in this first program section will be to scan the screen from left to right in 8*8 pixel squares, thus allowing a block of eight bytes to be translated for the benefit of the printer. Since the screen memory is arranged in lines of one pixel thickness which cross the entire width of the screen before beginning again on the next line down, not in neat 8*8 pixel squares, the first need is to identify the start position in memory of each line of 8*8 blocks.

Screen memory for the central part of the screen starts at address 133128. To move eight pixels down the screen we have to move on 1024 bytes through the screen memory — eight lines, each 128 bytes long. The loop which determines the start of each line of 8*8 pixel squares will therefore start at 133129 (remember that we are using only the second of each pair of bytes) and move through the two hundred lines of pixels down the screen in steps of eight lines (8*128).

Module 2.3.2: Lines 4000 – 4090

```
4000 REMark ************************
4010 DEFine PROCedure line_start
4020 REMark ************************
4030   FOR l_start=133129 TO 133129+199*128 STEP
       128*8
4040     t$=""
4050     read_line
4060     PRINT #8,CHR$(27);"L";CHR$(192);CHR$(1);
4070     PRINT #8,t$
4080   NEXT l_start
4090 END DEFine line_start
```

Commentary

Line 4060: In the case of the FX-80 printer, the sequence CHR$(27);"L" puts the printer into bit-mapped graphics mode. CHR$(192);CHR$(1) describe the number of items of data which the printer should expect to receive before printing a line. This is set to 448 (192 + 256*1) because we shall be printing one vertical byte for half the number of bits which represent the

width of the screen. The screen is 112 bytes wide, giving 8*56 vertical lines to be sent to the printer before one line is complete. For other printers, you will need to know what the sequence is to enter bit-mapped mode, and whether they accept a block of data or print each vertical line as it is received — consult your manual.

Line 4070: The string of characters which later modules have built up, representing the first eight lines on the screen.

Module 2.3.3: Stepping across the screen

Having found the beginning of each block of eight lines, this loop scans across the screen in steps of two bytes, thus selecting the second of the two bytes which refer to each group of pixels.

Module 2.3.3: Lines 3000 – 3060

```
3000 REMark ************************
3010 DEFine PROCedure read_line
3020 REMark ************************
3030   FOR start = l_start TO l_start+111 STEP 2
3040     read_bit
3050   NEXT start
3060 END DEFine read_line
```

Module 2.3.4: Stepping through eight bits

We can now identify the position of a single byte in the screen memory but, as we have seen, that byte is of no interest to the printer, which prints downwards rather than across. Referring back to Figure 2.3, what we have to do is to identify eight bytes which fall in a line downwards and then to read the 64 pixels in the form of eight downwards bytes. The first step is to set up a loop which reads through eight bit positions.

Module 2.3.4: Lines 2000 – 2060

```
2000 REMark ************************
2010 DEFine PROCedure read_bit
2020 REMark ************************
2030   FOR bit=0 TO 7
2040     read_8
2050   NEXT bit
2060 END DEFine read_bit
```

Module 2.3.5: Translating from horizontal bytes to vertical

The final module required by this method takes one bit from each of the eight bytes in the block and creates a new byte out of them which is a recording of one eight-pixel vertical line on the screen.

Module 2.3.5: Lines 1000 – 1080

```
1000 REMark ************************
1010 DEFine PROCedure read_8
1020 REMark ************************
1025   t_byte=0
1030   FOR byte=0 TO 7
1040     IF PEEK (start+byte*128) && 2^(7-bit)
1050       t_byte=t_byte+2^(7-byte)
1060     END IF
1070   NEXT byte
1075   t$=t$ & CHR$(t_byte)
1080 END DEFine read_8
```

Commentary

Line 1025: T_BYTE will be used to store the vertical byte as it is built up.

Lines 1040 – 1060: These lines use the && operator to discover whether a particular bit is set in each of the eight bytes which the loop shuttles through. The effect of && is to compare two numbers and to return a result consisting of a number which has a binary '1' wherever *both* of the compared numbers also have a binary '1'. Thus comparing:

16 — 00010000 *with* 127 — 01111111

the result is 16, since that is the only bit which is set in both. To find out if a particular bit is set within a byte, all we have to do is to compare 2^BIT NUMBER with the byte. If the result is still 2^BIT NUMBER, then that bit is set in the byte. The loop therefore shuttles down the eight bytes, using these lines to detect which of the eight bits are set and adding the value of the vertical byte. Note that though the memory is read from top to bottom, the particular printer involved requires the pixels to be represented in the opposite order, so the value added to the vertical byte is 2^(7 – BYTE), not 2^BYTE.

Line 1075: Having run through the eight bytes and created one vertical byte, the value of this is added to the string which will eventually be printed out.

Testing

If you have an Epson FX-80 printer, or one of the many other types which have a compatible set of commands, you are now in a position to type SCREEN, call for a slow printout and see the demonstration screen created and dumped to the printer. Be warned that this process will take some 20 minutes or more.

If you have a printer which is not compatible with the Epson command set, I am afraid it is a matter of adjusting the printer commands contained within the program to your own machine. Provided that you have a bit-mapped graphics capability, however, this should not be difficult.

Module 2.3.6: Stepping through the lines with the sideways method

We now turn to the first of the two modules which produce a quicker printout by turning the image sideways, thus reducing the amount of calculation to be done. I will not go into detail in the commentary on these three modules, since they are far simpler to follow than those for the previous method.

The current module starts the program reading the bottom lefthand byte of the screen. Later modules will dictate that each line is read from the bottom of the screen to the top, a single line of bytes. Each new line will start, therefore, on the next byte along the bottom of the screen, or rather the next but one since we are only using every other byte. The bottom to top lines are 200 bytes long but, as you can see from line 11060, the printer is told to expect 400 bytes. This is because on the particular printer being used, a better effect is gained by printing each set of pixels twice.

Module 2.3.6: Lines 11000 – 11090

```
11000 REMark ************************
11010 DEFine PROCedure line_start_2
11020 REMark ************************
11030   FOR l_start=158601 TO 158601+111 STEP 2
11040     t$=""
11050     read_line_2
11060     PRINT #8,CHR$(27);"L";CHR$(144);CHR$(1);
11070     PRINT #8,t$
11080   NEXT l_start
11090 END DEFine line_start_2
```

Module 2.3.7: Stepping up the screen

Having established the bottom of a line of bytes to be read, all that remains is to step up the line, jumping 128 bytes each time and placing the byte found into T$ to be printed. In fact, as mentioned, the effect is better with the FX-80 if each byte is printed twice.

Module 2.3.7: Lines 10000 – 10060

```
10000 REMark ************************
10010 DEFine PROCedure read_line_2
10020 REMark ************************
10030   FOR byte = l_start TO l_start-199*128
        STEP -128
10040     t$=t$ & CHR$(PEEK(byte)) & CHR$(PEEK
          (byte))
10050   NEXT byte
10060 END DEFine read_line_2
```


Testing

As with the previous method, but this time you will find that a column of characters is printed across the screen every eight seconds or so.

Using the program

As it is presently set up, the program is no more than a demonstration of the fact that a screen dump can be done. In use, it should be renumbered with a high starting number and line increments of 1, so that it can be merged in with the program from which you want a dump. When the screen display is as you want it, either stop the program and enter 'screen' or call up SCREEN from within the program. Remember that the program is designed to ignore green and black. If the colour combinations you have chosen give problems, you will have to use the RECOL command to achieve something which makes sense when translated into black and white.

PROGRAM 2.4: CHARACTERS

Program function

Having looked at some of the high resolution capabilities of the QL, we now turn to one slightly unusual aspect of using low resolution graphics — a program to allow you to change the shape of the characters which the QL prints on the screen. But before we go on to the program proper, a word of explanation is needed about the way characters are created and printed on the screen.

The QL's screen, in low resolution mode (mode 8), has space for 20 lines, each of 37 characters, a total of 740 character spaces. In other words you could print 740 separate items on the screen, though some of them would be the same since the QL cannot generate 740 *different* characters at the same time. That 740, however, is not the end of the story. If you were to look closely at any character on the screen you would find that it is not composed of solid lines, like the words you are reading now, but of dots. In fact every one of the character spaces on the screen is made up of 64 dot positions and it is combinations of these 64 dots which make up each character the QL can display. The letter 'A', for instance, might be made up as in **Figure 2.4**.

The dots which make up the characters are known as 'pixels', which is short for 'picture elements', and they represent the smallest item which the QL can handle on the screen, though, in mode 8, the QL will only handle two pixels at a time.

Such complex shapes do not appear by chance, and it is clear that somewhere in the QL's memory it must be laid down that, when you press the

```
          ====================
ROW 1 --> !        0 0         !
    2 --> !      0 0 0 0       !
    3 --> !    0 0     0 0     !
    4 --> !    0 0 0 0 0 0     !
    5 --> !    0 0     0 0     !
    6 --> !    0 0     0 0     !
    7 --> !    0 0     0 0     !
    8 --> !                    !
          ====================
```

Figure 2.4: Enlargement of Letter 'A' as it might Appear On-screen.

key labelled 'A', the pattern of dots shown in the illustration appears on the screen. In fact, all the characters which the QL can print are stored in the form of numbers, in the QL's ROM, an area called the 'character memory' or 'character ROM'. Each character is allocated nine bytes of memory, and each of those bytes determines where the pixels shall appear on one row of the character. This done by making the value of each byte into a picture of the row in the binary numbering system used by the QL, where numbers are expressed in terms of powers of the number 2 rather than the number 10, as in our normal counting.

Thus, 201300 in our usual way of counting means:

(2*10^6) + (0*10^5) + (1*10^4) + (3*10^3) + (0*10^2) + (0*10^1) + (6*10^0)

In binary, however, the only digits allowed are '1' and '0', and a number like:

11001010

means:

(2^7) + (2^6) + (2^3) + (2^1) — ignoring the zeros

A full understanding of binary is not necessary, provided that you remember that a single byte of memory in the QL is capable of holding one eight-digit binary number, and that all those ones and zeros are a perfect way of recording which pixels in a single row of a character are switched on. The letter 'A', for instance, could be represented by the eight binary numbers:

```
0 0 0 1 1 0 0 0
0 0 1 1 1 1 0 0
0 1 1 0 0 1 1 0
0 1 1 1 1 1 1 0
0 1 1 0 0 1 1 0
0 1 1 0 0 1 1 0
0 1 1 0 0 1 1 0
0 0 0 0 0 0 0 0
```


If you look closely you can still see the 'A' quite clearly, this time painted in '1's rather than pixels, though you'd have a hard time recognising it as:

24,60,102,126,102,102,102,0

which is what the binary numbers are when translated into our more comfortable decimal system of numbering.

The point of all this is that, when you call for a character to be printed on the screen, the QL looks at the contents of the 'character memory' and uses what it finds there to draw the character on the screen.

It follows from all of this that if it were possible to *change* the contents of the character memory, the shape of the characters printed on the screen would also change. We could have customised lettering, new graphics characters, anything at all which would fit into the basic 8*8 character square.

The problem, however, is that the character memory *cannot* be altered. It is part of the ROM, the 'read only memory' which is permanently built in to the QL. What we *can* do is to copy it to somewhere else in RAM, the kind of memory that can be changed, and then tell the ever-obedient QL that that is where character details are now to be taken from. Having done that, we can muck about with the character set as much as we like — and that is the purpose of the following program.

```
'F1'  INK IN SQUARE
'F5'  BLANK SQUARE
'B'  BLOCK SHIFT LEFT
'I'  INVERT
'M'  MIRROR
'T'  TURN
'P'  PLACE IN MEMORY
'S'  SAVE ON MICRODRIVE
'L'  LOAD FROM MICRODRIVE
'A'  ANOTHER CHARACTER
'H'  CSIZE HEIGHT
'W'  CSIZE WIDTH
'ESCAPE'  TERMINATE
CURSOR ARROWS TO MOVE
```

Figure 2.5: Typical Display from Characters.
The display shows the screen during the character editing code, with a letter 'A' which has been rotated by 180 degrees.

Module 2.4.1: Initialisation

A standard initialisation module, which also calls up the next module to transfer the character data from ROM to RAM. Note, as in previous pro-

grams, the provision to start the program without initialising, by entering DO 0. This is particularly important in the case of the present program because part of the initialisation process is the setting aside of an area of memory to hold character data. If, having set up the reserved area, the program is initialised again, *more* memory will be reserved.

Module 2.4.1: Lines 11000 – 11060

```
11000 REMark ************************
11010 DEFine PROCedure initialise
11020 REMark ************************
11030   DIM array(7,7)
11040   ch=32 : key=0 : high=0 : wide=0
11050   transfer
11060 END DEFine initialise
```

Module 2.4.2: Transferring the character set

This short module is all that is needed to transfer the entire character set for #0 into RAM and then to set up all the variables necessary to switch the QL's attention to it. The actual switching will be accomplished by the control module since there is no need to go through the whole process of transfer if the program is stopped and started again. The ease with which the whole process can be accomplished depends on the fact that the QL keeps a record in RAM of where the character set is to be found. This register is set up when the QL is switched on but it can be changed to indicate another start address.

Module 2.4.2: Lines 24000 – 24100

```
24000 REMark ************************
24010 DEFine PROCedure transfer
24020 REMark ************************
24030   char_reg=167722
24040   rom_start=PEEK_L(char_reg)
24050   user_start=RESPR(875)
24060   char_start=user_start+11
24070   FOR i=0 TO 875 STEP 4
24080     POKE_L user_start+i,PEEK_L(rom_start+i)
24090   NEXT i
24100 END DEFine transfer
```

Commentary

Line 24030: This the address at which is found the two-byte register recording the start of the character data for channel 0.

Line 24040: ROM_START is set equal to the contents of the register so that the program will (a) know where to copy the data from and (b) know where to tell the system to find the original characters when the program terminates.

Line 24050: The RESPR command is used to achieve two things, the setting aside of 876 bytes of memory in the space allocated to the procedures and the setting of the variable USER_START to the beginning of that area. This is the area which will be used to store the character data. The characters which can be printed, and therefore which we can manipulate, are 32 (space) to 127 (copyright symbol), 96 in all. Each of them requires nine bytes to define, making a total of 864 bytes, plus 11 attached to the beginning of the character set for internal housekeeping purposes. Note that this allocation of memory is permanent in any one session. Loading subsequent programs will leave the memory area reserved — to free it, you must restart the system.

Line 24060: As indicated in the commentary on the last line, the first 11 bytes of the character data are of no use to us, so the actual characters will start at USER_START plus 11.

Lines 24070 – 24090: The ROM character data is read, four bytes at a time using PEEK_L and placed into the specially reserved area of RAM.

Testing

Type:

transfer[ENTER]
poke_l char_reg,user_start[ENTER]
print 123

If everything is well, you will see 123 printed on the screen in a perfectly normal way, even though you have instructed the system to take its character data from the reserved area of RAM. Before continuing, it might be wise to enter:

poke_l char_reg,rom_start

to switch back to the ROM character set.

Module 2.4.3: Displaying a magnified character

The essence of this program is that it will make it easy to edit characters. One way in which this is achieved is by printing an enlarged version of a specified character so that a cursor can be moved around it. This module allows the character to be specified and then prints it.

Module 2.4.3: Lines 12000 – 12270

```
12000 REMark ************************
12010 DEFine PROCedure grid
12020 REMark ************************
```

```
12030   REPeat screen
12040     in$=""
12050     PAPER 4 : CLS
12060     PAPER 3
12070     FOR i=0 TO 7
12080       AT i,8 : PRINT " "
12090     NEXT i
12100     PRINT FILL$(" ",9)
12110     FOR i=0 TO 7
12120       byte=PEEK(char_start+(ch-32)*9+i)
12130       FOR j=7 TO 0 STEP -1
12140         array(i,7-j)=(2^j=(byte && 2^j))
12150       NEXT j
12160     NEXT i
12170     redraw
12180     INK 0 : PAPER 4
12190     AT 12,1 : PRINT "CHARACTER NUMBER: ";ch
12200     INPUT\\" NUMBER TO MOVE (0=REDEFINE): ";q$
12210     ch=ch+q$
12220     IF ch<32 THEN ch=32
12230     IF ch>100 THEN ch=100
12240     IF q$="0" THEN change
12250     IF key=27 THEN EXIT screen
12260   END REPeat screen
12270 END DEFine grid
```

Commentary

Lines 12060 – 12090: The outline of an 8*8 box is printed in the top left-hand corner of the screen, using inverse spaces. The character itself is defined on an 8*9 (9 bytes of 8 bits) grid but one of the lines is reserved to maintain the spacing between characters, so we shall work with only eight lines.

Lines 12110 – 12170: These two loops scan each of the binary digits of each of the eight bytes recording the character shape. The variable CH records the number of the character to be extracted — in the INITIALISATION module it is set to 32, or space, the first of the characters in the set. The && operator is used to determine whether a particular digit is '1' or '0'.

In the program itself, the value of the loop variable J is used to create the eight powers of 2 which can be contained in a single byte and then to test whether they are present in the byte being examined by use of &&. If a '1' *is* found, then the corresponding element of ARRAY is set to 1, indicating the presence of a set pixel. Eventually, the next module REDRAW will be used to draw the picture contained in ARRAY within the box, with an inverse space where there would be a pixel — if not, a space is printed. In this way an enlarged version of the character shape recorded in the bytes is printed.

Lines 12200 – 12230: You can change the character on display by adding to or subtracting from the value of CH, within the range of character codes

from 32 to 127. To move on to another number simply enter a number to add or subtract from the current value of CH (eg 10 or – 10).

Testing

This must wait for the entry of the next module, which is used to draw the enlarged character.

Module 2.4.4: Drawing the shape

Nothing complex here, simply a matter of printing a black space in the appropriate position for every element which is set to 1 in ARRAY.

Module 2.4.4: Lines 15000 – 15120

```
15000 REMark ************************
15010 DEFine PROCedure redraw
15020 REMark ************************
15030   FOR i=0 TO 7
15040     FOR j=0 TO 7
15050       AT i,j
15060       PAPER 4
15070       IF array(i,j)=1 THEN PAPER 0
15080       PRINT " "
15090     NEXT j
15100   NEXT i
15110   PAPER 4
15120 END DEFine redraw
```

Testing

You can now make a proper test of what you have entered so far. Make sure the program is initialised, run the test suggested for the TRANSFER module and then type:

grid[ENTER]

You should see the square drawn and the prompt for an input displayed. By entering positive or negative numbers you should be able to page through the available characters, seeing them displayed in large format on the screen. As with TRANSFER, it would be wise to move the character set back into ROM before proceeding.

Module 2.4.5: A flashing cursor

This is a straightforward module similar to the one you entered in Designer. The only real difference is that here what is being flashed is simply an asterisk.

Module 2.4.5: Lines 14000 – 14140

```
14000 REMark ************************
14010 DEFine PROCedure flash_test
14020 REMark ************************
14030   REPeat loop
14040     OVER -1 : INK 4
14050     AT y1,x1 : PRINT "*"
14060     FOR delay=1 TO 20 : NEXT delay
14070     AT y1,x1 : PRINT "*"
14080     FOR delay=1 TO 20 : NEXT delay
14090     in$=INKEY$(1)
14100     INK 0 : OVER 0
14110     IF in$<>"" THEN EXIT loop
14120   END REPeat loop
14130   key=CODE(in$)
14140 END DEFine flash_test
```

Commentary

Line 14130: It may seem odd to take the code of a string character to work on, but this deals with a major problem encountered on my particular version of the QL, and that is the inability to distinguish between characters whose codes are 160 apart. Tests for cursor characters and function keys are rendered a nonsense, since they also pick ordinary characters with codes 160 less than the characters being tested for.

Testing

Type:

x1 = 0 : y1 = 0[ENTER]
flash_test[ENTER]

and you should see a flashing '*' in the top lefthand corner of the screen. Press a key and the program will stop.

Module 2.4.6: Entering commands

This module combines to present a menu, to move the flashing cursor, to ink in or erase the magnified pixels and to accept commands to manipulate the current character. Instructions for its use are contained in the menu itself.

Module 2.4.6: Lines 13000 – 13600

```
13000 REMark ************************
13010 DEFine PROCedure change
13020 REMark ************************
13030   x1=0 : x2=0 : y1=0 : y2=0
13040   AT 10,0 : CLS 2
13050   OPEN #4,scr
```



```
13060   WINDOW #4,320,240,160,16
13070   PAPER #4,4 : INK #4,0
13080   AT #4,1,0
13090   PRINT #4,"'F1' INK IN SQUARE"
13100   PRINT #4,"'F5' BLANK SQUARE"
13110   PRINT #4,"'B' BLOCK SHIFT LEFT"
13120   PRINT #4,"'I' INVERT"
13130   PRINT #4,"'M' MIRROR"
13140   PRINT #4,"'T' TURN"
13150   PRINT #4,"'P' PLACE IN MEMORY"
13160   PRINT #4,"'S' SAVE ON MICRODRIVE"
13170   PRINT #4,"'L' LOAD FROM MICRODRIVE"
13180   PRINT #4,"'A' ANOTHER CHARACTER"
13190   PRINT #4,"'H' CSIZE HEIGHT"
13200   PRINT #4,"'W' CSIZE WIDTH"
13210   PRINT #4,"'ESCAPE' TERMINATE"
13220   PRINT #4,"CURSOR ARROWS TO MOVE"
13230   CLOSE #4
13240   REPeat keys
13250     AT 12,0 : PRINT "          "
13260     PRINT "          "
13270     CSIZE wide,high
13280     AT 12/(high+1),4 : PRINT CHR$(ch)
13290     CSIZE 0,0
13300     flash_test
13310     IF x1>0 AND key=192 THEN x1=x1-1
13320     IF x1<7 AND key=200 THEN x1=x1+1
13330     IF y1>0 AND key=208 THEN y1=y1-1
13340     IF y1<7 AND key=216 THEN y1=y1+1
13350     IF key=232
13360       array(y1,x1)=1
13370       AT y1,x1 : PAPER 0 : PRINT " "
13380     END IF
13390     IF key=248
13400       array(y1,x1)=0
13410       AT y1,x1 : PAPER 4 : PRINT " "
13420     END IF
13430     SELect ON key
13440       ON key=66 : block_shift
13450       ON key=73 : invert
13460       ON key=77 : mirror
13470       ON key=84 : rotate
13480       ON key=80 : memory
13490       ON key=72 : high=1-high
13500       ON key=87 : wide=wide+1-4*(wide>2)
13510       ON key=83 : store : EXIT keys
13520       ON key=76 : recall : EXIT keys
13530       ON key=65 : EXIT keys
13540       ON key=27
13550         POKE_L char_reg,rom_start
13560         EXIT keys
13570     END SELect
13580     PAPER 4
13590     END REPeat keys
13600 END DEFine change
```

Commentary

Lines 13060 – 13230: The menu for the program is printed in a WINDOW opened to the righthand side of the screen. This has the advantage of making it easier to position the items in the menu without having to use AT, but also ensures that the menu will not become corrupted by changes made to the character set. All we are editing with this program, remember, is the character set for #0 — opening a new window called #4 means an unaffected character set.

Lines 13250 – 13290: The current character is printed underneath the 8*8 square, in the current character size. Since it is printed on the normal screen, #0, if its configuration in memory is changed, its appearance will change.

Lines 13310 – 13340: These lines detect the use of the cursor keys and alter the coordinates of the flashing cursor.

Lines 13350 – 13420: The tests for the two function keys F1 and F5.

Lines 13490 – 13500: Pressing the H or W key shuttles through the legal values for the CSIZE command. The reason for allowing the user to change the CSIZE of the character displayed by lines 13250 – 13290 is that the QL interprets character data differently according to the CSIZE of the character — setting pixels in columns 1, 7 and 8 has unpredictable results. When designing a character to be used at a larger CSIZE than 0,0 the H and W keys should be used to view it at the appropriate size.

Line 13540: If the program is terminated properly, through the menu, the address of the ROM character set is POKEd back into the register which tells the QL where to look for character data. It is important to terminate the program *only* through the menu, since otherwise running the program again could result in the system becoming confused as to the correct value for ROM_START and so on.

Testing

Start the program with 'grid' and select a character. When it has been drawn you should be able to enter '0' and see the menu displayed. You should be able to move the flashing cursor around the square without corrupting it. Pressing the function key F1 will ink in a new pixel, while F5 will erase one. None of the other commands, with the exception of 'A' to return to the previous module, will have any effect apart from stopping the program with an error message. Note that throughout the use of this menu, *only capital letters will be accepted as inputs*, lower case letters will have no effect.

Module 2.4.7: Creating an inverse character

We now turn to a series of modules which make the task of editing a character a little easier by performing operations on the whole character, like turning it into its mirror image or rotating it through 90 degrees. The current module simple creates an inverse of the existing character by swapping all the ones and zeros in ARRAY and then calling REDRAW. Any pixel which was set will be erased, and any position which was blank will have a pixel printed in it.

There is a difficulty about using this module, however. Although the program gives you the ability to edit the character on the 8*8 grid, bits 7, 1 and 0, that is to say the leftmost and two rightmost columns, are not used by the QL for the character. Setting pixels in these columns can lead to unpredictable results when the character is displayed, with chunks of it disappearing completely. This needs to be remembered when inverting characters, which will usually mean inking in all the pixels in those columns. Unless you are looking for special effects, you may want to go through the character after inversion, clearing columns 1, 7 and 8.

Module 2.4.7: Lines 17000 – 17130

```
17000 REMark ************************
17010 DEFine PROCedure invert
17020 REMark ************************
17030   FOR i=0 TO 7
17040     FOR j=0 TO 7
17050       IF array(i,j)=0
17060         array(i,j)=1
17070       ELSE
17080         array(i,j)=0
17090       END IF
17100     NEXT j
17110   NEXT i
17120   redraw
17130 END DEFine invert
```

Testing

Start the program with 'grid' and enter the edit mode. When the menu is displayed, press 'I' and you should see an inverse character created. Now press 'I' again and the character should be restored to normal. Note that this only refers to the magnified character — not the normal-sized one to the right of the 8*8 square. The normal-sized character will only change if you decide to enter your edited character into memory using a later module.

Module 2.4.8: Copying the array

Before we go on to the other routines to manipulate a character, we shall need this one which copies an array called TEMP into ARRAY. TEMP will

be used as a temporary storage place when manipulations are being made on ARRAY.

Module 2.4.8: Lines 16000 – 16080

```
16000 REMark ************************
16010 DEFine PROCedure array_copy
16020 REMark ************************
16030   FOR i=0 TO 7
16040     FOR j=0 TO 7
16050       array(i,j)=temp(i,j)
16060     NEXT j
16070   NEXT i
16080 END DEFine array_copy
```

Module 2.4.9: Creating a mirror image

This module takes the character displayed and 'turns it over' as if it were being viewed in a mirror. Unlike the INVERT module, which operated directly on ARRAY, this one uses the array TEMP to store the result until the transformation is completed. The reason for this is that when moving items around within the 8*8 square, the program would be overwriting the original contents and so would be come confused as to what had to *be* moved and what had already *been* moved.

Module 2.4.9: Lines 18000 – 18130

```
18000 REMark ************************
18010 DEFine PROCedure mirror
18020 REMark ************************
18030   DIM temp(7,7)
18040   FOR i=0 TO 7
18050     FOR j=0 TO 7
18060       IF array(i,j)=1
18070         temp(i,7-j)=1
18080       END IF
18090     NEXT j
18100   NEXT i
18110   array_copy
18120   redraw
18130 END DEFine mirror
```

Commentary

Lines 18060 – 18080: The lines which are being read from left to right in ARRAY are copied into TEMP from right to left. Note that there is only a need to copy the elements set to 1, since TEMP was filled with zeros when it was dimensioned at the beginning of the module.

Testing

Start the program with 'grid' and call up the editing menu. Press 'M' and,

after a pause while the character is being copied into the array, you should see it printed in mirrored form.

Module 2.4.10: Turning a character

If you think of the character you are editing as being printed on a sheet of transparent plastic, then apart from holding it at an angle, everything you could do with that plastic sheet can be accomplished by a combination of mirroring the character and/or turning it through 90 degrees one or more times. The current module again uses the array TEMP, but this time to turn the character in the 8*8 square 90 degrees anti-clockwise.

Module 2.4.10: Lines 19000 – 19130

```
19000 REMark ************************
19010 DEFine PROCedure rotate
19020 REMark ************************
19030   DIM temp(7,7)
19040   FOR i=0 TO 7
19050     FOR j=0 TO 7
19060       IF array(i,j)=1
19070         temp(7-j,i)=1
19080       END IF
19090     NEXT j
19100   NEXT i
19110   array_copy
19120   redraw
19130 END DEFine rotate
```

Commentary

Lines 19060 – 19080: Rotation is achieved by reversing the two coordinates I and J when an element is copied into TEMP, and also inverting J so that 7 – J is used.

Testing

Start the program with 'grid', call up the editing menu and then press 'T'. After a pause, the character will be reprinted, turned through 90 degrees. If you press 'T' three more times, the character should be restored to its original position. Try experimenting with combinations of mirror, turn and inverse until you are familiar with their effects.

Module 2.4.11: Shifting a character left

One final simple operation is provided by this module, which allows a character to be shifted to the left by one position. If pixels move off the lefthand edge of the square, they reappear at the right. In fact, by rotating the character before it is shifted and then rotating it again, characters can be shifted in any direction within the square.

Module 2.4.11: Lines 20000 – 20130

```
20000 REMark ************************
20010 DEFine PROCedure block_shift
20020 REMark ************************
20030   DIM temp(7,7)
20040   FOR i=0 TO 7
20050     FOR j=0 TO 7
20060       IF array (i,j)=1
20070         temp(i,j-1+8*(j=0))=1
20080       END IF
20090     NEXT j
20100   NEXT i
20110   array_copy
20120   redraw
20130 END DEFine block_shift
```

Module 2.4.12: Entering an edited character into memory

So far, you have been able to manipulate the magnified character in the 8*8 square until it perhaps bears no relation to the original pattern. All of this, however, has made absolutely no difference to the normal-sized version of the character which is printed underneath the 8*8 square. The changes you have made have not yet been entered into the character memory and they will not be entered until you are satisfied with what you have created. Once you *have* arrived at what you want, however, this module will make the pattern in the square into part of your customised character set.

Module 2.4.12: Lines 21000 – 21100

```
21000 REMark ************************
21010 DEFine PROCedure memory
21020 REMark ************************
21030   FOR i=0 TO 7
21040     byte=0
21050     FOR j=0 TO 7
21060       byte=byte + 2^(7-j)*(array(i,j)=1)
21070     NEXT j
21080     POKE char_start+(ch-32)*9+i,byte
21090   NEXT i
21100 END DEFine memory
```

Commentary

Line 21060: The effect of this line is to take a position in which there is a '1' in ARRAY and to translate it into a binary digit in a number which will represent the row of eight elements, zeros and ones, in which it lies. Each element set to 1 will represent a power of 2 in the same way as when we earlier analysed how a binary number could be used to represent one line across the character. When each of the eight lines has been translated into a number, it is POKEd into the character memory at a position corresponding to one of the lines across the current character.

Testing

Start the program and move the character pointer on to character 65, which is the 'A'. Move into edit mode and turn the character twice — if you turn it only once and store it in memory you will turn the letter A into two dots, since the top of the letter will be in the forbidden lefthand column. Now press 'P' and watch the screen. The 'A' beneath the 8*8 square is transformed so that it is turned on its side. The QL is taking its character information from your customised set. It is wise to remember, when editing characters, that unless you want customised lettering, it is often best to edit either upper or lower case letters only. Making too many changes to letters and numbers can result in a situation where you can no longer understand what the program is saying to you!

Module 2.4.13: Storing the character set

Having edited the character set, we now want to be able to keep it so that it may be used in future, otherwise the whole exercise is rather pointless. This module stores your customised characters on microdrive. The module is simpler than that employed in many other programs, since what we are saving is an easily identifiable block of memory and it can be sent to the microdrive with the simple use of SBYTES.

Module 2.4.13: Lines 22000 – 22090

```
22000 REMark ************************
22010 DEFine PROCedure store
22020 REMark ************************
22030   CLS
22040   AT 1,14 : PRINT "SAVE DATA"
22050   INPUT\\" Name of data file:";file$
22060   tfile$="mdv1_" & file$
22070   DELETE tfile$
22080   SBYTES "mdv1_" & file$,user_start,876
22090 END DEFine store
```

Testing

Having edited a few characters and placed them into memory, call up this module to save them on to microdrive. The only check at this moment is that the module executes without producing any kind of error. After the next module has been entered you will be able to reload the character set to check that it has been properly stored.

Module 2.4.14: Reloading a character set

Having stored the character set on disk, this module performs the task of reloading the character data, using LBYTES. Note that, in order for this

module to work, the value of USER_START must have been set. This is, of course, done automatically in the present program but, once you have designed a customised character set, you will want to be able to load it back into memory for the use of other programs. This module will do the job for you, provided that you have chosen the place in memory where the character data is to go, using RESPR.

Module 2.4.14: Lines 23000 – 23080

```
23000 REMark ************************
23010 DEFine PROCedure recall
23020 REMark ************************
23030   CLS
23040   AT 1,14 : PRINT "RECALL DATA"
23050   DIR mdv1_
23060   INPUT\\" Name of data file:";file$
23070   LBYTES "mdv1_" & file$,user_start
23080 END DEFine recall
```

Testing

You should now be able to reload the character set which you SAVEd as part of the test of the previous module by pressing 'L' in the character edit mode. Before you reload what you have just saved, do ensure that you are back with the normal (unedited) character set or it will be impossible to tell whether different characters have been loaded from disk.

Module 2.4.15: The control module

A standard module.

Module 2.4.15: Lines 10000 – 10090

```
10000 REMark ************************
10010 DEFine PROCedure do (fresh)
10020 REMark ************************
10030   CLS : CLS#0
10040   OVER 0
10050   IF fresh=1 THEN initialise
10060   POKE_L char_reg,user_start
10070   grid
10080   STOP
10090 END DEFine do
```

Testing

The whole program should now be available to you. In addition, if everything works satisfactorily here, you will know that you are safe to lift modules out to enable you to employ customised characters in other programs.

PROGRAM 2.5: SOUND DEMO

Program function

Sound Demo, consisting of two modules, can hardly be dignified with the name of a program. It is a tool, designed to allow you to explore the QL's rather confusing set of sound commands and to get the most from them. I have refrained from giving advice as to the way the sound parameters can be be set for the simple reason that, like the writers of the QL manual who simply advise experiment, I have found them not to be consistent in their effects. With Sound Demo, however, you will at least be able to experiment with ease, rather than continually enter the relatively complex BEEP command.

Figure 2.6: Screen Display from Sound Demo.

Module 2.5.1: Displaying parameters

The module is designed to clear an area of screen towards the bottom and to print out a clear display of the current settings of the various sound parameters. The variables on which it functions will be explained during the course of the commentary on the next module.

Module 2.5.1: Lines 2000 – 2100

```
2000 REMark ************************
2010 DEFine PROCedure parameters
2020 REMark ************************
2030   BLOCK 448,45,0,155,4
2040   PAPER 4 : INK 0
2050   AT 16,0
2060   PRINT,"Pitch1=";tp1,"Pitch2=";tp2
2070   PRINT,"Grad X=";tg1,"Grad Y=";tg2
```

```
2080    PRINT,"Wrap = ";twr,"Fuzz = ";tfu
2090    PRINT,"       Random=";tra
2100 END DEFine parameters
```

Module 2.5.2: A menu for experimenting

This is the main module, which is designed to present the user with a range of choices as to the note characteristics to be changed and to allow the sounding of the results.

Module 2.5.2: Lines 1000 – 1610

```
1000 REMark ************************
1010 DEFine PROCedure demo
1020 REMark ************************
1030   p1=5 : p2=5 : g1=0 : g2=0
1040   wr=0 : fu=0 : ra=0
1050   tp1=p1 : tp2=p2
1060   tg1=g1 : tg2=g2
1070   twr=wr : tfu=fu : tra=ra
1080   PAPER 2 : CLS
1100   PAPER 0 : INK 7
1120   AT 0,9 : PRINT "SOUND DEMONSTRATION"
1130   PAPER 2
1140   PRINT\"COMMANDS AVAILABLE:"
1150   PRINT\"    0) Stop demo"
1160   PRINT "    1) Sound note"
1170   PRINT "    2) Alter pitch 1"
1180   PRINT "    3) Alter pitch 2"
1190   PRINT "    4) Alter gradient X"
1200   PRINT "    5) Alter gradient Y"
1210   PRINT "    6) Alter wrap"
1220   PRINT "    7) Alter fuzz"
1230   PRINT "    8) Alter random"
1240   PRINT "    9) Stop note"
1250   REPeat response
1260     CLS#0
1270     parameters
1280     REPeat number
1290       q$=INKEY$(-1)
1300       IF q$>="0" AND q$<=9 THEN EXIT number
1310     END REPeat number
1320     q=q$
1330     SELect ON q
1340       ON q=0
1350         BEEP
1360         STOP
1370       ON q=1
1380         BEEP 0,p1,p2,g1,g2,wr,fu,ra
1390         tp1=p1 : tp2=p2
1400         tg1=g1 : tg2=g2
1410         twr=wr : tfu=fu
1420         tra=ra
1430       ON q=2
```



```
1440         INPUT#0,"PITCH 1: ";p1
1450       ON q=3
1460         INPUT#0,"PITCH 2: ";p2
1470       ON q=4
1480         INPUT#0,"GRADIENT X: ";g1
1490       ON q=5
1500         INPUT#0,"GRADIENT Y: ";g2
1510       ON q=6
1520         INPUT#0,"WRAP (1-15): ";wr
1530       ON q=7
1540         INPUT#0,"FUZZ (0-15):";fu
1550       ON q=8
1560         INPUT#0,"RAND (-32768 to 32767): ";
             ra
1570       ON q=9
1580         BEEP
1590     END SELect
1600   END REPeat response
1610 END DEFine demo
```

Commentary

Lines 1030 – 1040: The initial settings for the demonstration. P1 and P2 are the two pitches. G1, G2, WR, FU and RA refer to GRADIENTS 1 and 2, WRAP, FUZZ and RANDOM.

Lines 1050 – 1070: The variables beginning with T represent temporary copies of the active parameter settings.

Lines 1080 – 1240: There is only one main screen for the demonstration. This menu remains in the centre of the screen at all times.

Lines 1280 – 1310: Since there are only 10 choices on the menu, they can all be accessed with a single key press, using INKEY$.

Lines 1340 – 1360: Pressing '0' results in the current note being silenced and the demonstration terminating. If you stop the program using CTRL/SPACE the current note will continue to sound until you enter BEEP.

Lines 1370 – 1420: Pressing '1' results in the note defined by the current parameters being sounded. The temporary copies are made at this point because the other program functions, which allow the note parameters to be changed, do not immediately affect the note being sounded. It is possible to change every parameter without changing the sound of the note until '1' is pressed again. The temporary variables record the parameters actually being played, while the main variables may have been changed.

Lines 1430 – 1560: Pressing '2' to '8' gives the user the opportunity to change one of the parameters.

Lines 1570 – 1580: Pressing '9' silences the current note without terminating the program.

Testing

The only way to test the program is to play with it by entering 'demo' and messing around with parameters. In practice, I renumber Sound so that it will fit in with the next program, Music. This allows me to try out different combinations of parameters before they are entered into a tune to be played.

PROGRAM 2.6: MUSIC

Program function

We have already noted, in the comments on the last program, that the QL's sound commands are complex and a little confusing. The purpose of Music is to take the effort out of the programming of simple tunes by allowing the user to specify musical notation in an easily understood format, making full use of the various controls possible over the notes produced. The program adopts a 'two pass' approach, first processing the tune specified and translating it into a series of numbers, then using the numbers themselves to activate the BEEP command. The advantage of this method is that music can be played more quickly, without gaps creeping in when large numbers of short notes are to be played.

Module 2.6.1: A table of note values

Musically, microcomputers can be divided into two categories: those whose sound commands are set up so that they can be programmed in a comprehensible way, and those that aren't. Unfortunately, the QL falls into the latter group so, instead of being able to specify a note, either as a number or in some kind of musical notation, a value bearing a vague (and inverse) relation to frequency has to be used. If we are going to program music, then we need to be able to specify notes. The table contained in these lines of data represents my best attempt to translate two chromatic octaves into the values used by the BEEP command. The two octaves are in C major and represent about as far as the QL can usefully go.

Module 2.6.1: Lines 5000 – 5050

```
5000 REMark ************************
5010 REMark note values
5020 REMark ************************
5030    DATA 78,73,69,64,60,56,53,49
5040    DATA 42,40,37,34,34,32,30,28
5050    DATA 26,24,22,20,17,16,14,13
```


Module 2.6.2: Initialisation

A standard initialisation module.

Module 2.6.2: Lines 2000 – 2100

```
2000 REMark ************************
2010 DEFine PROCedure initialise
2020 REMark ************************
2030   DIM notes%(1,11),play%(1000,7),temp(7)
2040   RESTORE 5000
2050   FOR i=0 TO 1
2060     FOR j=0 TO 11
2070       READ notes%(i,j)
2080     NEXT j
2090   NEXT i
2100 END DEFine initialise
```

Commentary

Line 2030: NOTES% will be used to store the values of the 24 notes the program can deal with. PLAY% will contain the tune in its final form, up to to 1000 notes. TEMP will hold the values associated with a particular note as it is translated.

Lines 2040 – 2090: The 24 note values are read into the two sides on NOTES%, representing the two octaves.

Module 2.6.3: The data for the tune

Before the program can begin to process anything, it must have a tune to work on. Tunes are entered in the form of DATA statements, for ease of examination and editing. The notation will be explained during the course of the commentary on the next module. Note that, due to the QL's limitation in only being able to READ strings enclosed in quotes, the whole of each DATA line is enclosed in quotes. It would be simpler from the programming point of view to enclose each item separately but would make the entry of the tune itself far more laborious.

Module 2.6.3: Lines 6000 – 6100

```
6000 REMark ************************
6010 REMark data for tune
6020 REMark ************************
6030 DATA "O2,L3,1,L1,1,L4,3,L4,1,6"
6040 DATA "L8,5"
6050 DATA "O2,L3,1,L1,1,L4,3,L4,1,8"
6060 DATA "L8,6"
6070 DATA "L3,1,L1,1,L4,12,9,6,5,3"
6080 DATA "L3,10,L1,10,L4,9,6,8"
6090 DATA "L8,6"
6100 DATA "end"
```

Module 2.6.4: Translating the tune

This is the module which performs the most laborious part of the task of playing a tune, the translation from the easily-understood format, in which the music is recorded in the DATA statements, into a series of numbers which represent parameters for the BEEP command. The module looks complicated but, once analysed, it boils down to a series of simple decisions as to what a particular set of characters in the tune DATA means and then one or more calculations to translate that instruction into a BEEP parameter.

Module 2.6.4: Lines 3000 – 3700

```
3000 REMark ************************
3010 DEFine PROCedure process
3020 REMark ************************
3030   RESTORE 6000
3040   count=0
3050   pitch_number=1
3060   octave=1
3070   next_note=0
3080   REPeat get_note
3090     READ temp$
3100     IF temp$="end" OR temp$="END" THEN EXIT
         get_note
3110     :
3120     :
3130     REPeat slice
3140       comma="," INSTR temp$
3150       IF comma=0
3160         chars=LEN(temp$)
3170       ELSE
3180         chars=comma-1
3190       END IF
3200       t$=temp$(1 TO chars)
3210       IF chars<LEN(temp$)
3220         temp$=temp$(chars+2 TO)
3230       END IF
3240       IF "0" & t$>0
3250         temp(pitch_number-1)=notes%(octave,
             t$-1)
3260         next_note=1
3270       END IF
3280       IF t$(1)="O" OR t$(1)="o"
3290         octave=t$(2)-1
3300       END IF
3310       IF t$(1)="L" OR t$(1)="l"
3320         temp(7)=t$(2 TO)
3330       END IF
3340       IF t$="W" OR t$="w"
3350         temp(2)=t$(2 TO)
3360       END IF
3370       IF t$(1)="X" OR t$(1)="x"
3380         temp(3)=t$(2 TO)
```



```
3390       END IF
3400       IF t$(1)="Y" OR t$(1)="y"
3410         temp(4)=t$(2 TO)
3420       END IF
3430       IF t$(1)="F" OR t$(1)="f"
3440         temp(5)=t$(2 TO)
3450       END IF
3460       IF t$(1)="R" OR t$(1)="r"
3470         temp(6)=t$(2 TO)
3480       END IF
3490       IF t$(1)="P" OR t$(1)="p"
3500         pitch_number=t$(2)
3510       END IF
3520       IF t$="S" OR t$="s"
3530         temp(0)="-1"
3540         next_note=1
3550       END IF
3560       :
3570       :
3580       IF next_note
3590         FOR i=0 TO 7
3600           play%(count,i)=temp(i)
3610         NEXT i
3620         count=count+1
3630         next_note=0
3640       END IF
3650       :
3660       :
3670       IF comma=0 THEN EXIT slice
3680     END REPeat slice
3690   END REPeat get_note
3700 END DEFine process
```

Commentary

Line 3040: The variable COUNT will be used to record the number of the note currently being worked on. This is not the same as the number of the item of DATA being read. If the tune DATA specifies a series of changes to the note quality, such as GRAD and WRAP, these do not affect the value of COUNT. It is only when the note value is encountered that COUNT is incremented.

Line 3050: The number of the pitch which will be changed for each note — for the purposes of the program, the two pitches are numbered as 1 and 2.

Line 3060: The octave in which a note to be played will fall. The two octaves are specified as 1 or 2.

Line 3070: NEXT_NOTE is used to decide whether COUNT needs to be incremented. If, after translation of an item from the tune, NEXT_NOTE is set to 1, then a note has been specified and COUNT is incremented.

Lines 3090 – 3100: The whole of a DATA line is READ into TEMP$. The single word 'END' on a DATA line indicates the end of the tune.

Lines 3140 – 3230: Using the INSTR function, the program looks for the first comma in TEMP$, indicating the end of the first tune item. If no comma is found then the whole of TEMP$ is taken as the item to be processed. If a comma *is* found, the characters up to the comma are transferred to T$, which is the item to be processed and that item is sliced out of TEMP$ so that it will not be processed twice.

Lines 3240 – 3270: If the tune item is a number, then placing a zero in front of it will not change its value — if it is not a number then the zero will prevent the program from crashing when we try to extract a value from it. When a number is encountered, it is taken as an instruction to play a note of the given value within the current octave. There are 12 notes within each octave, each placed a semitone apart — these provide all the notes used with an octave in western forms of music. The note value is placed into TEMP%, from where it will be eventually transferred to ARRAY%.

Lines 3280 – 3300: If the first letter of the tune item is O, then the following number is taken as an instruction to reset the octave to either 1 or 2, though note that the program thinks of them as 0 and 1.

Lines 3310 – 3330: If the first character is L, then the following number is taken as the length of the notes to be played until further notice. The note length must be set up at the start of the tune.

Lines 3340 – 3480: Numbers preceded by W, X, Y, F or R will be used to set wrap, grad_x, grad_y, fuzz or random.

Lines 3490 – 3510: 0 or 1 preceded by P will change the pitch number which is currently being acted upon. Note that with the program as currently set up, you cannot change both pitch parameters between one note and the next.

Lines 3520 – 3550: If the item consists of S, then a silence of the current note length will be played.

Lines 3580 – 3640: The contents of ARRAY%(COUNT), the current note, are set equal to the contents of TEMP%. Note that this only happens when COUNT is incremented. All changes to parameters will be fed into the same line of ARRAY%, in other words changes to fuzz, wrap, etc., will all affect the current line, since COUNT is not incremented by them.

Line 3670: If COMMA is equal to zero then there are no more items in the current line of DATA and another line must be read.

Testing

Type:

initialise[ENTER]
process[ENTER]

and you should be able to complete the analysis of the tune without generating any errors. Of course you will not *hear* anything yet, since we have not added the module which actually plays the tune.

Module 2.6.5: Playing the tune

We now come to the module which makes it all worthwhile, since the current module has the job of taking the values stored in ARRAY% and playing a tune based on them.

Module 2.6.5: Lines 4000 – 4130

```
4000 REMark ************************
4010 DEFine PROCedure play (start)
4020 REMark ************************
4030   FOR i=start TO count-1
4040     IF play%(i,0)<>-1
4050       BEEP 0,play%(i,0),play%(i,1),play%
           (i,2),play%(i,3),play%(i,4),play%
           (i,5),play%(i,6)
4060     END IF
4070     FOR delay=1 TO 50*play%(i,7)
4080     NEXT delay
4090     BEEP
4100     t$=INKEY$(0)
4110     IF t$<>"" THEN RETurn
4120   NEXT i
4130 END DEFine play
```

Commentary

Line 4010: This module is called by entering 'play' followed by a value which is taken as the first note in ARRAY% to be played. This allows parts of a tune to be heard during the process of tune development. If changes are made to a tune, these will not be registered by the PLAY module until PROCESS has first been called to translate the changed DATA.

Line 4030: The number of notes to be played has been automatically stored in COUNT.

Lines 4040 – 4060: The two possibilities dealt with by these lines are that a note is to be played or that there is to be a silence. If a note is to be played, a BEEP of infinite duration (BEEP 0) is commenced, using the parameters contained in ARRAY%. If the first parameter in ARRAY(I) is minus one, however, then a silence is called for and no note is played.

Lines 4070 – 4080: The length of the note or silence is dictated by the length of the delay loop. Using the loop variable to create the length of the loop allows more ease in specifying the note lengths than if the huge values necessary for BEEP were used directly.

Line 4090: At the end of the period specified for the delay, any note which is being sounded is switched off by use of BEEP without parameters attached.

Lines 4100 – 4110: The playing of the tune can be stopped at any point by pressing a key.

Testing

Provided that the program has already been initialised and the tune processed, entering PLAY 0 will play the tune 'Happy Birthday To You'. Entering PLAY followed by a parameter other than zero will start the tune off at another point. You can now try changing the DATA statements which make up the tune to see what difference they make.

Module 2.6.6: The control module

A standard module.

Module 2.6.6: Lines 1000 – 1060

```
1000 REMark ************************
1010 DEFine PROCedure do
1020 REMark ************************
1030   initialise
1040   process
1050   play 0
1060 END DEFine do
```

Using the program

You can use this program on its own or as a tool to supply music for other programs. In order to do this, all you need to do is to write microdrive STORE and RECALL modules to store the variable COUNT and the contents of ARRAY%. Once a tune has been developed and processed, store the information on microdrive. A subsequent program, which would have to include RECALL and PLAY modules, would then summon up the contents of ARRAY% from microdrive and play the tune exactly as the current program would.

CHAPTER 3

Seriouser and Seriouser

At this stage in your progress you should be becoming more familiar with the capabilities of your QL and with some of the techniques needed to put it to work for you. The time has come, therefore, to look at some substantial programs which will allow your QL to do what microcomputers do best — handle, sort and retrieve information for their owners.

In doing this we shall, in some ways, be re-inventing the wheel supplied by Psion in the form of Quill, Archive, Easel, and Abacus since as a package these four can cover most tasks imaginable. Why then write applications packages in BASIC — well there are two answers to that. First, and I think most importantly, is the question of who runs your QL, you or a group of software writers you've never met. This may seem a trivial question but the fact is that only you can decide whether you are going to run a system which responds exactly to *your* needs, taking the best from commercial programs and your own inventions, or whether you are going to limit yourself entirely to what someone else thinks that you need. Without some practice in developing and writing your own applications, you are going to be left with a black box which forces you to become just another consumer, not the creative originator that most micro owners want to be.

The second reason, however, is more immediately practical, and that is that there is no such thing as a perfect software package for every purpose. There will many occasions in the use of the QL when a simple BASIC package will serve you far better for a limited purpose than a sophisticated commercial package. The programs in this chapter are, I think, cases in point but even if they do not fit your needs, the techniques contained within them will allow you to tackle your own applications programming with greater confidence.

The programs included in this chapter are:

UNIFILE: A powerful personal filing system capable of storing a wide variety of information for instant recall.

NNUMBER: A program which creates a dictionary of *N*ames and *Numbers* for almost anything you wish, allowing the user to create invoices, stock valuations or even a calorie count of the day's menu.

MULTIQ: A multiple choice test generator.

PROGRAM 3.1: UNIFILE

Program function

Unifile is a program which has been developed over the years in the 'Working Micro' books. Readers of previous books have written to say that they are using it in their businesses, to teach schoolchildren about the way micros handle information, to help with clubs and voluntary organisations or simply to keep track of their books and records at home. This current version for the QL uses some fairly sophisticated and extremely fast data structures, to make it an ideal card-index type tool for quantities of information capable of being held in the QL's memory at one time.

```
Entry No: 1
Name: Smith

Commands available:

   'ENTER' leaves field unchanged
    Input item to replace one shown
    'DDD' deletes whole record
    'ZZZ' leaves record unchanged
Which do you require:
```

Figure 3.1: Typical Display in Unifile's Search Mode.

The new concepts introduced in this program include:

1) Binary searching.
2) Packing items into continuous strings.
3) Pointer arrays

Module 3.1.1: Setting up the structure of the file

Many books aimed at the home micro owner offer inferior filing programs which are extremely inflexible in use. It is built into the program that, every time the user stores something, it will be under the headings, Name, Address, Phone No., or some similar structure. The beauty of Unifile is that, while it will certainly allow you to use such a structure, it will also let you create other files with very different structures — perhaps with one heading, perhaps with 10 — without making any changes to the program itself. Unifile is what I like to call a chameleon program: one that adapts to

a variety of different uses, reacting to the user in different ways depending on the task that it is performing at the time.

The purpose of this first module is to initialise some variables, and to allow data to be recalled from microdrive, but most importantly to allow the user to set up the original file exactly as desired.

Module 3.1.1: Lines 10000 – 10260

```
10000 REMark ************************
10010 REMark initialise
10020 REMark ************************
10030   PAPER 5 : INK 1
10040   CLS : CLS#0
10050   AT 1,15 : PRINT "UNIFILE"
10060   INPUT\\"Load from Microdrive (y/n)";q$
10070   IF q$="y" OR q$="Y"
10080     recall
10090   ELSE
10100     f_total=0
10110     empty$=""
10120     INPUT\\"How many fields in a record: ";
          n_fields\\
10130     DIM fields(n_fields-1)
10140     DIM field$(n_fields-1,15)
10150     FOR i=0 TO n_fields-1
10160       INPUT\("Name for field " & (i+1) &
            ": ");field$(i)
10170       INPUT "Length of field: ";fields(i)
10180       f_total=f_total+fields(i)
10190     NEXT i
10200     lines=INT(50000/f_total)
10210     DIM array$(lines,f_total)
10220     DIM new_temp$(n_fields,38)
10230     array$(0)=FILL$(CHR$(200),38)
10240     array$(1)=FILL$("z",100)
10250     ptr$="00000001"
10260   END IF
```

Commentary

Lines 10120 – 10190: Part of the secret of Unifile's flexibility. For each 'record', which you can think of as a filing card if it helps you, you can define for yourself how many 'fields', or individual headings, will appear in the record. If you were storing your record collection, you might use headings like: TRACK, ALBUM, COMPOSER, ARTIST, LENGTH. In this case you would specify five fields and then input the names of the five. In future, whenever you use Unifile to store or retrieve information on your record collection, you will be asked to input an item of information under each of those headings. For each field, you will also be asked to specify the length of the field in characters — in future uses of the file, you will be limited to that number of characters when inputting information to

the particular field. The overall names given to each field, and their maximum lengths, are stored in the two arrays FIELD$ and FIELDS, which are dimensioned according to the specified number of fields, recorded in N_FIELDS.

Lines 10200 – 10210: Having defined the sizes of all the fields and therefore of the whole record, it is possible to determine how many such records will fit into memory. I have allocated 50,000 bytes of the total memory to the main data file, which is conservative. There are other variables, apart from the program itself, to be stored but you will probably be able to increase the 50,000 substantially for most applications. For files containing a large number of very short records, however, the associated variables will take up a proportionately larger part of the memory and you may even find that 50,000 is too much for the main file. Experimentation will quickly reveal the best figures for your particular usage.

Lines 10230 – 10240: In any program which inserts data into an array in order, some provision has to be made for the recognition of the beginning and end of the data when new items are being inserted. One simple method of achieving this is to set up the file with two dummy entries which, according to the order to be imposed, will always fall at the beginning and end of any sensible data. The use of a string of zs for the final item is self-explanatory, but the CHR$(200) in line 10230 needs some explanation. The problem encountered in setting up an entry which will automatically be seen as the first item by any alphabetical search is that, on early versions of the QL, the string comparison facilities are a total mess. According to the version of the QL on which the programs in this book were written, any normal printing character outside the ranges 0 – 9, A – Z and a – z is actually greater than any character inside those ranges. This means that a space (character code 32), which is normally treated as the lowest normal printing character when string comparisons are done, is considered greater than 'z', normally the highest normal printing character. You can verify for yourself whether this applies to your machine by typing:

```
print " "> "z"
```

If the result is zero, then you can ignore the rest of this part of the commentary and replace CHR$(200) with a space (CHR$(32)). If, however, the result printed is '1', then your machine suffers from the same limitation as mine and the space character cannot be used. CHR$(200) has been adopted to replace the space because it is one of a limited range of characters which are considered by the QL to be alphabetically less than the '0' character. I am less than happy about the compromise because CHR$(200) is a cursor control character and its use may have unintended side effects on future QL versions. It is to be hoped that the string comparison facilities will soon have been sorted out and the space character can assume its rightful role.

Module 3.1.2: The menu

A standard menu module.

Module 3.1.2: Lines 11000 – 11360

```
11000 REMark ************************
11010 REMark menu
11020 REMark ************************
11030   REPeat display
11040     PAPER 4 : INK 0 : CSIZE 1,1
11050     CLS : CLS#0
11060     AT 1,10 : PRINT "UNIFILE" : CSIZE 0,0
11070     PRINT\\" COMMANDS AVAILABLE: "
11080     PRINT\"    1) INPUT NEW ITEMS"
11090     PRINT "    2) SEARCH/DELETE"
11100     PRINT "    3) DATA FILES"
11110     PRINT "    4) STOP"
11120     INPUT\" WHICH DO YOU REQUIRE: ";Z
11130     IF (Z=2 OR Z=3) AND LEN(ptr$)=8
11140       PRINT "NO DATA YET"
11150       t$=INKEY$(-1)
11160       Z=0
11170     END IF
11180     SELect ON Z
11190       ON Z=1 : n_items
11210       ON Z=2 : user_search
11230       ON Z=3 : store
11310       ON Z=4
11320         CLS
11330         PRINT#0,"Filing system closed"
11340         STOP
11350     END SELect
11360   END REPeat display
```

Module 3.1.3: Storing data

As you begin to develop more complex programs, whether from this book or on your own, you will find it more and more desirable to enter the data file module as early as possible. The reason for this is that it is only possible to make proper tests of Unifile by entering fairly considerable quantities of data. Since you are bound to make mistakes and will need to change lines, you are faced with the prospect of having to re-enter data which may have been corrupted time and time again as the program is being developed. The answer is to store it on microdrive from the very earliest of stages, recalling information from the microdrive to make your tests if necessary.

The variables contained in these two modules will be explained during the course of the program.

Module 3.1.3: Lines 20000 – 20320

```
20000 REMark ************************
20010 DEFine PROCedure store
```

```
20020 REMark ************************
20030   CLS
20040   AT 1,14 : PRINT "SAVE DATA"
20050   INPUT\\" Name of data file:";file$
20060   tfile$="mdv1_" & file$
20070   DELETE tfile$
20080   OPEN_NEW #8,"mdv1_" & file$
20090   PRINT #8,LEN(ptr$)
20100   IF LEN(ptr$)<>0
20110     FOR i=1 TO LEN(ptr$) STEP 100
20120       PRINT #8,ptr$(i TO i+99)
20130     NEXT i
20140   END IF
20150   PRINT #8,LEN(empty$)
20160   IF LEN(empty$)<>0
20170     FOR i=1 TO LEN(empty$) STEP 100
20180       PRINT #8,empty$(i TO i+99)
20190     NEXT i
20200   END IF
20210   PRINT #8,f_total
20220   PRINT #8,lines
20230   PRINT #8,n_fields
20240   FOR i=0 TO n_fields-1
20250     PRINT #8,field$(i)
20260     PRINT #8,fields(i)
20270   NEXT i
20280   FOR i=1 TO (LEN(ptr$)+LEN(empty$))/4
20290     PRINT#8,array$(i-1)
20300   NEXT i
20310   CLOSE#8
20320 END DEFine store
```

Module 3.1.4: Recalling data

This is a standard data recall module with the necessary addition of facilities to dimension arrays. You will remember from the initialisation module that the arrays in Unifile are not of a fixed size, but vary according to the kind of structure set up by the user. The same approach is necessary when picking up items from microdrive. Only once the variables which describe the structure have been recalled, can the arrays be dimensioned.

Module 3.1.4: Lines 21000 – 21340

```
21000 REMark ************************
21010 DEFine PROCedure recall
21020 REMark ************************
21030   CLS
21040   ptr$="" : empty$=""
21050   AT 1,14 : PRINT "RECALL DATA"
21060   DIR mdv1_
21070   INPUT\\" Name of data file:";file$
21080   OPEN_IN #8,"mdv1_" & file$
21090   INPUT #8,chars
```



```
21100   IF chars<>0
21110     FOR i=1 TO chars STEP 100
21120       INPUT#8,temp$
21130       ptr$=ptr$ & temp$
21140     NEXT i
21150   END IF
21160   INPUT #8,chars
21170   IF chars<>0
21180     FOR i=1 TO chars STEP 100
21190       INPUT #8,temp$
21200       empty$=empty$ & temp$
21210     NEXT i
21220   END IF
21230   INPUT #8,f_total
21240   INPUT #8,lines
21250   INPUT #8,n_fields
21260   DIM array$(lines,f_total),fields
        (n_fields-1),field$(n_fields-1,15)
21270   FOR i=0 TO n_fields-1
21280     INPUT #8,field$(i),fields(i)
21290   NEXT i
21300   FOR i=1 TO (LEN(ptr$)+LEN(empty$))/4
21310     INPUT #8,array$(i-1)
21320   NEXT i
21330   CLOSE#8
21340 END DEFine recall
```

Module 3.1.5: Setting up a pointer

This module and the one which follows, though simple, are at the core of the method of data storage used by this program. The problem to be faced with all programs which hold large quantities of data, is how to insert new items into the file. It is perfectly possible to find the right place in an ordered file and then shift everything one place upwards to make room for the new item. The only problem with this is that it takes *time*. What we shall do in the case of this program is to take advantage of one very fast method of data manipulation available on the QL, string slicing. If we have a string of 1000 characters (OLD$), a new character (NEW$) can be inserted at position 500 by a simple line such as:

old$ = old$(1 to 499) & new$ & old$(500 to)

Clearly, this is going to be a much faster method, but how can it be adapted to the much larger masses of information involved in a filing program, which is far more easily dealt with by means of multi-dimensional arrays like ARRAY$, dimensioned in Module 3.1.1. The answer commonly adopted is to use what is known as a 'pointer array'.

The effect of a pointer array is to allow data to be inserted into the main file (ARRAY$) in no particular order, items simply being placed in the first empty space, with no need to move other items to make room. The *order* of

the data, in this case alphabetical order, is preserved by the fact that each entry has an associated pointer which records its correct position in the order. Take a look at the table given below:

ORDER OF DATA ITEMS	ORDER OF POINTERS
FFF	5
CCC	4
DDD	2
BBB	3
AAA	6
EEE	1

The data items on the left, which need to be recalled in alphabetical order, are clearly a jumble. If we turn to the value of the pointers, however, we can see that the correct place of each, in terms of alphabetical order, is safely recorded, with the first pointer indicating the position of the first item in alphabetical order (AAA), the second pointer indicating the position of the second item in alphabetical order (BBB), and so on. To read through the apparently jumbled file in perfect order, all we have to do is to read the items indicated by the pointers.

What this module is going to accomplish is to create a four-character pointer which a later module will place in the correct position within a single long string, PTR$ (PoinTeR$) The next module will retrieve the value of a four-character pointer from a particular position within PTR$. How the pointers will be used in detail will only become apparent during the commentary on the rest of the program.

Module 3.1.5: Lines 19000 – 19060

```
19000 REMark ************************
19010 DEFine FuNction four$ (ptr)
19020 REMark ************************
19030   t$=ptr
19040   t$=FILL$("0",4-LEN(t$)) & t$
19050   RETurn t$
19060 END DEFine four$
```

Testing

Type:

print four$(1) [ENTER]

and the result should be '0001'. You should be able to enter any number up to 9999 and see it transformed into a four-character string.

Module 3.1.6: Retrieving a pointer

Having given ourselves the ability to create a pointer which can be inserted into a string, we turn our attention to the problem of getting a value out of the same string. This simple module accepts an argument in the form of the variable SS and slices out a four-character pointer from PTR$.

Module 3.1.6: Lines 18000 – 18060

```
18000 REMark ************************
18010 DEFine FuNction p_val (ss)
18020 REMark ************************
18030   t_ptr=ss*4
18040   t_ptr=ptr$(t_ptr+1 TO t_ptr+4)
18050   RETurn t_ptr
18060 END DEFine p_val
```

Testing

Type:

ptr$ = "000300020001"[ENTER]
print p_val(2)[ENTER]

The result should be '1', the value of the third four-character pointer in the string.

Module 3.1.7: A better way of searching

In this module, and the two that follow, we take a look at how a new entry is added to the main file contained in ARRAY$, or rather how the correct position for its pointer is found in the string PTR$. The core of the method is contained here, however, because it is this module which allows Unifile to search rapidly through a large file of entries to find the correct place to insert a new one, or to conduct a fast search for the presence of a key entry in the file.

The method is known as 'binary searching' and it can be used to dramatically reduce the time for searching in any programs you write which hold long lists of ordered data. Consider the following example.

A file has been established containing 2000 names and the task is to insert a new name into the file, in the correct alphabetical order. If we cheat and look at the list of names, we can determine that the new name 'YOUNGER' should actually go into the file at position 1731, though the computer has no way of knowing this in advance.

One thing we could do is to set the computer examining the names one by one from the beginning. It will begin with 'ADAMS' and note that 'YOUNGER' should come after, then go on to 'ADAMSON' and so on. Eventually, after examining 1732 names, the search will hit upon a name

like 'YOUNGMAN' which must come *after* 'YOUNGER', so the correct position has been found.

This is a reliable method, but how much better if the number of comparisons made could be cut down a little. Well, in the case of our file of 2000 names, the whole process can be accomplished by just *10 comparisons*. Here's how it's done.

The computer begins the search by examining the name in position 1024 of the file, because 1024 is the greatest power of 2 (2^10) which will fit into the total number of entries (2000). The name at position 1024 is found to be alphabetically less than 'YOUNGER' and so the computer adds 1024/2, or 512, or 2^9 to the original search position, arriving at 1536. Once again, the name at this position is alphabetically less than 'YOUNGER' so this time 256, or 2^8 is added to 1536, making 1792. Now something different happen because the name at position 1792 is *later* in the alphabetical order than 'YOUNGER' so instead of adding 128, or 2^7, it is *subtracted* from the search position, giving 1664.

The search goes on, adding or subtracting decreasing powers of two to build a search pattern that looks like this:

COMPARISON NO.	POSITION	ACTION
1	1024	+512
2	1536	+256
3	1792	−128
4	1644	+64
5	1728	+32
6	1760	−16
7	1744	−8
8	1736	−4
9	1732	−2
10	1730	+1

Try it yourself for different numbers of entries and different target positions in the order — you will find that it always works.

Module 3.1.7: Lines 13000 – 13160

```
13000 REMark ************************
13010 DEFine PROCedure binary_search (t1$)
13020 REMark ************************
13030   po=INT(LN(LEN(ptr$)/4)/LN(2))
13040   ss=2^po-1
13050   FOR i=po TO 0 STEP -1
13060     IF array$(p_val(ss))<t1$
13070       ss=ss+2^i
13080       IF ss>LEN(ptr$)/4-1 THEN ss=LEN
            (ptr$)/4-1
13090     END IF
13100     IF array$(p_val(ss))>t1$
```



```
13110       ss=ss-2^i
13120       IF ss<0 THEN ss=0
13130     END IF
13140   NEXT i
13150   IF array$(p_val(ss))<t1$ THEN ss=ss+1
13160 END DEFine binary_search
```

Commentary

Line 13030 – 13040: These two expressions find the greatest power of 2 which will fit into the number of items in the file and then set the search pointer (SS) equal to that number. The total number of items is recorded by the length of PTR$ divided by four, since each pointer contains four characters. The '−1' in the second expression takes account of the fact that the array is numbered from zero, not one.

Lines 13050 – 13140: This loop conducts the search, using reducing powers of 2. The main file is contained in ARRAY$ and the new entry in T1$. Note that, as you would expect from previous explanations of the use of pointer arrays, the search pointer does *not* scan through the main array but through PTR$, using the P__VAL function. If the item to be compared with the new entry is item number 500, it is the value of the pointer at position 500 which dictates which entry in ARRAY$ is to be compared with the new entry.

Line 13150: In some cases the position arrived at will be 1 below the correct position — in this case SS is increased by 1.

Module 3.1.8: Inserting an item

This module inserts the new entry into the first available empty space in ARRAY$, then records that position in PTR$ at a place indicated by the variable SS.

Module 3.1.8: Lines 14000 – 14150

```
14000 REMark ************************
14010 DEFine PROCedure insert (t1$)
14020 REMark ************************
14030   IF empty$<>""
14040     place=empty$(1 TO 4)
14050     IF LEN(empty$)>4
14060       empty$=empty$(5 TO)
14070     ELSE
14080       empty$=""
14090     END IF
14100   ELSE
14110     place=LEN(ptr$)/4
14120   END IF
```

```
14130    ptr$=ptr$(1 TO ss*4) & four$(place) &
         ptr$(ss*4+1 TO)
14140    array$(place)=t1$
14150 END DEFine insert
```

Commentary

Lines 14030 – 14100: In order to explain what is going on here, I'm afraid we have to jump ahead of ourselves a little to consider how items are going to be deleted from the file.

We have already said that one of the main advantages of using a pointer string is that we do not have to shift all the data when a new item is being entered — all that has to be moved is the pointers to the data, while the data itself can simply be placed in the first empty space. But where is the first empty space? To begin with, the answer to that question will be simple. As each new item is entered, it can simply be placed at the end of the file. If three items have already been entered into positions 0 to 2, the fourth item is put into position 3. That does not, however, solve the problem of deletions from the file. When an item is deleted, it will leave a hole in the array and, eventually, as the program is used, the whole of the array will consist of such holes, with no space left at the end for new items. Clearly, the holes have to be filled.

We could retreat to the cruder method of shifting everything down to cover the hole — a straightforward method used by other programs in the book — but that seems to defeat the object of using a pointer array, especially since all the pointers for the moved data would have to be changed to reflect the move.

Much simpler, and much quicker, is to work out a method which allows the positions of holes to be recorded and to use them for new items of data, since we already know that it does not matter in the least where a new item is placed, provided that its pointer is in the right place. This recording of holes is the job of EMPTY$, which is created by the deletion module you will enter later. EMPTY$ is the same in structure as PTR$, consisting of four-character pointers, but this time the positions recorded are those of empty slots in the array.

With this explanation under your belt, you can see that these lines are quite simple. If there is an empty slot in the main part of the array (indicated by there being a pointer in EMPTY$) then the address of the slot is obtained and that is where the new item is placed, the address being sliced out of EMPTY$. If EMPTY$ contains no record of holes in the array, then the new item is simply placed above the existing data.

Line 14130: We have already obtained the correct position of the new item in terms of alphabetical order and it is contained in the variable SS. A new pointer is added to PTR$ at position SS, its contents being the address in ARRAY$ of the new data item.

Module 3.1.9: Making entries to the file

Having entered the modules which do the real work, we can now proceed to the one the user will have contact with when placing material into the filing system. The function of this module is to prompt the user to input the correct number of items, in the correct order, for each entry, to combine those items into a single string which will fit into one line of ARRAY$ and then to call up the two previous modules to insert the entry into the main file.

Module 3.1.9: Lines 12000 – 12300

```
12000 REMark ************************
12010 DEFine PROCedure n_items
12020 REMark ************************
12030   REPeat loop
12040     REPeat confirm
12050       t1$=""
12060       CLS
12070       AT 1,14 : PRINT "NEW ITEMS"
12080       PRINT\\"Input item or 'zzz' to
            quit."\\
12090       FOR i=0 TO n_fields-1
12100         CLS#0
12110         PRINT #0,field$(i);": "
12120         PRINT #0,\FILL$("=",fields(i))
12130         AT #0,1,0
12140         INPUT #0,new_temp$(i)
12150         IF new_temp$(i)='zzz' OR new_temp$
              (i)="ZZZ" THEN RETurn
12160         new_temp$(i)=new_temp$(i) & FILL$
              (CHR$(200),38)
12170         CLS #0
12180         PRINT field$(i);": ";
12190         PAPER 6 : INK 0
12200         PRINT new_temp$(i,1 TO fields(i))
12210         PAPER 5 : INK 1
12220         t1$=t1$ & new_temp$(i,1 TO fields
              (i))
12230       NEXT i
12240       INPUT \\"ARE THESE CORRECT (Y/N): ";q$
12250       IF q$="y" OR q$="Y" THEN EXIT confirm
12260     END REPeat confirm
12270     binary_search (t1$)
12280     insert (t1$)
12290   END REPeat loop
12300 END DEFine n_items
```

Commentary

Lines 12090 – 12230: These are the lines which request the input of the individual items for the new entry. The name for each item is taken from the array FIELD$, which was set up in the initialisation module. Each item

is input into the array NEW__TEMP$, and padded out with character code 200 to a standard length of 38. As an aid to the user, the inputs are done at the bottom of the screen and the maximum length of the individual field to be entered is indicated by a line of '=' signs underneath what is to be input. Note that, as in the initialisation module, I am not entirely happy about the use of CHR$(200) for the purposes of padding. The problem once again is that we need to add something to the end of the string which will not make a nonsense of its position in an alphabetically-arranged file. When I started to write the program, each new entry was padded to a standard length with spaces but this created the ludicrous situation that an entry of 'A' as the first field of a record, was placed *after* an entry of 'AZ', since the space after the 'A' was assessed as being greater than the letter 'Z'. If, by the time you read this, your QL is capable of recognising that a space is alphabetically less than a letter or digit, CHR$(200) can and should be replaced by a space.

The last line of the loop accumulates the individual fields into a single string, T1$, which, as we have already seen, is the string used for the binary search. Each field is chopped to precisely the right length, as recorded by the array FIELDS, which was set up in the initialisation module.

Testing

You are now in a position to test the last three modules that you have entered. Run the program and set up a file with two fields per record, called 'ONE' and 'TWO', each three characters long. When you have finished initialising the program and you get to the main menu, specify option 1. You will now be faced with the display created by the current module and asked to input item 'ONE'. Enter 'AA1'. The prompt will repeat for 'TWO'and you should enter 'AA2'. Repeat the process for 'DD1', 'DD2', 'CC1', 'CC2', 'BB1', 'BB2' to successive prompts. Now enter 'ZZZ' and you will return to the main menu. Select option 4 to stop the program. Now enter:

for i = 0 to 3 : print array$(p__val(i)):next i[ENTER]

and you should see:

AA1AA2
BB1BB2
CC1CC2
DD1DD2

If all has worked correctly, you might like to start the program again with GOTO 11000 and use menu option 3 to store the data you have input on microdrive. This will make subsequent tests less onerous if the data should become corrupted in some way.

Module 3.1.10: Searching for items in the file

We can now proceed to the module which makes the program *useful* by allowing the data which has been stored to be retrieved. The current module will allow entries to be retrieved by one of four methods:

1) One by one in order from the current position.
2) By jumping forwards or backwards a specified number of items.
3) By entering a key item — the first item in the entry — for a fast search.
4) By searching for any occurrence of a combination of characters, wherever it is within an entry.

Module 3.1.10: Lines 15000 – 15660

```
15000 REMark ************************
15010 DEFine PROCedure user_search
15020 REMark ************************
15030   s1=1
15040   CLS
15050   IF LEN(ptr$)=8
15060     PRINT\\"No data yet!"
15070     t$=INKEY$(-1)
15080     RETurn
15090   END IF
15100   PRINT\\"Commands available:"
15110   PRINT\" > Input item for normal search"
15120   PRINT " > '*' first for initial search"
15130   PRINT " > Return for first item on file"
15140   PRINT\"********************************
        ***"
15150   INPUT "Search command: ";target$
15160   IF CODE(target$)=42
15170     target$=target$(2 TO)
15180     binary_search (target$)
15190     s1=ss
15200     target$=""
15210   END IF
15220   REPeat main_search
15230     IF target$<>""
15240       found=0
15250       FOR i=s1 TO LEN(ptr$)/4-1
15260         found=target$ INSTR array$(p_val(i))
15270         IF found<>0
15280           s1=i
15290           EXIT i
15300         END IF
15310       NEXT i
15320       RETurn
15330       END FOR i
15340     END IF
15350     REPeat print_loop
15360       IF s1>LEN(ptr$)/4-2 THEN s1=LEN
          (ptr$)/4-2
15370       IF s1<1 THEN s1=1
```

```
15380       CLS
15390       PRINT "Entry No: ";s1\\
15400       start=1
15410       FOR i=0 TO n_fields-1
15420         PRINT field$(i);
15430         PAPER 6 : INK 0
15440         PRINT array$(p_val(s1),start TO
              start+fields(i)-1)
15450         PAPER 5 : INK 1
15460         start=start+fields(i)
15470       NEXT i
15480       PRINT#0," > 'ENTER' = next     'AAA'
            = amend"
15490       PRINT#0," > 'CCC' = continue   '#nn'
            = move"
15500       PRINT#0," > 'ZZZ' = quit"
15510       INPUT#0,"Which do you require: ";q$
15520       CLS#0
15530       IF q$="" THEN q$="#1"
15540       IF q$="zzz" OR q$="ZZZ" THEN RETurn
15550       IF q$="ccc" OR q$="CCC"
15560         s1=s1+1
15570         EXIT print_loop
15580       END IF
15590       IF q$="aaa" OR q$="AAA"
15600         amend
15610         q$="#0"
15620       END IF
15630       IF q$(1)="#" THEN s1=s1+q$(2 TO)
15640     END REPeat print_loop
15650   END REPeat main_search
15660 END DEFine user_search
```

Commentary

Line 15030: S1 is the pointer to the current entry, and starts at 1 every time a search is begun. In normal circumstances it would start at 0, but the first record in the main file is a dummy entry.

Lines 15050 – 15090: An error message is generated if no items have yet been placed into the file.

Lines 15100 – 15130: This is the start-up menu for the search module. On each search you will see it only once, when you begin. Using this menu, the type of search to be made is specified — if you wish to change the search type you will need to quit the current search and start again with this menu. The term 'NORMAL SEARCH' indicates a search for a given combination of characters — the first record returned will be the first one in the file which contains those characters, in any position. 'INITIAL SEARCH' refers to a search for a record which *begins with* the combination of characters specified by the user. Thus, inputting '*SMI' would find any record beginning with the letters SMI, though not always the first one. If the string

specified is not present at the beginning of any record, the record returned will be the one occupying the place in which it would be inserted if input as a new entry.

On a normal search, if the specified string is not found in any of the records in the file, program execution will return to the main program menu.

Lines 15160 – 15210: This line does all the work necessary for an 'initial search', by stripping off the leading asterisk and simply calling up the binary search module to find the correct position.

Lines 15220 – 15650: The main loop which will repeat a normal search if the user so requests at a later menu. Note that the initial search routine does not fall in this loop since there is no point in repeating an initial search — the result will always be the same record.

Lines 15230 – 15340: By this point in the execution of the module, any input must be a string to be searched for using the normal search. This is done quite simply by scanning records using INSTR. If an item is found, the variable FOUND is set to its position within the record, the position is recorded in S1 and EXIT takes program execution out of the loop. If the end of the loop is reached with FOUND still equal to zero, the search is terminated and the program returns to the main menu.

Lines 15350 – 15640: This loop will continue the secondary form of the search, where the user can continue a normal search or page through the items in the file, until the user specifies that the search is to be terminated.

Lines 15360 – 15370: Within the loop the user has the option to move around the file by number — these lines check on subsequent passes through the loop that the pointer has not been moved outside the valid range for the current number of records.

Lines 15410 – 15470: These lines print out the record whose position (in the pointer array) is indicated by the search pointer S1.

Lines 15480 – 15510: The secondary menu which appears once a record is displayed. The user has the option to move on to the next entry, to call up the AMEND function (not yet entered), to search further for the previously specified string, to move through the file a specified number of records, or to return to the main program menu.

Line 15530: The '#' symbol will be used as an indication that the user wishes to move a specific number of places forwards or backwards through the file. If ENTER is pressed on its own, Q$ is set equal to '#1', which later lines will use to move the search pointer on to the next record.

Lines 15550 – 15620: Entering 'CCC' results in the continuation of a normal search for the current TARGET$.

Lines 15590 – 15620: The AMEND function, which has not yet been entered. Note that deletions are made in the AMEND mode, and, if this results in there being nothing left in the main array but the two dummy records, the search module is terminated.

Line 15630: Inputting a number preceded by the '#' sign allows the user to move forwards or backwards through the file by altering the search pointer S1.

Testing

If you have previously saved the series of four entries made in previous tests, run the program and reload the entries. Specify option 2 on the main menu and then, when the SEARCH menu appears, page through the records by pressing RETURN. Each record should be displayed on two lines, with the appropriate item name, eg:

ONE: AA1
TWO: AA2

When you reach the fourth record you should find that you can go no further using RETURN. Now enter '# – 1' and you should move back to record 3. Continue to move back — you should find that you cannot move off the beginning of the file either.

Enter 'ZZZ' to return to the main menu and specify option 2 again. This time, respond to the initial SEARCH menu with 'CC'. Record 3 should be displayed — the only one to contain the characters 'CC'. You are now on the second SEARCH menu, so enter 'CCC' to continue the search. You should find yourself back at the main menu since there are no more records which contain the specified characters.

Once again, choose option 2, and this time enter '2' as your search target. Entry 1 should appear, since it contains the character '2'. Enter 'CCC' to continue the search and entry 2 should be printed — it also contains the character 2. Continue to enter 'CCC' until all record entries have been displayed and the search fails, returning you to the main menu.

Finally, choose option 2 from the main menu and enter '*B' as your search target. Entry 2 should be displayed — the only record to begin with 'B'. Enter 'ZZZ' to return to the main menu and terminate the program.

You have now tested all the search functions.

Module 3.1.11: Deleting entries

The final touches to the program are added by the following two modules, which allow entries to be changed or deleted. The deletion module is added first, since it is used whenever an entry is changed. The overall method involved in deletion has already been described in the commentary on the

insertion module for this program. The function of the module is threefold, to clear an individual record, to store the address of the resulting hole in EMPTY$, and to remove the appropriate pointer from PTR$.

Module 3.1.11: Lines 17000 – 17060

```
17000 REMark ************************
17010 DEFine PROCedure remove
17020 REMark ************************
17030   array$(p_val(s1))=""
17040   empty$=empty$ & four$(p_val(s1))
17050   ptr$=ptr$(1 TO s1*4) & ptr$((s1+1)*4+1 TO)
17060 END DEFine remove
```

Commentary

Lines 17040 – 17050: Deletion is simply accomplished by slicing the relevant pointer out of PTR$ and recording a hole in EMPTY$.

Testing

Run the program and reload the four items from microdrive. Specify option 4 to stop the program. Now enter:

s1 = 1[ENTER]
remove[ENTER]

When the flashing cursor returns, type:

goto 11000[ENTER]

and then call up the SEARCH option. You should find that the AA1/AA2 entry has disappeared from the file.

Module 3.1.12: Changing entries

The program would be of limited use to us if we were not able to change existing data — this module fills that gap.

Module 3.1.12: Lines 16000 – 16340

```
16000 REMark ************************
16010 DEFine PROCedure amend
16020 REMark ************************
16030   temp$=""
16040   start=1
16050   FOR i=0 TO n_fields-1
16060     CLS
16070     PRINT "Entry No: ";s1\\
16080     PRINT field$(i);": ";
16090     PAPER 6 : INK 0
16100     PRINT array$(p_val(s1),start TO start
          +fields(i)-1)
```

```
16110      PAPER 5 : INK 1
16120      AT 10,0
16130      PRINT " Commands available:"
16140      PRINT\\"    'ENTER' leaves field
           unchanged"
16150      PRINT "     Input item to replace
           one shown"
16160      PRINT "     'DDD' deletes whole record"
16170      PRINT "     'ZZZ' leaves record
           unchanged"
16180      INPUT " Which do you require: ";q$
16190      IF q$="ddd" OR q$="DDD" THEN remove
           : RETurn
16200      IF q$="zzz" OR q$="ZZZ" THEN RETurn
16210      IF q$<>""
16220        q$=q$ & FILL$(CHR$(200),fields(i))
16230        q$=q$(1 TO fields(i))
16240      ELSE
16250        temp$=temp$ & array$(p_val(s1),start
             TO start+fields(i)-1)
16260      END IF
16270      start=start+fields(i)
16280      temp$=temp$ & q$
16290    NEXT i
16300    remove
16310    binary_search (temp$)
16320    insert (temp$)
16330    s1=ss
16340 END DEFine amend
```

Commentary

Lines 16050 – 16290: This loop, while it looks very similar to the loop which prints the record in the previous module, is different in that it prints only one item at a time.

Line 16190: Input of 'DDD' when any field is displayed deletes the whole record of which that field is a part — note that an individual *field* cannot simply be deleted, only replaced, since the number of items per entry is fixed.

Line 16200: Input of 'ZZZ' in response to any item returns execution to the SEARCH module. Any changes made to previous fields in the record will be ignored and the record will be unchanged.

Lines 16210 – 16230: If any input is made other than 'DDD' or 'ZZZ' then it is interpreted as being a replacement for the field displayed. The CHR$(200) padding is added, as in the input module.

Lines 16240 – 16260: Pressing ENTER without an input copies the item displayed without changes. If only one item is to be changed, simply press ENTER for all the others.

Lines 16300 – 16330: The amended entry has been built up in TEMP$. When the entry is complete, the original entry is deleted from the file. The reason for this is that the changes made may have altered the correct position of the entry in the ordered file. Once deleted, the entry is sent to the binary search module and re-inserted. Its place in the file is copied into the variable S1, so that the SEARCH module will know which item to display if the position has been changed.

Testing

Run the program and reload the four items of data from disk. Call up the search option and press ENTER to get entry 1 displayed. Now enter 'AAA' in response to the second SEARCH menu. You should see the first item 'AA1' displayed, together with the AMEND menu. Enter 'AA0' and, when the second item is displayed, press ENTER. You should now have returned to the SEARCH module, and the entry should be displayed as:

ONE: AA0
TWO: AA2

Try making other changes and deleting entries. If all works properly, the program is now complete.

PROGRAM 3.2: NNUMBER

Program function

Not all filing is concerned with words. One of the things which microcomputers do best is store and manipulate figures. The current program, Nnumber (short for 'Name and Number'), allows you to store the names of items, the units in which they are usually measured and an associated quantity. Now, before you say that you can't see a use for such a program, consider the average shopkeeper or even domestic cook.

The shopkeeper has a mass of items which are called stock. All of the items which make up the stock have names, they come in different units (box, bottle, bag, etc) and all have a very important quantity associated with them — their price. In order, therefore, to make a microcomputer help with stocktaking or make out an invoice, it must remember these three facts about each item. In the home, the foods we eat each have a name, they come in different units (spoonsful, pounds, pinch, etc) and if we are interested in their effect on our weight, then they all have a quantity associated with them known as 'calories'.

These are just two examples — you can think of many more for yourself — of the importance of being able to record names, units and an associated quantity for a whole variety of items.

The purpose of Nnumber is to allow you to create a 'dictionary' of items

— up to 1000 of them — together with the units in which they are measured and the values associated with those units. Based on that dictionary you will be able to construct lists of items which the program will display and total the quantity for you. Nnumber is as easy to use in adding up the calories for a day's recipes as it is in providing a total price for a collection of goods.

```
item: widgets
UNITS: 10 box @ 15.95
TOTAL: 159.5
oooooooooooooooooooooooooooooooooooo
item: flanges
UNITS: 22 bags @ 7.5
TOTAL: 165
oooooooooooooooooooooooooooooooooooo
item: gubbins
UNITS: 36 pots @ 3.75
TOTAL: 135
oooooooooooooooooooooooooooooooooooo
OVERALL TOTAL: 459.5

Any key to return to menu
```

Figure 3.2: Example of Nnumber in Current List Mode.

Module 3.2.1: Initialisation

A standard module, the only thing worthy of note being that the user is asked to specify the type of item the program will be dealing with. This will be a name such as 'Food item' or 'Stock item'. The phrase input will be used during the course of the program to prompt the user to input another item.

Module 3.2.1: Lines 10000 – 10100

```
10000 REMark ************************
10010 REMark initialise
10020 REMark ************************
10025   CLS
10030   DIM array$(1000,1,20),array(1000),
        current$(100,1,20),current(100)
10060   c_list=0 : it=0 : c_total=0
10070   INPUT "LOAD FROM MICRODRIVE (Y/N): ";Q$
10080   IF Q$="y" OR Q$="Y"
10085     recall
10090   ELSE
10095     INPUT "OVERALL NAME FOR ITEMS:";NAME$
10100   END IF
```

Commentary

Line 10030: The array ARRAY$ is used to record the item name and unit name for each of the items in the dictionary. The associated quantity for each unit is stored in the equivalent element of the array ARRAY.

CURRENT$ and CURRENT will serve a similar purpose to ARRAY$ and ARRAY but for the 'current list' which is extracted from the dictionary.

Line 10060: C_LIST will record the number of items in the 'current list' — the list derived from the main dictionary. It records the number of items in the main file — in this case the dictionary.

Module 3.2.2: Menu

A standard menu module.

Module 3.2.2: Lines 11000 – 11400

```
11000 REMark ************************
11010 REMark menu
11020 REMark ************************
11030   REPeat prompt
11040     CLS
11050     AT 1,15 : PRINT "NNUMBER"
11060     PRINT \\" COMMANDS AVAILABLE:"\\
11070     PRINT,"1) Display current list"
11080     PRINT,"2) Input to current list"
11090     PRINT,"3) Start new current list"
11100     PRINT,"4) Delete from current list"
11110     PRINT,"5) Add to dictionary"
11120     PRINT,"6) Examine dictionary items"
11130     PRINT,"7) Save data to microdrive"
11140     PRINT,"8) Stop"
11150     INPUT\\" WHICH DO YOU REQUIRE: ";choice$
11160     choice="0" & choice$
11170     CLS
11180     IF it=0 AND (choice$="1" OR choice$="4"
          OR choice$="6" OR choice$="7")
11190       AT 10,13 : PRINT "NO DATA YET"
11200       t$=INKEY$(-1)
11210       choice$="0"
11220     END IF
11230     SELect ON choice
11240       ON choice=1 : c_display
11250       ON choice=2 : c_input
11260       ON choice=3 : c_initialise
11270       ON choice=4 : c_delete
11280       ON choice=5 : d_input
11290       ON choice=6 : d_display
11300       ON choice=7 : store
11310       ON choice=8
11320         EXIT prompt
11330     END SELect
11340   END REPeat prompt
11350   CLS
11360   AT 10,11
```

```
11370   PRINT "NAME AND NUMBER"
11380   AT 12,9
11390   PRINT "PROGRAM TERMINATED"
11400   STOP
```

Modules 3.2.3 and 3.2.4: Data files

Two standard modules.

Module 3.2.3 and 3.2.4: Lines 21000 – 22260

```
21000 REMark ************************
21010 DEFine PROCedure store
21020 REMark ************************
21030   CLS
21040   AT 1,14 : PRINT "SAVE DATA"
21050   INPUT\\" Name of data file:";file$
21060   tfile$="mdv1_" & file$
21070   DELETE tfile$
21080   OPEN_NEW #8,"mdv1_" & file$
21090   PRINT #8,it
21100   PRINT #8,c_list
21110   PRINT #8,NAME$
21120   FOR i=0 TO it-1
21130     PRINT #8,array(i)
21140     FOR j=0 TO 1
21150       PRINT #8,array$(i,j)
21160     NEXT j
21170   NEXT i
21180   IF c_list>0
21190     FOR i=0 TO c_list-1
21200       PRINT #8,current(i)
21210       FOR j=0 TO 1
21220         PRINT #8,current$(i,j)
21230       NEXT j
21240     NEXT i
21250   END IF
21260   CLOSE#8
21270 END DEFine store
22000 REMark ************************
22010 DEFine PROCedure recall
22020 REMark ************************
22030   CLS
22040   AT 1,14 : PRINT "RECALL DATA"
22050   DIR mdv1_
22060   INPUT\\" Name of data file:";file$
22070   OPEN_IN #8,"mdv1_" & file$
22080   INPUT #8,it
22090   INPUT #8,c_list
22100   INPUT #8,NAME$
22110   FOR i=0 TO it-1
22120   INPUT #8,array(i)
22130     FOR j=0 TO 1
22140       INPUT #8,array$(i,j)
```



```
22150     NEXT j
22160   NEXT i
22170   IF c_list>0
22180     FOR i=0 TO c_list-1
22190       INPUT #8,current(i)
22200       FOR j=0 TO 1
22210         INPUT #8, current$(i,j)
22220       NEXT j
22230     NEXT i
22240   END IF
22250   CLOSE#8
22260 END DEFine recall
```

Module 3.2.5: Binary search

For a full commentary on this module, see the equivalent module in Unifile. Sorting is done on the basis of the item name in the zero column of ARRAY$. Note that pointer arrays will not be used in the case of this program, so the search takes place directly on the array, not via the value of a pointer.

Module 3.2.5: Lines 16000 – 16200

```
16000 REMark ************************
16010 DEFine PROCedure binary_search
16020 REMark ************************
16030   IF it=0
16040     ss=0
16050     RETurn
16060   END IF
16070   po=INT(LN(it)/LN(2))
16080   ss=2^po-1
16090   FOR i=po TO 0 STEP -1
16100     IF array$(ss,0)<t1$
16110       ss=ss+2^i
16120       IF ss>it-1 THEN ss=it-1
16130     END IF
16140     IF array$(ss,0)>t1$
16150       ss=ss-2^i
16160       IF ss<0 THEN ss=0
16170     END IF
16180   NEXT i
16190   IF array$(ss,0)<t1$ THEN ss=ss+1
16200 END DEFine binary_search
```

Module 3.2.6: Inserting items into the main dictionary

The principle of this module is the simpler moving of items up and down. Only you can decide whether it would be worth your while adapting the method to use a pointer array, as in Unifile.

Module 3.2.6: Lines 17000 – 17130

```
17000 REMark ************************
17010 DEFine PROCedure d_insert
17020 REMark ************************
17030   IF it<>0 AND it>ss
17040     FOR i=it TO ss+1 STEP -1
17050       array$(i,0)=array$(i-1,0)
17060       array$(i,1)=array$(i-1,1)
17070       array(i)=array(i-1)
17080     NEXT i
17090   END IF
17100   array$(ss,0)=t1$
17110   array$(ss,1)=t2$
17120   array(ss)=NN
17130 END DEFine d_insert
```

Module 3.2.7: Entering items for the dictionary

Considerably less complicated than the equivalent module in Unifile, this module accepts three inputs from the user: (a) the name of the item, (b) the name of the units in which it is measured and (c) the quantity associated with those units.

Module 3.2.7: Lines 15000 – 15270

```
15000 REMark ************************
15010 DEFine PROCedure d_input
15020 REMark ************************
15030   REPeat entry
15040     CLS
15050     AT 1,6
15060     PRINT "NEW ITEMS FOR DICTIONARY"
15070     IF it>1000
15080       AT 6,12
15090       PRINT "NO MORE ROOM"
15100       t$=INKEY$(-1)
15110       RETurn
15120     END IF
15130     REPeat confirm
15140       PRINT\\NAME$;
15150       INPUT ' ("ZZZ" to quit): ';t1$
15160       IF t1$="zzz" OR t1$="ZZZ" THEN RETurn
15170       INPUT\"UNITS: ";t2$
15180       PRINT\"QUANTITY PER ";t2$;": ";
15190       INPUT NN
15200       INPUT\\"ARE THESE CORRECT (Y/N):";Q$
15210       IF Q$="y" OR Q$="Y" THEN EXIT confirm
15220     END REPeat confirm
15230     binary_search
15240     d_insert
15250     it=it+1
15260   END REPeat entry
15270 END DEFine d_input
```


Testing

It is now possible to make a real test of what has been entered so far.

Run the program, specify that you are not loading from microdrive and give the name ITEM in response to the prompt for an overall name. At the main menu, choose option 5 'Add to Dictionary'. When the 'new items' screen comes up, input the following three entries:

THING1/BOX/10
THING2/BOTTLE/20
THING3/BAG/40

These items have no particular meaning, they are purely for test purposes.

When you have input the items, return to the main menu by entering 'ZZZ'. Now call up the data file module (option 7) to store the information. Stop the program with menu option eight and enter:

for i = 0 to 2:print array$(i,0),array$(i,1),array(i):next i[ENTER]

You should see this:

THING1	BOX	10
THING2	BOTTLE	20
THING3	BAG	40

Now run the program and specify that you *do* want to load from microdrive. Give the name you supplied when storing the data. When the drive has finished, you should be able to perform the same test of the contents of the arrays, with the same result.

Module 3.2.8: The search routine

As in Unifile, this module provides the user with the opportunity to move through the file of dictionary items, to search for named items or to delete items from the file. The module is simpler than that given in Unifile since it is designed to search for whole items only, rather than combinations of characters stored anywhere in an item. In addition, the structure of a complete entry in Nnumber is far simpler than the structure of an item in the main array of Unifile.

Module 3.2.8: Lines 18000 – 18400

```
18000 REMark ************************
18010 DEFine PROCedure d_display
18020 REMark ************************
18030   ss=0
18040   REPeat search_prompt
18050     CLS
18060     AT 1,15
18070     PRINT "SEARCH"
```

```
18080      PRINT \" ITEM NUMBER: ";ss+1
18090      PRINT\" ";NAME$;": ";array$(ss,0)
18100      PRINT " UNIT: ";array$(ss,1)
18110      PRINT " QUANTITY PER ";array$(ss,1);
           ": ";array(ss)
18120      PRINT\"**********************************
           ****"
18130      PRINT\" COMMANDS AVAILABLE:"
18140      PRINT\"    >Item to be searched for"
18150      PRINT "    >'#' then number to move
           pointer"
18160      PRINT "    >'ENTER' for next item"
18170      PRINT "    >'DDD' to delete item"
18180      PRINT "    >'ZZZ' to quit"
18190      INPUT\" WHICH DO YOU REQUIRE: "; t1$
18200      IF t1$="" THEN t1$="#1"
18210      IF t1$="ddd" OR t1$="DDD"
18220        d_delete
18230        IF it=0
18240          EXIT search_prompt
18250        END IF
18260        t1$="#0"
18270      END IF
18280      IF t1$="zzz" OR t1$="ZZZ" THEN
18290        EXIT search_prompt
18300      END IF
18310      IF t1$(1)<>"#"
18320        binary_search
18330      END IF
18340      IF t1$(1)="#"
18350        ss=ss+t1$(2 TO)
18360        IF ss>it-1 THEN ss=it-1
18370        IF ss<0 THEN ss=0
18380      END IF
18390    END REPeat search_prompt
18400  END DEFine d_display
```

Testing

Simply run the program, reload the three items of data from disk and then call up option 6 from the main menu. You should be able to page through the items, forwards or backwards, using a number preceded by '#', as in Unifile. You should also be able to recover an item by entering the item name — not the unit name.

Module 3.2.9: Deleting an item

The direct equivalent of the delete module in Unifile, but, once again, the method of storage employed here makes for greater simplicity.

Module 3.2.9: Lines 19000 – 19090

```
19000 REMark ************************
19010 DEFine PROCedure d_delete
```



```
19020 REMark ************************
19030   FOR i=ss TO it-1
19040     array$(i,0)=array$(i+1,0)
19050     array$(i,1)=array$(i+1,1)
19060     array(i)=array(i+1)
19070   NEXT i
19080   it=it-1
19090 END DEFine d_delete
```

Testing

Run the program, reload the data, call up option 6 from the main menu, and enter 'DDD' against one of the entries. You should find that it is removed from the file.

Module 3.2.10: Copying items into the current list

The purpose of Nnumber is not simply to keep a dictionary of items and their associated quantities, but to use that dictionary as the basis on which temporary lists can be constructed. The modules which follow are therefore designed to allow the user to add items to the 'current' list, to display that list, to delete single items from it or to delete the whole list in one operation. The current module allows the copying of items from the main dictionary into the current list.

Module 3.2.10: Lines 13000 – 13350

```
13000 REMark ************************
13010 DEFine PROCedure c_input
13020 REMark ************************
13030   REPeat c_prompt
13040     CLS
13050     AT 1,4
13060     PRINT "CURRENT LIST ADDITIONS"
13070     IF c_list=101
13080       AT 10,5
13090       PRINT "CURRENT LIST NOW FULL"
13100       t$=INKEY$(-1)
13110       EXIT c_prompt
13120     END IF
13130     PRINT\\NAME$;" ('ZZZ' to quit): ";
13140     INPUT t1$
13150     IF t1$="zzz" OR t1$="ZZZ"
13160       EXIT c_prompt
13170     END IF
13180     binary_search
13190     IF array$(ss,0)<>t1$
13200       PRINT\NAME$;" ";t1$;" unknown."\
           "Please check."
13210       t$=INKEY$(-1)
13220       EXIT c_prompt
13230     END IF
```

```
13240       PRINT\"UNITS: ";array$(ss,1)
13250       PRINT\"NUMBER OF ";array$(ss,1);
            " UNITS: ";
13260       INPUT qu
13270       INPUT\"Are these correct (Y/N): ";Q$
13280       IF Q$="y" OR Q$="Y"
13290         current$(c_list,0)=array$(ss,0)
13300         current$(c_list,1)=qu & " " & array$
              (ss,1)
13310         current(c_list)=qu*array(ss)
13320         c_list=c_list+1
13330       END IF
13340     END REPeat c_prompt
13350 END DEFine c_input
```

Commentary

Lines 13180 – 13230: These lines perform a check to see if the item input by the user, which is to be placed into the current list, is present in the main dictionary. This is done by calling up the binary search module to obtain the position at which the item would be inserted into the dictionary. The actual contents of the dictionary at this point are then compared with what the user has input. If the item input by the user is in the dictionary then the two items will be the same, otherwise an error message is printed and the module terminates.

Lines 13240 – 13260: Having found the item in the dictionary, the module prints out the units in which it is normally measured and asks how many of those units are to be included.

Lines 13270 – 13330: The user is requested to confirm the accuracy of the entry before it is added to the current list, contained in the variables CURRENT$ and CURRENT. Note that the quantity stored in CURRENT is not the quantity per unit taken from the dictionary, but the *total* quantity for the number of units specified by the user.

Testing

Run the program and reload the data from disk. Call up option 2 on the main menu. Enter the following items and numbers of units:

THING1,1
THING2,2
THING3,3

Now try to get the module to accept 'THING4', which is not present in the dictionary. You should receive an error message asking you to check the item name. Press any key and you will return to the main menu. Before you do anything else, call up option 7 — this will store the current list you have just created, along with the main dictionary. Finally choose option 8 to stop the program.

Now enter the following line in direct mode:

for i = 0 to 2:print current$(i,0),current$(i,1),current(i): next i[ENTER]

You should see this:

THING1	1 BOX	10
THING2	2 BOTTLE	40
THING3	3 BAG	120

Module 3.2.11: Displaying the current list

The sole purpose of this module is to print the entries which make up the current list, one by one on the screen. After each entry, the user is required to press a key before the next is displayed. This is because the list will normally be longer than the screen itself and the user will not want the list to scroll up off the screen faster than it can be read. At the end of the list, the total of the associated quantities for the contents of the current list is given.

Module 3.2.11: Lines 14000 – 14180

```
14000 REMark ************************
14010 DEFine PROCedure c_display
14020 REMark ************************
14030   IF c_list>0
14040     CLS
14050     c_total=0
14060     FOR i=0 TO c_list-1
14070       PRINT NAME$;": ";current$(i,0)
14080       PRINT "UNITS: ";current$(i,1);" @ ";
            current(i)/current$(i,1)
14090       PRINT "TOTAL: ";current(i)
14100       PRINT "oooooooooooooooooooooooooooo
            ooooooo"
14110       t$=INKEY$(-1)
14120       c_total=c_total+current(i)
14130     NEXT i
14140     PRINT "OVERALL TOTAL: ";c_total
14150     PRINT\"Any key to return to menu"
14160     t$=INKEY$(-1)
14170   END IF
14180 END DEFine c_display
```

Module 3.2.12: Deleting items from the current list

This is a simplified version of the kind of search module used for the main dictionary.

Module 3.2.12: Lines 20000 – 20370

```
20000 REMark ************************
20010 DEFine PROCedure c_delete
```

```
20020 REMark ************************
20030   count=0
20040   REPeat c_prompt
20050     IF c_list=0 THEN EXIT c_prompt
20060     CLS
20070     AT 1,8
20080     PRINT "CURRENT LIST DELETION"
20090     PRINT\\"ITEM NUMBER ";count+1;" OF ";
          c_list
20100     PRINT\current$(count,0)
20110     PRINT\current$(count,1);" @ ";current
          (count)/current$(count,1)
20120     PRINT\\" COMMANDS AVAILABLE:"
20130     PRINT\"    >'DDD' = delete"
20140     PRINT "    >'ENTER' = next item"
20150     PRINT "    >'ZZZ' = quit"
20160     PRINT "    >'#' + number to move pointer"
20170     INPUT\"WHICH DO YOU REQUIRE: ";Q$
20180     IF Q$="zzz" OR Q$="ZZZ"
20190       EXIT c_prompt
20200     END IF
20210     IF Q$="" THEN Q$="#1"
20220     IF Q$(1)="#"
20230       count=count+ Q$(2 TO)
20240     END IF
20250     IF Q$="ddd" OR Q$="DDD"
20260       FOR i=count TO c_list-1
20270         current$(i,0)=current$(i+1,0)
20280         current$(i,1)=current$(i+1,1)
20290         current(i)=current(i+1
20300       NEXT i
20310       c_list=c_list-1
20320       IF c_list=0 THEN EXIT c_prompt
20330     END IF
20340     IF count>c_list-1 THEN count=c_list-1
20350     IF count<0 THEN count=0
20360   END REPeat c_prompt
20370 END DEFine c_delete
```

Testing

Run the program and reload the data file containing the current list. Call up option 4, 'Delete from Current List', from the main menu. You should now be able to page through the three items in the current list and delete one — the fact that it has been deleted may be checked by displaying the current list.

Module 3.2.13: Initialising the current list

For many applications, the need will be to construct a current list, obtain the total involved, and then quickly move on to a fresh list. This simple module wipes out the contents of the current list in one operation.

Module 3.2.13: Lines 12000 – 12060

```
12000 REMark ************************
12010 DEFine PROCedure c_initialise
12020 REMark ************************
12030   DIM current$ (100,1,20)
12040   DIM current (100)
12050   c_list=0
12060 END DEFine c_initialise
```

Testing

Run the program, reload the datafile which includes the current list and specify option 3, 'Start Fresh List', from the main menu. The main menu should be almost instantly re-printed. Now try calling option 1 from the main menu. Once again, all that happens is that the main menu reprints itself — the display module has been called but since there is nothing to be displayed, execution immediately returns.

If this test is completed satisfactorily, the program is complete and ready for use.

PROGRAM 3.3: MULTIQ

Program function

In the final program in this chapter, we turn to the worthy subject of education for a little bit of fun. I'm not entirely sure how much you can learn about your chosen topic using this program, but it is fun to use, and makes answering questions addictive. Not only that, the program is a star in its own right, for an earlier version of MultiQ on the Sinclair Spectrum was the first program (at least according to the press hand-out) to set a listeners' quiz on radio in Britain.

MultiQ is, as the name suggests, a multi-purpose program that can at one moment be a language tutor and the next be quizzing you on abstruse points of 19th century history. It does all this by creating random answer tests of the type increasingly used in public examinations, setting a question and providing five possible answers, only one of which is correct. A running score is kept, providing an assessment of the user's knowledge on the current topic. Of course, the main work is on the part of the programmer, since not only does the program itself have to be entered, there is also the small matter of entering a large enough body of questions to ensure that the tests are meaningful.

Module 3.3.1: Initialisation

As with all the multi-purpose programs in this chapter, this module includes provision for describing the type of file to be handled in the current session.

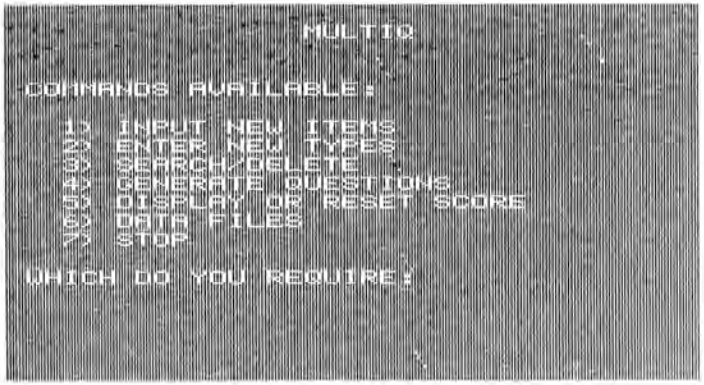


Figure 3.3: MultiQ's Command Menu.

Module 3.3.1: Lines 10000 – 10240

```
10000 REMark ************************
10010 REMark initialise
10020 REMark ************************
10030   PAPER 3: INK 0
10040   DIM name$(1,20),qu(4),array$(499,1,20),
        q_type$(9,20),n_type(1,9)
10050   right=0 : total=0 : IT=0
10060   CLS : CLS#0
10070   INPUT\"LOAD FROM MICRODRIVE (Y/N): ";Q$
10080   IF Q$="y" OR Q$="Y"
10090     recall
10100   ELSE
10110     REPeat check
10120       CLS
10130       AT 1,12 : PRINT "TEST STRUCTURE"
10140       INPUT\\"NAME FOR ANSWER: ";T1$
10150       INPUT\"NAME FOR QUESTION: ";T2$
10160       INPUT\\"Are these correct (y/n): ";Q$
10170       IF Q$="Y" OR Q$="y" THEN EXIT check
10180     END REPeat check
10190     q_type$(0)="No type"
10200     n_types=1
10210     name$(0)=T1$
10220     name$(1)=T2$
10230     types
10240   END IF
```

Commentary

Line 10040: The array NAME$ will be used to record the general names given by the user to questions and answers, QU will be used in the setting of random tests, ARRAY$ will hold the main file of questions and answers,

Q_TYPES will hold the names of types which may be allocated to questions and answers and N_TYPE will store the number of each type in the main file.

Lines 10110 – 10180: MultiQ, as we have already noted, sets tests, ie it asks questions and displays possible answers. These lines enable the user to give a general title to questions and answers. If the program were to be used for the purposes of learning French, for instance, you might call the question 'ENGLISH WORD' and the answer 'EQUIVALENT FRENCH WORD'.

Module 3.3.2: The menu

A standard menu module.

Module 3.3.2: Lines 11000 – 11330

```
11000 REMark ************************
11010 REMark menu
11020 REMark ************************
11030   REPeat display
11040     PAPER 3 : INK 0
11050     CLS : CLS#0
11060     AT 1,16 : PRINT "MULTIQ"
11070     PRINT\\" COMMANDS AVAILABLE: "
11080     PRINT\"    1) INPUT NEW ITEMS"
11090     PRINT "    2) ENTER NEW TYPES"
11100     PRINT "    3) SEARCH/DELETE"
11110     PRINT "    4) GENERATE QUESTIONS"
11120     PRINT "    5) DISPLAY OR RESET SCORE"
11130     PRINT "    6) DATA FILES"
11140     PRINT "    7) STOP"
11150     INPUT\" WHICH DO YOU REQUIRE: ";Z
11160     IF (Z>2 AND Z<7) AND IT=0
11170       PRINT "NO DATA YET"
11180       T$=INKEY$(-1)
11190       Z=0
11200     END IF
11210     SELect ON Z
11220       ON Z=1 : n_items
11230       ON Z=2 : types
11240       ON Z=3 : search
11250       ON Z=4 : questions
11260       ON Z=5 : score
11270       ON Z=6 : store
11280       ON Z=7
11290         CLS
11300         PRINT#0,"Classroom closed"
11310         STOP
11320     END SELect
11330   END REPeat display
```

Module 3.3.3: Setting question types

In the course of entering later modules you will discover that MultiQ makes provision for two different levels of difficulty in the tests that it sets. It does this on the basis of question types. Turning again to the example of using the program as a French language tutor, it is clearly possible to divide up the kinds of words being displayed into different grammatical groups, like verbs, nouns, adjectives and so on. If an English word is displayed which is a verb, and of the five possible French answers only one is a verb, the test is a great deal easier than if all five of the possible answers were verbs. The purpose of the current module is to allow the user to define up to 10 types into which questions or answers will fall, with the facility to tag a type on to a question as it is entered. When MultiQ later comes to set a test, it will ask whether the user wishes possible answers to questions to be drawn only from the same type as the correct answer or from the whole stock of answers.

Module 3.3.3: Lines 12000 – 12210

```
12000 REMark ************************
12010 DEFine PROCedure types
12020 REMark ************************
12030   REPeat loop
12040     CLS
12050     AT 1,16 : PRINT "TYPES"
12060     PRINT\\" Types so far:"\\
12070     FOR i=0 TO n_types-1
12080       PRINT "   ";i+1;") ";q_type$(i)
12090     NEXT i
12100     IF n_types=10
12110       PRINT\\"NO ROOM FOR MORE TYPES."
12120       T$=INKEY$(-1)
12130       RETurn
12140     END IF
12150     PRINT\\"INPUT 'ZZZ' TO QUIT OR:-"
12160     INPUT\"INPUT NEW TYPE: ";Q$
12170     IF Q$="zzz" THEN RETurn
12180     q_type$(n_types)=Q$
12190     n_types=n_types+1
12200   END REPeat loop
12210 END DEFine types
```

Testing

You should now be in a position to run the program and enter up to 10 question types. You can confirm that the types have been accepted by stopping the program and typing:

for i = 0 to 9:print q_type$(i):next i[ENTER]

Module 3.3.4: Binary search

A standard search module, working alphabetically on the basis of the answers to questions. Note that, as you will discover when you enter the new items module in a moment, each answer is preceded by a single character (0 – 9) to indicate which type it is. The file will therefore be sorted first of all on the basis of types and, within types, on the basis of the alphabetical order of answers.

Module 3.3.4: Lines 14000 – 14200

```
14000 REMark ************************
14010 DEFine PROCedure b_search
14020 REMark ************************
14030   IF IT=0
14040     ss=0
14050     RETurn
14060   END IF
14070   po=INT(LN(IT)/LN(2))
14080   ss=2^po-1
14090   FOR i=po TO 0 STEP -1
14100     IF array$(ss,0)<T1$
14110       ss=ss+2^i
14120       IF ss>IT-1 THEN ss=IT-1
14130     END IF
14140     IF array$(ss,0)>T1$
14150       ss=ss-2^i
14160       IF ss<0 THEN ss=0
14170     END IF
14180   NEXT i
14190   IF array$(ss,0)<T1$ THEN ss=ss+1
14200 END DEFine binary_search
```

Module 3.3.5: Inserting an item

A standard insert module.

Module 3.3.5: Lines 15000 – 15130

```
15000 REMark ************************
15010 DEFine PROCedure insert
15020 REMark ************************
15030   IF IT<>0 AND IT>ss
15040     FOR i=IT TO ss+1 STEP -1
15050       array$(i,0)=array$(i-1,0)
15060       array$(i,1)=array$(i-1,1)
15070     NEXT i
15080   END IF
15090   array$(ss,0)=T1$
15100   array$(ss,1)=T2$
15110   IT=IT+1
15120   update
15130 END DEFine insert
```

Module 3.3.6: Keeping track of types

We have already entered the module which allows types to be recorded but we also need to give the program the ability to record how many of each type there are in the file and where each group of types starts. The record of how many of each type there are is dealt with by the input module, which simply adds 1 to the relevant element in the array N__TYPE. This module is called whenever a new entry or deletion is made. Its purpose is to record in the other side of N__TYPE the cumulative totals of types from 0 – 9. Eventually the answers will be arranged in type order within the main file, so knowing the total of items which fall under types 0 to 2, for instance, tells us where the items under group 3 start.

Module 3.3.6: Lines 16000 – 16080

```
16000 REMark ************************
16010 DEFine PROCedure update
16020 REMark ************************
16030   su=0
16040   FOR i=0 TO 9
16050     n_type(1,i)=su
16060     su=su+n_type(0,i)
16070   NEXT i
16080 END DEFine update
```

Module 3.3.7: Entering a new item

A straightforward module which allows the user to enter a new question and answer, then attach a type to it.

Module 3.3.7: Lines 13000 – 13320

```
13000 REMark ************************
13010 DEFine PROCedure n_items
13020 REMark ************************
13030   REPeat loop
13040     CLS
13050     AT 1,15 : PRINT "NEW ITEMS"
13060     IF IT=500
13070       PRINT\\"NO MORE ROOM"
13080       T$=INKEY$
13090       RETurn
13100     END IF
13110     PRINT\"INPUT ITEM OR 'ZZZ' TO QUIT:"\\
          name$(0);": ";
13120     INPUT T1$
13130     IF T1$="zzz" OR T1$="ZZZ" THEN EXIT loop
13140     PRINT name$(1);": ";
13150     INPUT T2$
13160     PRINT\"TYPES:"
13170     FOR i=0 TO n_types-1
13180       PRINT " ";i+1;") ";q_type$(i)
13190     NEXT i
```



```
13200     INPUT\"Which is it: ";t3
13210     t3=t3-1
13220     IF t3<0 OR t3>n_types THEN t3=0
13230     PRINT "Type: ";q_type$(t3)
13240     INPUT\ "ARE THESE CORRECT (Y/N): ";0$
13250     IF 0$="y" OR 0$="Y"
13260       n_type(0,t3)=n_type(0,t3)+1
13270       T1$=t3 & T1$
13280       b_search
13290       insert
13300     END IF
13310   END REPeat loop
13320 END DEFine n_items
```

Commentary

Line 13260: The element of the array N__TYPE representing the type specified for the current question and answer is increased by 1. It is this array, as we have seen, which is worked on by the UPDATE module in recording where each group starts in the array.

Testing

You should now be able to run the program, specify types and then call up menu option 1 to begin entering questions and answers. To verify that items are being received correctly, enter the following data:

QUESTION	ANSWER	TYPE
Q111	A111	3
Q222	A222	2
Q333	A333	1

and then quit the program. Now type:

for i = 0 to 2:for j = 0 to 1:print array$(i,j):next j:next i[ENTER]

You should see:

0A333
Q333
1A222
Q222
2A333
Q111

Module 3.3.8 and 3.3.9: Storing data

Now that items of data can be entered, it is time to enter the two standard modules which will store and recall items.

Modules 3.3.8 and 3.3.9: Lines 22000 – 23220

```
22000 REMark ************************
22010 DEFine PROCedure store
22020 REMark ************************
22030   CLS
22040   AT 1,14 : PRINT "SAVE DATA"
22050   INPUT\\" Name of data file:";file$
22060   tfile$="mdv1_" & file$
22070   DELETE tfile$
22080   OPEN_NEW #8,"mdv1_" & file$
22090   PRINT#8,IT
22100   FOR i=0 TO 1
22110     PRINT#8,name$(i)
22120     FOR j=0 TO 9
22130       PRINT #8,n_type(i,j)
22140     NEXT j
22150     FOR j=0 TO IT-1
22160       PRINT#8,array$(j,i)
22170     NEXT j
22180   NEXT i
22190   FOR i=0 TO 9
22200     PRINT #8,q_type$(i)
22210   NEXT i
22220   CLOSE#8
22230 END DEFine store
23000 REMark ************************
23010 DEFine PROCedure recall
23020 REMark ************************
23030   CLS
23040   AT 1,14 : PRINT "RECALL DATA"
23050   DIR mdv1_
23060   INPUT\\" Name of data file:";file$
23070   OPEN_IN #8,"mdv1_" & file$
23080   INPUT#8,IT
23090   FOR i=0 TO 1
23100     INPUT#8,name$(i)
23110     FOR j=0 TO 9
23120       INPUT #8,n_type(i,j)
23130     NEXT j
23140     FOR j=0 TO IT-1
23150       INPUT #8,array$(j,i)
23160     NEXT j
23170   NEXT i
23180   FOR i=0 TO 9
23190     INPUT #8,q_type$(i)
23200   NEXT i
23210   CLOSE#8
23220 END DEFine recall
```

Module 3.3.10: The user search

A simple search module which allows the user to search backwards and forwards through the main file, displaying and, after the next module is entered, deleting items.

Module 3.3.10: Lines 20000 – 20360

```
20000 REMark ************************
20010 DEFine PROCedure search
20020 REMark ************************
20030   ss=0
20040   REPeat search_prompt
20050     CLS
20060     AT 1,15
20070     PRINT "SEARCH"
20080     PRINT \" ITEM NUMBER: ";ss+1
20090     PRINT " ";name$(0);": ";array$(ss,0,2
          TO)
20100     PRINT " ";name$(1);": ";array$(ss,1)
20110     PRINT " Type: ";q_type$(array$(ss,0,1))
20120     PRINT\"**********************************
          ****"
20130     PRINT\" COMMANDS AVAILABLE:"
20140     PRINT "    >'#' then number to move
          pointer"
20150     PRINT "    >'ENTER' for next item"
20160     PRINT "    >'DDD' to delete item"
20170     PRINT "    >'ZZZ' to quit"
20180     INPUT\" WHICH DO YOU REQUIRE: "; T1$
20190     IF T1$="" THEN T1$="#1"
20200     IF T1$="ddd" OR T1$="DDD"
20210       d_delete
20220       IF IT=0
20230         EXIT search_prompt
20240       END IF
20250       T1$="#0"
20260     END IF
20270     IF T1$="zzz" OR T1$="ZZZ"
20280       EXIT search_prompt
20290     END IF
20300     IF T1$(1)="#"
20310       ss=ss+T1$(2 TO)
20320       IF ss>IT-1 THEN ss=IT-1
20330       IF ss<0 THEN ss=0
20340     END IF
20350   END REPeat search_prompt
20360 END DEFine d_display
```

Testing

Run the program and recall the data you have stored on microdrive. Using menu item 3 you should be able to page forwards and backwards through the items.

Module 3.3.11: Deleting an item

A standard deletion module.

Module 3.3.11: Lines 21000 – 21100

```
21000 REMark ************************
21010 DEFine PROCedure d_delete
21020 REMark ************************
21030   n_type(0,array$(ss,0,1)-1)=n_type(0,
        array$(ss,0,1)-1)-1
21040   FOR i=ss TO IT-1
21050     array$(i,0)=array$(i+1,0)
21060     array$(i,1)=array$(i+1,1)
21070   NEXT i
21080   IT=IT-1
21090   update
21100 END DEFine d_delete
```

Testing

Follow the procedure as for the previous module but enter 'DDD' against one or other of the items. You should find that the specified item has been removed.

Module 3.3.12: Setting questions

We now turn to the modules which constitute the novelty of MultiQ by setting the multiple choice tests. The current module handles the visible part of the process, the display of the questions and possible answers and the user's choice of correct answer.

Module 3.3.12: Lines 17000 – 17450

```
17000 REMark ************************
17010 DEFine PROCedure questions
17020 REMark ************************
17030   CLS
17040   AT 1,15 : PRINT "QUESTIONS"
17050   PRINT\\"Do you wish answers to be drawn"
17060   PRINT "from one type only (harder) or
        from"
17070   PRINT "the whole stock (easier)?"
17080   PRINT\\" 1) one type only"
17090   PRINT " 2) all types"
17100   INPUT\"Which: ";rq
17110   IF rq<1 OR rq>2 THEN rq=2
17120   REPeat loop
17130     r_select
17140     CLS
17150     PRINT name$(1);": ";array$(qu(q_pos),1)
17160     PRINT\\\\name$(0);":"
17170     FOR i=0 TO 4
17180       PRINT " ";i+1;") ";array$(qu(i),0,2
          TO)
17190     NEXT i
```



```
17200     REPeat check
17210       INPUT\"Which is the right answer: ";ra
17220       IF ra>0 AND ra<6 THEN EXIT check
17230     END REPeat check
17240     IF ra-1=q_pos
17250       CLS
17260       FLASH 1 : CSIZE 3,1 : FILL 1 : INK 7
            : STRIP 7
17270       CIRCLE 82,55,15,2,0
17280       INK 2
17290       AT 4,11 : PRINT "RIGHT!"
17300       T$=INKEY$(-1)
17310       FLASH 0 : CSIZE 0,0 : FILL 0 : INK 0
            : PAPER 3 : STRIP 3
17320       right=right+1
17330     ELSE
17340       PRINT\"Sorry, that's wrong"
17350       PRINT\"The correct answer was ";
17360       UNDER 1
17370       PRINT array$(main_q,0,2 TO)
17380       UNDER 0
17390     END IF
17400     total=total+1
17410     AT 18,0
17420     INPUT "Any more (y/n): ";Q$
17430     IF Q$<>"Y" AND Q$<>"y" THEN EXIT loop
17440   END REPeat loop
17450 END DEFine questions
```

Commentary

Lines 17050 – 17110: We have already noted that MultiQ is capable of setting two levels of test. These lines allow the user to specify whether possible answers are to be drawn from the whole file or from the same type as the correct answer.

Lines 17170 – 17230: The question and the five possible answers, which will be selected by the next module, are displayed. The positions of the five possible answers are held in the array QU, and the position of the correct answer within QU is recorded by the variable Q_POS.

Lines 17240 – 17390: If the user's chosen answer, as represented by the variable RA, corresponds with the correct answer's position (Q_POS), then the screen flashes the enlarged word 'RIGHT' in the centre of the screen. The variable RIGHT, which records the number of right answers, is incremented by 1. If the wrong answer is given, the user is informed that the answer is wrong and told what the correct answer was.

Lines 17400: TOTAL, the variable which records the total number of questions asked, is incremented by 1.

Module 3.3.13: Selecting the random questions

Having given ourselves the ability to display the questions and answers, we turn to the considerably more complex matter of *selecting* the questions and answers. Before the detailed commentary, we shall take a general look at the method involved.

What we want is to select one question and its corresponding correct answer and then fill the array QU with five numbers, representing the positions in the main file of the five potential answers, including the correct answer. The main question and answer are first chosen randomly from the entire file and the number of the question in the main array placed in a random position within the array QU.

Having placed the main question into QU, four alternative answers have now to be found. Depending on whether the user wants the easy or harder form of the test, the four alternative answers will be selected either from the whole of the main file or from that section which contains answers whose type is the same as that of the main answer. The four answers are chosen randomly from the appropriate section of the file, with checks being made that the same answer is not included twice and that no answer is included which appears to be identical to the correct answer.

Module 3.3.13: Lines 18000 – 18320

```
18000 REMark ************************
18010 DEFine PROCedure r_select
18020 REMark ************************
18030   start=0 : finish=IT-1
18040   main_q=RND(start TO finish)
18050   c_type=array$(main_q,0,1)
18060   IF rq=1
18070     start=n_type(1,c_type)
18080     finish=start+n_type(0,c_type)-1
18090     IF finish-start<4
18100       start=0
18110       finish=IT-1
18120     END IF
18130   END IF
18140   q_pos=RND(4)
18150   qu(q_pos)=main_q
18160   FOR i=0 TO 4
18170     IF i<>q_pos
18180       REPeat choose
18190       duplicate=0
18200       duff=RND(start TO finish)
18210       IF array$(duff,0,2 TO)=array$(main_q,
            0,2 TO) THEN duplicate=1
18220       IF i>0
18230         FOR j=0 TO i-1
18240           IF array$(duff,0,2 TO)=array$(qu
                (j),0,2 TO) THEN duplicate=1
```



```
18250           NEXT j
18260         END IF
18270         IF duplicate=0 THEN EXIT choose
18280         END REPeat choose
18290       qu(i)=duff
18300       END IF
18310     NEXT i
18320 END DEFine r_select
```

Commentary

Line 18030: START and FINISH represent the range of the file from which random selections of questions will be made. Originally they are set so that the whole file is included.

Line 18040: The main question and answer are selected and their position stored in MAIN_Q. The type of the answer is recorded by the variable C_TYPE.

Lines 18060 – 18130: If the user has specified the harder type of test, then START and FINISH are reset so that they point to the beginning and end of the group of questions of the same type as the main question. If it turns out that there are less than five questions in that group, so that it would be impossible to choose five different answers, START and FINISH are again set to the beginning of the file. If you specify the harder form of the test and find that the program does not provide it, the probable reason is that there are no answers of the same type as the main question.

Lines 18140 – 18150: The position in the main array of the main question is placed in a random position with QU.

Lines 18160 – 18310: This loop chooses the four alternative answers from the part of the file indicated by START and FINISH, each temporarily stored in the variable DUFF. Within the loop, checks are made comparing the new answer with the correct answer, which may be anywhere in QU and the answers previously placed in QU. Note that it is not sufficient simply to check that the same answer from the main file is not duplicated. Two questions from different parts of the main file may well have identical answers. By the end of the loop, QU contains the positions of five different answers within the main file.

Testing

The only effective way to test these modules is to enter a sufficient body of data to allow tests to be generated. The best short test would be first to register two question types, entitled TYPE 1, TYPE 2...TYPE 6. Now enter a series of questions in the form Q1, Q2...Q10, with answers in the form A1,A2...A10. The type for the first five questions should be TYPE 1, with the remaining questions as TYPE 2...TYPE 6. This provides one set

of questions capable of generating the harder form of the tests, and five others with only one question each.

Call up the random question generator and specify the harder form of the test. You should find that you can continue to answer questions and to be correctly informed as to whether your answers are right or wrong. When a question from the first five is chosen, the five answers should be in the range 1 – 5. When other main questions are chosen, you should be able to see that the answers are drawn from the whole of the file of 10 questions.

Module 3.3.14: Calculating the score

The final touch we shall give the program is to enable it to calculate a meaningful score for the tests. This is not quite as easy as it seems, since it is not just a matter of taking the number of right answers as a percentage of the total. If the user simply specifies the first answer for each test, on average that will be the right answer once in every five questions. A straight score of 20% may well indicate that the user has absolutely no clue as to the correct answer. The solution adopted is to subtract one-fifth of the total questions (the number that could be expected by sheer chance) from the right answers and to express that figure as a percentage of four-fifths of the total number of questions.

Module 3.3.14: Lines 19000 – 19180

```
19000 REMark ************************
19010 DEFine PROCedure score
19020 REMark ************************
19030   CLS
19040   AT 1,15 : PRINT "SCORE"
19050   IF total=0
19060     PRINT\\" NO SCORE YET"
19070     T$=INKEY$(-1)
19080     RETurn
19090   END IF
19100   PRINT \\"TOTAL ANSWERS: ";total
19110   PRINT\"CORRECT ANSWERS: ";right
19120   PRINT\"SCORE: ";INT(((right-total/5)/
        (total*.8))*100);"%"
19130   INPUT\\\"Do you wish to reset score
        (y/n):";Q$
19140   IF Q$="y" OR Q$="Y"
19150     total=0
19160     right=0
19170   END IF
19180 END DEFine score
```

Testing

Run the test for the previous module again. When you have answered a few

questions, go back to the main menu and call up the score module. You should find that the score you are given makes rough sense, even though it will not be easy to relate it exactly to the number of right answers you have given. You should also be given the option to zero the score and start a new test from scratch.

If performance on this test is satisfactory, the program is ready for use.

CHAPTER 4

Money Matters

In this final chapter we turn our attention to one important aspect that we have so far overlooked, the QL and money. It is a subject which cannot realistically be ignored because microcomputers deal so superbly with financial matters. The sums involved are seldom vast — or if they are then it is unlikely that they are being dealt with on an inexpensive micro — and the calculations involved are usually simple — a matter of addition and subtraction as money comes in and goes out.

The real advantage of the microcomputer, however, is not simply that it can deal with money, for so can the human brain: the microcomputer can store information, retrieve it quickly and then present it in such a way that it can be immediately understood.

The three programs in this chapter are:

BANKER: Allows a clear record of all payments into and out of a bank account for a 12-month period. Optional printout of monthly accounts.

ACCOUNTANT: Produces a clear set of traditionally laid out accounts from a set of figures.

BUDGET: Stores and processes large amounts of information about family finances and produces a revealing analysis of the picture over a 12-month period. Allows 'what if' decisions about possible expenditure to be explored.

PROGRAM 4.1: BANKER

Program function

The object of this program is to allow the user to keep a clear and continuously updated record of a single bank account, the names of payments, their date and the amount, including the ability to specify not only single payments, but recurring expenses or receipts, no matter how irregular the period. The program is designed to deal with an account for the period of one calendar year and will output either to the screen or to a printer.

```
STATEMENT FOR FEBRUARY

BALANCE C/F:    1000.00

                          ITEM      TOTAL

 3 MORTGAGE           -   250.00     750.00
 9 ELECTRICITY        -    57.97     692.03
12 CAR REPAIRS        -    82.56     609.47
15 RATES              -    98.45     511.02
18 GAS                -    45.12     465.90
22 GROCERIES          -    56.78     409.12
27 SINCLAIR QL        -   399.00      10.12
```

Figure 4.1: Monthly Statement Prepared by Banker.

Module 4.1.1: Initialisation

A standard initialisation module.

Module 4.1.1: Lines 1000 – 1150

```
1000 REMark ************************
1010 REMark initialise
1020 REMark ************************
1030   PAPER 2 : INK 7
1040   CLS : CLS#0
1050   sum = 0 : PA=0 : space$=fill$(" ",8)
1060   cl$=fill$(" ",37)
1070   DIM payment$(499,15),p_month$(499,11),
       amount(499,1)
1080   RESTORE
1090   DIM mo$(11,9)
1100   FOR i=0 TO 11
1110     READ mo$(i)
1120   NEXT i
1130   INPUT "LOAD FROM MICRODRIVE (Y/N):";Q$
1140   IF Q$="Y" OR Q$="y" THEN recall
1150   DATA "JANUARY","FEBRUARY","MARCH","APRIL",
       "MAY","JUNE","JULY","AUGUST","SEPTEMBER",
       "OCTOBER","NOVEMBER","DECEMBER"
```

Commentary

Line 1070: The array PAYMENT$ will be used to store the names of payments. P_MONTH$ will contain a special string, explained later, which records the months in which the particular payment is made. The numerical array AMOUNT will store the amount of each payment and the day of the month on which it is made.

Lines 1080 – 1120: This loop reads the names of the months of the year into the array MO$.

Module 4.1.2: The program menu

A standard menu module.

Module 4.1.2: Lines 2000 – 2330

```
2000 REMark ************************
2010 REMark menu
2020 REMark ************************
2030   REPeat prompt
2040     PAPER 2 : INK 7
2050     CLS
2060     AT 1,15
2070     PRINT "BANKER"
2080     PRINT\\" COMMANDS AVAILABLE:"
2090     PRINT\\,"1) NEW PAYMENTS"
2100     PRINT,"2) EXAMINE/DELETE PAYMENTS"
2110     PRINT,"3) PRINT STATEMENT"
2120     PRINT,"4) SAVE FILE"
2130     PRINT,"5) STOP"
2140     INPUT\\" WHICH DO YOU REQUIRE:";Z
2150     IF PA=0 AND (Z=2 OR Z=3 OR Z=4)
2160       PRINT\\,"SORRY, NO DATA YET"
2170       T$=INKEY$(-1)
2180       Z=0
2190     END IF
2200     SELect ON Z
2210       ON Z=1 : new_entries
2220       ON Z=2 : search
2230       ON Z=3 : statement
2240       ON Z=4 : store
2250       ON Z=5 : EXIT prompt
2260     END SELect
2270   END REPeat prompt
2280   CLS
2290   AT 10,15
2300   PRINT "BANKER"
2310   AT 12,8
2320   PRINT "CLOSED FOR BUSINESS"
2330   STOP
```

Module 4.1.3: Entering new items

This is a more complex input module than we have been used to so far, for the simple reason that the entries themselves are more complex. For each item recorded, five facts need to be known: whether the payment is a credit or a debit (money received or money paid out), the name of the payment, the amount, the months in which the payment is due, and the day of the month on which the payment is made.

Module 4.1.3: Lines 3000 – 3660

```
3000 REMark ************************
3010 DEFine PROCedure new_entries
3020 REMark ************************
```

```
3030 REPeat n_prompt
3040     CLS
3050     AT 1,14
3060     PRINT "NEW ITEMS"
3070     REPeat CREDIT
3080       PRINT\\"1) CREDIT"\"2) DEBIT"
3090       INPUT\"WHICH DO YOU REQUIRE: ";CD
3100       IF CD=1 OR CD=2 THEN EXIT CREDIT
3110     END REPeat CREDIT
3120     CLS
3130     AT 1,14
3140     PRINT "NEW ITEMS"
3150     CD=CD-1
3160     IF CD=0
3170       PRINT\"CREDIT ITEM"
3180     ELSE
3190       PRINT\"DEBIT ITEM"
3200     END IF
3210     INPUT\"NAME OF PAYMENT: ";tpay$
3220     INPUT\"AMOUNT: ";tpay
3230     IF CD=1 THEN tpay=tpay*-1
3240     rec_m$=""
3250     FOR i=0 TO 11
3260       AT 11,0
3270       INPUT (mo$(i)&" (Y/N): ");tm$
3280       IF tm$="Y" OR tm$="y"
3290         rec_m$=rec_m$ & "1"
3300       ELSE
3310         rec_m$=rec_m$ & "0"
3320       END IF
3330       AT 11,0
3340       PRINT cl$
3350     NEXT i
3360     AT 11,0
3370     PRINT "TO BE PAYED IN: ";
3380     FOR i=1 TO 12
3390       IF rec_m$(i)="1"
3400         PRINT !mo$(i-1)!
3410       END IF
3420     NEXT i
3430     INPUT\\"DAY OF PAYMENT (0-31): ";day
3440     INPUT\"ARE THESE CORRECT (Y/N): ";T$
3450     IF T$="y" OR T$="Y" THEN EXIT n_prompt
3460     PRINT\"NOT REGISTERED"
3470     T$=INKEY$(-1)
3480   END REPeat n_prompt
3490   count=PA
3500   REPeat loop
3510     IF day>amount(count,1)
3520       payment$(count+1)=tpay$
3530       p_month$(count+1)=rec_m$
3540       amount(count+1,0)=tpay
3550       amount(count+1,1)=day
3560       EXIT loop
```



```
3570       END IF
3580       payment$(count+1)=payment$(count)
3590       p_month$(count+1)=p_month$(count)
3600       amount(count+1,0)=amount(count,0)
3610       amount(count+1,1)=amount(count,1)
3620       count=count-1
3630       IF count<0 THEN EXIT loop
3640     END REPeat loop
3650     PA=PA+1
3660 END DEFine new_entries
```

Commentary

Lines 3030 – 3480: The overall loop which allows the user to confirm or reject the information input.

Lines 3070 – 3200: The program clearly needs to know whether the item is to be paid out or received, debit or credit. This is recorded in the form of the variable CD (Credit/Debit), and an appropriate heading placed on the screen.

Line 3230: If the user has specified a debit item, ie a payment out of the account, the amount input is multiplied by minus one.

Lines 3240 – 3420: The months in which the payment is to be made are input in response to a series of prompts. For each of the 12 months, a character is added to the temporary string REC__M$. If the payment is to be made in the corresponding month, the character added is a '1', otherwise it is a '0'. The FOR loop beginning at line 3380 prints out the names of the specified months as recorded in REC__M$, so that the user can determine that they are the months intended.

Lines 3500 – 3640: This is the loop which inserts a new item into the main file, in order of day of payment. The technique is a very simple one. Starting with the last entry, the loop compares the day on which the new payment is made with the day of payment of the item in the main file. If the day of payment in the main file is less than the day of payment for the new item, then the new item is inserted in the space following the item in the main file. If the item in the main file has a day of payment *after* the day of payment for the new item, the loop shifts the existing item one space up the file. In this way, as it scans down the file, it carries a spare line with it until the correct location for the new item is found. Note that this technique means that the first element in the main file, element zero, is always left unused, acting as a buffer so that the loop will always know when the beginning of the data has been reached.

Testing

Run the program and call up option 1 from the menu. Enter a new item as follows:

Debit (option 2 on the prompt)
Name: TEST
Amount: 100
Months: FEBRUARY/MAY/AUGUST/NOVEMBER
Day: 15

After a pause you should return to the main menu. Stop the program by using menu option 5. Now type:

print payment$(1),p_month$(1),amount(1,0),amount(1,1)

The result should be:

TEST 010010010010 – 100 15

Module 4.1.4: Formatting a number

Before we can go on to enter the module which prints out the statement of the account, two short modules must be dealt with. The first of them will be used to translate numerical data into a standard format so that it can easily be printed in columns, with decimal points neatly aligned. In addition, we need to overcome the rather annoying limitation that (on early versions at least) the QL kicks into what is called 'scientific notation' for any value less than 0.1. Try PRINT 0.09 on your machine and you will see what I mean — what is displayed on the screen is 9E – 2, or 9 multiplied by 10 to the power – 2. This is a ludicrous situation for a machine aimed to some extent at the business market, since it means the QL cannot be relied on to print out pence (or cents, or centimes, or...) in a normal format. Fortunately there are ways round the problem and this module illustrates one of them.

Module 4.1.4: Lines 8000 – 8170

```
8000 REMark ************************
8010 DEFine FuNction format$ (nn,type)
8020 REMark ************************
8030      LOCal i,n
8040      n=INT(ABS(nn*100)+5E-2)
8050      n$=""
8060      FOR i=6 TO 0 STEP -1
8070        n$=n$ & INT(n/10^i)
8080        n=n-10^i*INT(n/10^i)
8090      NEXT i
8100      FOR i=1 TO 4
8110        IF n$(i)<>"0" THEN EXIT i
8120        n$(i)=" "
8130      END FOR i
8140      n$=n$(1 TO LEN(n$)-2) & "." & n$(LEN(n$)
          -1 TO)
8150      IF type=1 THEN RETurn n$
8160      IF type=2 THEN RETurn n$(4 TO 5)
8170 END DEFine format$
```


Commentary

Line 8030: Since the procedure may be called from within a loop, the loop variable I is declared as a local variable — any changes made to it will not affect its value anywhere else in the program.

Line 8040: The number being sent to the module, NN, is multiplied by 100 to remove any decimal fraction. In addition, since rounding errors were encountered when the module was first used, resulting in figures like 12.99999999999 being produced rather than 13, a tiny decimal fraction is added, and then the integer of the resultant number taken — the effect of this is that the correct whole number is always produced.

Lines 8060 – 8090: Using powers of 10 to divide the number successively, individual digits can be be identified. Thus, if the number were 1234, dividing by 10^3 would produce 1. Subtracting the thousand, which has already been analysed, 234 divided by 10^2 produces 2, and so on. Each digit is stored in N$ as it is extracted. Since the loop runs from 6 to 0 in powers of 10 (1,000,000 to 1), the resulting string will be seven digits long, with leading zeros if the number being worked on is less than 1,000,000.

Lines 8100 – 8130: The number in N$ is scanned to see whether it has any leading zeros. If so, they are replaced with spaces.

Line 8140: N$ is translated back into a number with a decimal fraction by adding a decimal point before the last two characters — in effect, dividing by 100 but doing the operation on a string so that the QL cannot go into scientific notation. In addition, if the original number had no decimal fraction, it will now have '.00' tagged on to the end, ensuring a standardised format. Note, however, that the resultant string can only accurately contain a figure of up to 99,999.99 due to the limitation to seven digits — it could easily be adapted to cope with more than this but the range is sufficient for the current program.

Lines 8150 – 8160: The module will be used to format two types of numbers, the day of payment and actual cash values. What kind of format it will return for a given number will depend on the value sent in the form of the parameter TYPE.

Testing

To extract a number from the module, simply enter:

print format$(XX,type) [ENTER]

where XX is the number you want formatted and TYPE is either 1 or 2. If you set TYPE to 1, you should get an eight character string, including leading spaces and two characters after the decimal point. If TYPE is 2, the format will be two characters, including a leading space for values under 10.

Module 4.1.5: Dealing with negative numbers

One further problem, when it comes to formatting numbers, is raised by negative quantities. It is quite possible simply to place minus signs in front of them on the screen, but this does not stand out and so can lead to incorrect interpretation. A much clearer and more unambiguous method is to print the numbers, or their backgrounds, in red, and this is the solution adopted here when output is made to the screen. On the other hand, the program can also output a statement to the printer, which would not recognise such colour instructions — in this case, a simple minus sign must be used.

Module 4.1.5: Lines 7000 – 7170

```
7000 REMark ************************
7010 DEFine PROCedure colour (flag)
7020 REMark ************************
7030   IF hard=0
7040     IF flag>=0
7050       PAPER #5,6 : INK #5,0
7060     ELSE
7070       PAPER #5,2 : INK #5,7
7080     END IF
7090   END IF
7100   IF hard=1
7110     IF flag>=0
7120       PRINT #5," ";
7130     ELSE
7140       PRINT #5,"-";
7150     END IF
7160   END IF
7170 END DEFine colour
```

Commentary

Lines 7030 – 7090: HARD is the variable which indicates whether printer output is required (hardcopy). This extended IF will only be acted upon when output is to the screen. The number which is about to be printed is sent to this module, which accepts it under the name FLAG. If FLAG is a positive number, the paper colour is set to yellow and the ink to black. If flag is negative, the colour is white ink on red paper. Explanation of the #5 contained in the commands will be kept until the commentary on the following module.

Lines 7110 – 7160: These lines are acted upon in HARD equals 1, indicating that output is to be to the printer. In this case a space is printed in front of positive numbers and a minus sign in front of negative ones.

Testing

Type:

open #5,scr_[ENTER]
colour – 1[ENTER]
print #5,1234[ENTER]

You should find that the number has been printed in white lettering on a red background, since the parameter sent to the procedure was negative. Now try 'colour 1' and you should find the number printed on a yellow background, indicating a positive number. Before moving on, type 'close #5'. The use of channel 5 will be explained shortly.

Module 4.1.6: Displaying the statement

Though there are more modules to come, the final task for the main part of the program is to take the items which have been entered using the previous module and compile them into a statement for any specified month of the year. The statement will include a calculation of the balance carried forward from previous months, and will also display in full all the payments for the month and the continuing balance created by each payment.

Module 4.1.6: Lines 6000 – 6560

```
6000 REMark ************************
6010 DEFine PROCedure statement
6020 REMark ************************
6030   PAPER 6 : CLS : INK 0
6040   INPUT \\" OUTPUT TO PRINTER (Y/N)";hard$
6050   IF hard$="Y" OR hard$="y"
6060     OPEN #5,ser1
6070     hard=1
6080   ELSE
6090     OPEN #5,scr_
6100     hard=0
6110   END IF
6120   REPeat check
6130     INPUT\" NUMBER OF MONTH (1-12): ";mo
6140     mo=mo-1
6150     IF mo>=0 AND mo<=11 THEN EXIT check
6160   END REPeat check
6170   CLS
6180   PRINT#5,"STATEMENT FOR ";mo$(mo)
6190   sum=0
6200   IF mo>0
6210     FOR J=1 TO mo
6220       FOR i=1 TO PA
6230         IF p_month$(i,J)="1"
6240           sum=sum+amount(i,0)
6250         END IF
6260       NEXT i
6270     NEXT J
6280   END IF
6290   PRINT#5,\" BALANCE C/F: ";
```

```
6300    colour sum
6310    PRINT #5,format$(sum,1)
6320    colour 1
6330    IF NOT hard THEN UNDER #5,1
6340    PRINT#5,\"                              ITEM
        TOTAL "\\;
6350    IF NOT hard THEN UNDER#5,0
6360    IF hard THEN PRINT #5," ";
6370    FOR i= 1 TO PA
6380      IF p_month$(i,mo+1)="1"
6390        PRINT #5,format$(amount(i,1),2);" ";
6400        T$=payment$(i) & cl$
6410        PRINT #5,T$(1 TO 15);"   ";
6420        colour amount(i,0)
6430        PRINT #5,format$(amount(i,0),1);" ";
6440        colour 1
6450        sum=sum+amount(i,0)
6460        colour sum
6470        PRINT#5,format$(sum,1)
6480        colour 1
6490        IF hard=0 THEN T$=INKEY$(-1)
6500      END IF
6510    NEXT i
6520    CLOSE #5
6530    AT 18,0
6540      PRINT\"ANY KEY TO RETURN TO MENU"
6550    T$=INKEY$(-1)
6560 END DEFine statement
```

Commentary

Lines 6040 – 6110: You have already had notice that at some point a decision has to be taken as to where the output of the program is going to be sent — to the screen or to the printer. These lines accomplish the choice. Throughout this module all output is made to channel number 5. What these lines do is decide whether that channel is a line of communication to the screen or to the printer. The printer, as we have seen in earlier programs, is accessed through the SER1 port, whereas the screen, when opening a channel, is simply called SCR_. Note that, if you do not have a printer connected, you should not try to output data to SER1 as you will lock up the QL.

Line 6190: The variable SUM will be used to hold the balance in the account — both the balance carried forward and the balance after each item.

Lines 6200 – 6280: Provided that the statement is not for the first month, in which case there is no balance to be carried forward, these two loops scan the whole of the payments list once for each month which precedes the month of the statement. In this way, each payment is examined to see whether it is made in any of the preceding months, in which case the appropriate amount is added to the total in SUM. By the end of the two loops, SUM contains the full total of any changes in the balance since the beginning of the year. (Keeping a complete balance, including any monies which were in the account at the beginning of the year, can be easily achieved by entering the balance from the end of the previous year as a payment on 1st January.)

Lines 6300 – 63320: An example of the use of the two short modules just entered, which will deal with the print colour and formatting of SUM.

Lines 6330 – 6350: Like the colour characteristics, the UNDER characteristic means nothing to the printer and is only sent when the screen is being used.

Lines 6370 – 6510: This loop scans through the complete list of payments, while the extended IF from lines 6380 to 6500 selects only those which have a '1' in the relevant position of the string recording the months in which the payment is to be made. When a payment is to be made in the month specified for the statement, the loop prints out the day, AMOUNT(I,1), the name, PAYMENT$(I), the amount of the payment, AMOUNT(I,0), and finally the balance the payment produces, obtained by adding the amount to the previous total in SUM. Notice that each time a number is to be printed, it is sent to COLOUR and FORMAT$ to ensure that the correct colour is set and that the number is in the correct format. When output is to the screen, a key must be pressed before each item is displayed — this is to prevent the statement scrolling quickly upwards off the screen if there are too many lines of information.

Testing

Run the program and enter some test data. You should find that you get a display something like the example given at the beginning of the program. Try the statement for different periods of the year to ensure that the module can cope with the different months.

Module 4.1.7: Saving data

A standard data storage module.

Module 4.1.7: Lines 9000 – 9180

```
9000 REMark ************************
9010 DEFine PROCedure store
9020 REMark ************************
9030   CLS
9040   AT 1,14 : PRINT "SAVE DATA"
9050   INPUT\\" Name of data file:";file$
9060   tfile$="mdv1_" & file$
```

```
9070    DELETE tfile$
9080    OPEN_NEW #8,"mdv1_" & file$
9090    PRINT#8,PA
9100    FOR i=1 TO PA
9110      PRINT#8,payment$(i)
9120      PRINT#8,p_month$(i)
9130      FOR J=0 TO 1
9140        PRINT#8,amount(i,J)
9150      NEXT J
9160    NEXT i
9170    CLOSE#8
9180 END DEFine store
```

Module 4.1.8: Loading data

A standard data recall module.

Module 4.1.8: Lines 10000 – 10170

```
10000 REMark ************************
10010 DEFine PROCedure recall
10020 REMark ************************
10030   CLS
10040   AT 1,14 : PRINT "RECALL DATA"
10050   DIR mdv1_
10060   INPUT\\" Name of data file:";file$
10070   OPEN_IN #8,"mdv1_" & file$
10080   INPUT#8,PA
10090   FOR i=1 TO PA
10100     INPUT#8,payment$(i)
10110     INPUT#8,p_month$(i)
10120     FOR J=0 TO 1
10130       INPUT#8,amount(i,J)
10140     NEXT J
10150   NEXT i
10160   CLOSE#8
10170 END DEFine store
```

Module 4.1.9: Changing and deleting items

As in Nnumber, a simple user search module, which allows the user to scan backwards and forwards or delete an item, though the delete function will not be available until the next module has been entered.

Module 4.1.9: Lines 4000 – 4340

```
4000 REMark ************************
4010 DEFine PROCedure search
4020 REMark ************************
4030   count=1
4040   REPeat display
4050     CLS
4060     PAPER 0
```



```
4070     BLOCK 448,90,0,0,0
4080     PRINT\"PAYMENT: ";payment$(count)
4090     PRINT "AMOUNT: ";amount(count,0)
4100     PRINT"MONTHS: ";
4110     FOR i=1 TO 12
4120       IF p_month$(count,i)="1"
4130         PRINT !mo$(i-1)!
4140       END IF
4150     NEXT i
4160     PRINT\"DAY OF PAYMENT: ";amount(count,1)
4170     PAPER 2
4180     AT 10,0
4190     PRINT " COMMANDS AVAILABLE:"
4200     PRINT\"    'ENTER' > NEXT ENTRY"
4210     PRINT "    'DDD' > DELETE ENTRY"
4220     PRINT "    '#' THEN NUMBER > MOVE"
4230     PRINT "    'ZZZ' > QUIT"
4240     INPUT\" WHICH DO YOU REQUIRE: ";Q$
4250     IF Q$="ddd" OR Q$="DDD" THEN remove
4260     IF Q$="zzz" OR Q$="ZZZ" THEN EXIT display
4270     IF Q$="" THEN Q$="#1"
4280     IF Q$(1)="#"
4290       count=count+Q$(2 TO)
4300     END IF
4310     IF count>PA THEN count=PA
4320     IF count<1 THEN count=1
4330   END REPeat display
4340 END DEFine search
```

Testing

Though you cannot delete items, you should be able to page backwards and forwards through any material you have in memory.

Module 4.1.10: Deleting items

A standard delete module, collapsing the file down on to the deleted item.

Module 4.1.10: Lines 5000 – 5100

```
5000 REMark ************************
5010 DEFine PROCedure remove
5020 REMark ************************
5030   FOR i=count TO PA
5040     payment$(i)=payment$(i+1)
5050     p_month$(i)=p_month$(i+1)
5060     amount(i,0)=amount(i+1,0)
5070     amount(i,1)=amount(i+1,1)
5080   NEXT i
5090   PA=PA-1
5100 END DEFine remove
```

Testing

You should now be able not only to page through your material but to delete items at will. The program is now complete and ready to use.

PROGRAM 4.2: ACCOUNTANT

Program function

The second program in this final chapter is more complex than Banker. Its function is to keep two sides of a simple set of accounts, setting them out in the traditional format, with some items standing alone and others clearly divided into groups representing different types of expenditure. Separate screens are produced for the credit and debit side of the accounts, with the overall balance of the account displayed.

```
                    DEBIT

HOUSE
  MORTGAGE                  250.00
  RATES                      98.45
                            -------
                                        348.45
SINCLAIR QL                             399.00
HOLIDAY                                 456.21
QL BOOKS                                 29.90
                                        -------
TOTAL:                                  1233.56

CREDIT BALANCE IS       0.00

OVERALL BALANCE IS - 1233.56

PRESS ANY KEY TO QUIT
```

Figure 4.2: Account Prepared by Accountant.

Module 4.2.1: Initialisation

A standard initialisation module.

Module 4.2.1: Lines 1000 – 1070

```
1000 REMark ************************
1010 REMark initialise
1020 REMark ************************
1030   PAPER 4 : INK 0
1040   CLS : CLS#0
1050   DIM account$(1,499,15),account(1,499),
       cand(1)
1060   INPUT\"LOAD FROM MICRODRIVE (Y/N): ";0$
1070   IF 0$="Y" OR 0$="y" THEN recall
```

Commentary

Line 1050: The two sides of the accounts, credit and debit, including the

names associated with each payment, are stored in the two sides of the arrays ACCOUNT and ACCOUNT$. Up to 500 items can be stored on both sides, although it would be be quite feasible to increase the size of the arrays if more space were required.

Module 4.2.2: The main menu

A standard menu module.

Module 4.2.2: Lines 2000 – 2410

```
2000 REMark ************************
2010 REMark menu
2020 REMark ************************
2030   REPeat menu
2040     PAPER 4 : INK 0
2050     CLS
2060     AT 1,13
2070     UNDER 1
2080     PRINT "ACCOUNTANT"
2090     UNDER 0
2100     PRINT\\\"COMMMANDS AVAILABLE: "
2110     PRINT\"    1) INPUT NEW HEADINGS"
2120     PRINT "    2) CHANGE/DELETE ITEMS"
2130     PRINT "    3) PRINT ACCOUNTS"
2140     PRINT "    4) DATA FILES"
2150     PRINT "    5) STOP"
2160     INPUT\"WHICH DO YOU REQUIRE: ";Z
2170     IF cand(0)=0 AND cand(1)=0 AND (Z=2 OR
         Z=3 OR Z=4)
2180       PRINT\\"       SORRY, NO DATA YET"
2190       Z=0
2200       T$=INKEY$(-1)
2210     END IF
2220     SELect ON Z
2230       ON Z=1 : TYPE : heading
2260       ON Z=2 : TYPE : search
2290       ON Z=3 : TYPE : output
2320       ON Z=4 : store
2340       ON Z=5
2350         CLS
2360         AT 10,13
2370         PRINT "ACCOUNTANT"
2380         PRINT #0,"Program Terminated"
2390         STOP
2400     END SELect
2410   END REPeat menu
```

Module 4.2.3: Credit or debit?

Unlike Banker, several parts of this program need to know whether a credit

or debit item is being specified, so the routine to request this information is included in a separate module, the item type being recorded in the variable CORD (C or D).

Module 4.2.3: Lines 3000 – 3110

```
3000 REMark ************************
3010 DEFine PROCedure TYPE
3020 REMark ************************
3030   REPeat check
3040     PRINT\\" 1) CREDIT"\" 2) DEBIT"
3050     INPUT\"WHICH IS IT: ";cord
3060     IF cord=1 OR cord=2 THEN EXIT check
3070   END REPeat check
3080   cord=cord-1
3090   cord$="CREDIT"
3100   IF cord=1 THEN cord$="DEBIT"
3110 END DEFine TYPE
```

Module 4.2.4: The type of item

This module is trivial in itself but it gives a clue as to why this program is bound to be longer than something like Banker. The purpose of the module is to allow the user to specify which of three types an item about to be input falls under. The three types are:

1) **A single item:** All that is required for this is the name of the item and the amount. When the eventual account is printed out, individual items will have their names printed on the lefthand side and the amount associated in the main column of figures on the right.

2) **A main heading:** It is this type which allows groups of items to be specified within the overall account. If you were using the program to prepare domestic accounts, for instance, you might set up 'CAR' as a main heading for a group of items including items like tyres, fuel, repairs and so on. In the eventual account, the name of the main heading will be printed on the lefthand side but there will be no amount printed against the main heading itself.

3) **Subheadings:** As illustrated under (2) above, each main heading can have a list of items following it, which are part of a separate group. In the accounts, the names of subheadings will be printed under their relevant main heading, inset from the left, while the amount associated with each subheading will be printed to the left of the main column of figures.

Module 4.2.4: Lines 4000 – 4200

```
4000 REMark ************************
4010 DEFine PROCedure heading
4020 REMark ************************
```



```
4030   REPeat h_loop
4040     CLS
4050     AT 1,14 : PRINT "NEW ITEMS"
4060     PRINT\\cord$
4070     PRINT\\"Is the item:"
4080     PRINT\"    1) A single item"
4090     PRINT "    2) A main heading"
4100     PRINT "    3) A sub-heading"
4110     PRINT "    (Input '0' to quit)"
4120     INPUT\"Please specify: ";h_type
4130     SELect ON h_type
4140       ON h_type=0 : EXIT h_loop
4150       ON h_type=1 : single
4160       ON h_type=2 : single
4170       ON h_type=3 : sub_head
4180     END SELect
4190   END REPeat h_loop
4200 END DEFine heading
```

Testing

Having entered all the parts of the program which do not employ any calculation, it is probably best if you run the program and quickly test the menu. If you specify that you wish to input a new item, you should be asked whether it is a credit or a debit, and then asked to specify the type — though that is as far as you can go. The only other menu function which will have any effect is option 5 to stop the program. The menu itself should stop you from accessing the functions to alter data or print the accounts, since no data has yet been entered.

Module 4.2.5: Entry of single items and main headings

Two separate modules take care of the input of subheadings on the one hand (see the next module), and single items or main headings on the other. It is important, in understanding later parts of the program, that you try to follow the way in which the items are stored and the special indicator characters which record the item type.

Module 4.2.5: Lines 5000 – 5210

```
5000 REMark ************************
5010 DEFine PROCedure single
5020 REMark ************************
5030   INPUT\"Name of item: ";name$
5040   IF h_type=1
5050     INPUT "Amount for item: ";amount
5060   END IF
5070   INPUT "Is this correct (Y/N): ";Q$
5080   IF Q$<>"Y" AND Q$<>"y"
5090     PRINT\"NOT REGISTERED"
5100     T$=INKEY$(-1)
```

```
5110      RETurn
5120    END IF
5130    IF h_type=1
5140      name$="%" & name$
5150    ELSE
5160      name$="*" & name$
5165      amount=0
5170    END IF
5180    account$(cord,cand(cord))=name$
5190    account(cord,cand(cord))=amount
5200    cand(cord)=cand(cord)+1
5210 END DEFine single
```

Commentary

Lines 5040 – 5060: As mentioned in the introduction to the previous module, main headings have no money figure associated with them, so these lines accept a figure only for single items.

Line 5130 – 5170: There are no separate storage areas for the different types of item, apart from the credit and debit sides of the arrays. Later parts of the program will determine the item type by looking at a special indicator character attached to the beginning of the item name. This will be '%' for a single item and '*' for a main heading.

Lines 5180 – 5200: You have already met the variable CORD, which records whether an item is a credit or a debit. Here CORD is used to decide on which side of the arrays ACCOUNT and ACCOUNT$ the new item is to be placed. In addition, we need to keep a record of the number of items on the credit and debit sides, since these will normally be different. This is done by the array CAND (C and D). The array was declared in the initialisation module and has only two elements, CAND(0) and CAND(1), corresponding to the credit and debit sides of the main arrays. Once again, the value of CORD is used to indicate which of the two elements of CAND is to be used. Applying this, we can see that when reference is made to:

```
ACCOUNT (CORD,CAND(CORD))
      1        2        3
```

what is meant is:

1) An element in the numeric array ACCOUNT.
2) On the side indicated by the value of CORD, ie credit or debit.
3) The first empty element on that side, determined by what is already stored.

Testing

Run the program and call up the new entry option. Specify that you wish to enter a main heading on the credit side, then enter TEST MAIN for the item name — no value should be requested. Now specify a single item on

the credit side, with the name TEST and the value 100. Now do exactly the same thing but on the debit side of the accounts.

Stop the program from the menu and enter, in direct mode:

```
print account(0,0),account(1,0),account$(0,0),account$(1,0)[RETURN]
```

You should see:

```
0        0        *TEST MAIN        *TEST MAIN
```

Now perform the same procedure for line 1 of the arrays, eg ACCOUNT(0,1) etc. You should see:

```
100      100      %TEST      %TEST
```

Finally, print out the value of CAND(0) and CAND(1) — both should equal 2.

Module 4.2.6: Entering a subheading

The question of entering a new subheading is not quite as simple as that for a single item. For each new subheading that is entered, a check has to be made for the presence of the relevant main heading and the item placed next to its main heading rather than simply tagged on to the end of the items previously stored.

Module 4.2.6: Lines 6000 – 6260

```
6000 REMark ************************
6010 DEFine PROCedure sub_head
6020 REMark ************************
6030   REPeat check
6040     INPUT "Main heading: ";search$
6050     search$="*" & search$
6060     FOR p1=0 TO cand(cord)-1
6070       IF account$(cord,p1)=search$ THEN
           EXIT p1
6080     NEXT p1
6090       PRINT "Heading not found."
6100       T$=INKEY$(-1)
6110       RETurn
6120     END FOR p1
6130     INPUT "Name of sub-heading: ";name$
6140     INPUT "Amount: ";amount
6150     INPUT\"Are these correct (Y/N): ";Q$
6160     IF Q$="Y" OR Q$="y" THEN EXIT check
6170   END REPeat check
6180   name$="$" & name$
6190   FOR i=cand(cord)+1 TO p1+2 STEP -1
6200     account$(cord,i)=account$(cord,i-1)
```

```
6210      account(cord,i)=account(cord,i-1)
6220    NEXT i
6230    account$(cord,pl+1)=name$
6240    account(cord,pl+1)=amount
6250    cand(cord)=cand(cord)+1
6260 END DEFine sub_head
```

Commentary

Lines 6040 – 6120: The name of the relevant main heading is requested and a check is made of the items already in the file that the heading actually exists — if not, an error message is printed and the program returns to the menu. Note the use here of both NEXT and END FOR, to provide a section of program which is only executed if the loop finishes normally. If an object is found, the EXIT command jumps to the END FOR, *not* to the NEXT.

Lines 6190 – 6250: As previously mentioned, the whole point of a subheading is that it should appear in the main accounts as part of a group printed under the relevant main heading. In order to achieve this simply, the method employed is to store it in the file next to its main heading. The position of the first item following the main heading has already been found by the FOR loop at line 6060, so all that needs to be done is to move up all the items above that point and place the new item into the array directly after its main heading — note that this means that the latest subheading is always the first item under its relevant main heading.

Testing

Input the items specified for the test of Module 4.2.5, then call up the new entry module again to place a new subheading on the credit side, named TEST SUB, with a value of 200. Do the same for the debit side.

Enter the following in direct mode:

for i = 0 to 2:print account(0,i),account(1,i),account$(0,i), account$(1,i): next i [RETURN]

You should see:

0	0	*TEST MAIN	*TEST MAIN
200	200	$TEST SUB	$TEST SUB
100	100	%TEST	%TEST

Print out the values of CAND(0) and CAND(1) — these should both be 3.

Module 4.2.7: Data files — store

Since the data for Accountant is fairly complex to use, it is probably wise to enter the data file module at this point to eliminate the need for constant

re-entry of data when testing. Once the module has been entered, enter and save the data specified for the test of the previous module. Both modules are completely standard.

Module 4.2.7: 11000 – 11170

```
11000 REMark ************************
11010 DEFine PROCedure store
11020 REMark ************************
11030   CLS
11040   AT 1,14 : PRINT "SAVE DATA"
11050   INPUT\\" Name of data file:";file$
11060   tfile$="mdv1_" & file$
11070   DELETE tfile$
11080   OPEN_NEW #8,"mdv1_" & file$
11090   FOR side=0 TO 1
11100     PRINT#8,cand(side)
11110     FOR i=0 TO cand(side)-1
11120       PRINT#8,account$(side,i)
11130       PRINT#8,account(side,i)
11140     NEXT i
11150   NEXT side
11160   CLOSE#8
11170 END DEFine store
```

Module 4.2.8: Data files — recall

Module 4.2.8: 12000 – 12160

```
12000   REMark ************************
12010   DEFine PROCedure recall
12020   REMark ************************
12030   CLS
12040   AT 1,14 : PRINT "RECALL DATA"
12050   DIR mdv1_
12060   INPUT\\" Name of data file:";file$
12070   OPEN_IN #8,"mdv1_" & file$
12080   FOR side=0 TO 1
12090     INPUT#8,cand(side)
12100     FOR i=0 TO cand(side)-1
12110       INPUT#8,account$(side,i)
12120       INPUT#8,account(side,i)
12130     NEXT i
12140   NEXT side
12150   CLOSE#8
12160 END DEFine recall
```

Module 4.2.9: Changes to items

A standard module with some added features to take account of the fact that some items do not stand alone but as part of groups of items under a common main heading.

Module 4.2.9: Lines 7000 – 7420

```
7000 REMark ************************
7010 DEFine PROCedure search
7020 REMark ************************
7030   count=0
7040   REPeat display
7050     CLS
7060     PAPER 0 : INK 7
7070     BLOCK 448,70,0,0,0
7080     temp$=account$(cord,count)
7090     IF temp$(1)<>"$" THEN PRINT\\temp$(2 TO)
7100     IF temp$(1)="*" THEN hh$=temp$(2 TO)
7110     IF temp$(1)="$"
7120       PRINT\\hh$
7130       PRINT\temp$(2 TO)
7140     END IF
7150     IF temp$(1)<>"*" THEN PRINT format$
         (account(cord,count))
7160     PAPER 4 : INK 0
7170     AT 8,1
7180     PRINT " COMMANDS AVAILABLE: "
7190     PRINT\"    'ENTER' > NEXT ITEM"
7200     PRINT "    'CCC' > CHANGE AMOUNT"
7210     PRINT "    '#' NUMBER > MOVE POINTER"
7220     PRINT "    'DDD' > DELETE ITEM"
7230     PRINT "    'ZZZ' > RETURN TO MENU"
7240     INPUT\" WHICH DO YOU REQUIRE: ";Q$
7250     IF Q$="DDD" OR Q$="ddd" THEN REMOVE
7260     IF Q$="ZZZ" OR Q$="zzz" THEN EXIT display
7270     IF Q$="" THEN Q$="#1"
7280     IF Q$(1)="#"
7290       count=count+Q$(2 TO)
7300     END IF
7310     IF count>cand(cord)-1 THEN count=cand
         (cord)-1
7320     IF count<0 THEN count=0
7330     IF Q$="ccc" OR Q$="CCC"
7340       INPUT "Amount to be added: ";amount
7350       INPUT "Is that correct: ";r$
7360       IF r$="y" OR r$="Y"
7370         account(cord,count)=account(cord,
             count)+amount
7380       END IF
7390     END IF
7400     IF cand(cord)=0 THEN EXIT display
7410   END REPeat display
7420 END DEFine search
```

Commentary

Lines 7060 – 7070: For the purposes of the search module, the upper part of the screen, on which the items will be printed, will be coloured black, and the simplest way to accomplish this is to have the QL draw a block over the top of the screen.

Lines 7090 – 7150: If the item recalled from the file is a single item, then it is printed — though stripped of the indicator character which is tagged on to the beginning of the name. If the item is a main heading, not only is it printed, but its name is stored in HH$ so that it can be printed out above any of its subheadings which follow. If the item is a subheading, HH$ is printed, followed by the subheading, stripped of its indicator character.

Lines 7330 – 7390: Apart from deleting items, changes can be made to the value associated with a heading. This is done by entering a positive or negative figure by which the value of an item may be changed — not an absolute value which the item is to take. The advantage of this is that most changes will result in the need to add amounts to existing items as further expenditures or receipts are made under items which already exist. Thus, if an extra £100 is to be spent on car repairs, for which there is already a heading, all that needs to be done is to page through the file to that heading and enter '100'.

Testing

Run the program and call up the data which you have previously stored on microdrive. Now call up option 2 from the menu and check that you can page through the three items, finally returning to the menu again. Call up option 2 again, and this time try adding to or subtracting from the two totals you have previously entered. Paging through the items again should reveal that you have successfully altered their values.

Module 4.2.10: Deleting items

One final facility to be added in relation to existing items is deletion. In the case of Accountant, the deletion module is more complex than previous examples of the type. The reason for this is the existence of the groups formed around main headings. While there are no difficulties associated with the deletion of a single item or a subheading, what happens when a main heading is deleted? The answer, obviously, is that not only has the main heading to be taken out, but also all the subheadings associated with it — otherwise the account would become clogged with subheadings not attached to a main heading, making nonsense of the account.

Module 4.2.10: Lines 8000 – 8200

```
8000 REMark ************************
8010 DEFine PROCedure REMOVE
8020 REMark ************************
8030   pl=count : group=1
8040   IF account$(cord,pl,1)="*"
8050     REPeat d_loop
8060       IF account$(cord,pl+group,1)="$"
```

```
8070          group=group+1
8080        ELSE
8090          EXIT d_loop
8100        END IF
8110      END REPeat d_loop
8120    END IF
8130    FOR k=p1 TO cand(cord)-group
8140      account(cord,k)=account(cord,k+group)
8150      account$(cord,k)=account$(cord,k+group)
8160    NEXT k
8170    cand(cord)=cand(cord)-group
8180    account$(cord,cand(cord))=""
8190    account(cord,cand(cord))=0
8200 END DEFine REMOVE
```

Commentary

Line 8030: The position at which the deletion is to take place is sent from the previous module in the form of the variable COUNT. This is transferred for the purposes of the current module to PL. The variable GROUP records how many items need to be deleted. It is initially set to 1, and will only be increased if the item to be deleted is a main heading with subheadings attached.

Lines 8040 – 8120: These lines will only be activated if the item specified for deletion is a main heading. The loop scans down the following entries, counting how many of those items which follow are preceded by a '$', indicating that they are sub-items for the main heading. The result of the count is kept in GROUP.

Lines 8130 – 8160: A typical loop to collapse an array and delete an item. The difference here is that instead of copying item X into space X – 1, and therefore copying each element down one place, items are transferred GROUP places, thus deleting GROUP items, or the number of items in the group based around a main item.

Testing

Following the test procedure for the previous module, you should not only be able to page through the items and alter them, you should also be able to delete items. If you delete the item labelled MAIN TEST, you should find that SUB-TEST disappears with it.

Module 4.2.11: Formatting a number

With the exception that this module does not need to be able to format dates in two character format, this module is identical to that used and commented on in Banker.

Module 4.2.11: Lines 10000 – 10160

```
10000 REMark ************************
10010 DEFine FuNction format$ (nn)
10020 REMark ************************
10030     LOCal i,n
10040     n=INT(ABS(nn*100)+5E-2)
10050     n$=""
10060     FOR i=6 TO 0 STEP -1
10070       n$=n$ & INT(n/10^i)
10080       n=n-10^i*INT(n/10^i)
10090     NEXT i
10100     FOR i=1 TO 4
10110       IF n$(i)<>"0" THEN EXIT i
10120       n$(i)=" "
10130     END FOR i
10140     n$=n$(LEN(n$)-6 TO LEN(n$)-2) & "."
          & n$(LEN(n$)-1 TO)
10150     RETurn n$
10160   END DEFine format$
```

Module 4.2.12: Displaying the accounts

After all the preparation, the one module which makes sense of the whole thing, by displaying the account in its final form. Like the equivalent module in Banker, it looks complex, but, once you have seen the display, you will quickly see why everything is arranged as it is. Note that, unlike Banker, this module does not make provision for a printer output. Given the simplicity of the print output and the absence of colour controls it would be a simple matter to copy the techniques used in Banker across to the program if you require a hardcopy.

Module 4.2.12: Lines 9000 – 9460

```
9000 REMark ************************
9010 DEFine PROCedure output
9020 REMark ************************
9030   CLS
9040   tt=0 : ss=0
9050   AT 1,15 : PRINT cord$
9060   FOR i=0 TO cand(cord)-1
9070     IF account$(cord,i,1)="*" THEN PRINT
9080     IF account$(cord,i,1)="$" THEN PRINT
         "  ";
9090     PRINT account$(cord,i,2 TO);
9100     IF account$(cord,i,1)="*" THEN PRINT
9110     IF account$(cord,i,1)<>"*"
9120       tt=tt+account(cord,i)
9130       PRINT "    ";
9140       IF account$(cord,i,1)="%" THEN PRINT
           "              ";
9150       PRINT format$ (account(cord,i))
```

```
9160        IF account$(cord,i,1)="$" THEN ss=ss+
            account(cord,i)
9170      END IF
9180      IF ss<>0 AND account$(cord,i+1,1)<>"$"
9190        PRINT "                              --------"
9200        PRINT "                                      ";
9210        PRINT format$(ss)
9220        ss=0
9230      END IF
9240      T$=INKEY$(-1)
9250    NEXT i
9260    PRINT "                                    ----
        ---"
9270    PRINT "TOTAL:                             ";
9280    PRINT format$(tt)
9290    t2=0
9300    FOR i=0 TO cand(1-cord)
9310      IF account$(1-cord,i,1)<>"*"
9320        t2=t2+account(1-cord,i)
9330      END IF
9340    NEXT i
9350    IF cord=1
9360     cord2$="CREDIT"
9370    ELSE
9380     cord2$="DEBIT"
9390    END IF
9400    PRINT\cord2$;" BALANCE IS ";format$(t2)
9410    PRINT\"OVERALL BALANCE IS ";
9420    IF (tt-t2)*(1-2*(cord=1))<0 THEN PRINT "-";
9430    PRINT format$(tt-t2)
9440    PRINT\"PRESS ANY KEY TO QUIT"
9450    T$=INKEY$(-1)
9460 END DEFine account
```

Commentary

Line 9040: The variable TT will be used to store the running total for the account as it is printed. SS will hold sub-totals for groups of sub-items.

Lines 9070 – 9090: These lines print the item name. For a main item, one blank line is printed first to separate it from what has gone before, while for a subheading, the two spaces inset the item name by two spaces.

Lines 9130 – 9150: These lines deal with the printing of the amounts for single items and subheadings. Subheadings will be printed at the twenty-first position along the line, single items at position 30. If the item being dealt with is a subheading, then the sub-total for the items in the current group is stored temporarily in SS.

Lines 9180 – 9230: This IF operates only when a group is being processed, as indicated by the fact that SS is not equal to 0, and the next item is not part of the group — ie the group is complete. The effect of the loop is to print the total for the group in the main column of figures at position 30 along the line.

Line 9240: Only one item is printed at a time. This is to prevent the top part of the account scrolling off the top of the screen before it can be read. For each fresh item, a key must be pressed.

Lines 9290 – 9340: These lines scan the opposite side of the accounts to the one currently being printed and obtain a total of the figures in the variable T2.

Lines 9350 – 9430: The final items to be output are the total of the opposite side of the account and the balance of the account.

Testing

Reload the data you have previously stored on disk and simply call up option 3 on the main menu. Remember that to print out the whole account you will need to press a key for each item — the whole account will not appear immediately. If this test is successfully performed, the program is ready for use.

PROGRAM 4.3: BUDGET

Program function

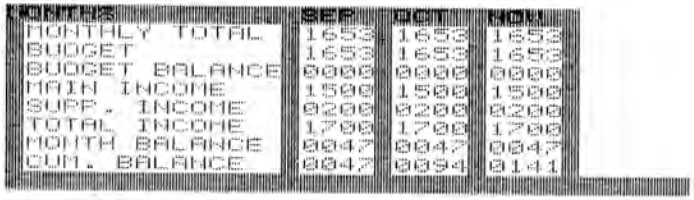
Budget, the final program in this book, is, for most readers, one of the largest programs they will ever enter (or *want* to enter for that matter). From letters I receive about earlier and less capable versions, however, entering the program seems to be a worthwhile chore, given the results.

There is nothing terribly complex about Budget, it's just that it is designed to carry a great deal of data and perform a range of calculations on this data which help reveal a household financial picture over a rolling 12 month period. As I said at the beginning of this chapter, one of the strengths of the micro is its ability to present figures in an orderly and comprehensible fashion. Budget illustrates well a second strength, and that is that, when large bodies of data are being handled, it becomes possible to perform fairly straightforward calculations which would defeat most people working manually and yet which, when performed, open up new and enlightening ways of looking at the data.

The purpose of Budget is to allow the user to input figures for income and up to 60 monthly payments (regular or irregular) and to have an analysis performed upon them which will reveal the monthly balance of income over expenditure, the cumulative balance over the year, the average budget allocation needed to meet commitments over the year and any shortfall, from month to month, of the average budget payment compared to actual payments to date. In other words, Budget presents a straightforward picture of how the user's finances will appear over the year ahead. In addition, however, Budget is designed to be a 'what if' tool,

which allows the user to speculate about the effects of changes through the year without corrupting the original data. This is achieved by means of a set of 'shadow arrays' into which entries can be made, and the same analysis carried out, with no effect on the original data. To simplify the matter of entry, the whole of the current data on the 'real' side of the arrays can be copied into the 'hypothetical' side with a single keystroke, so that only changes relative to the real data have to be entered.

When all is said and done, really the only way to understand the attractions of Budget is to enter it and use it. An hour's playing with the program will demonstrate just how much it can tell you about your finances.



MONTHLY TOTAL	1653	1653	1653
BUDGET	1653	1653	1653
BUDGET BALANCE	0000	0000	0000
MAIN INCOME	1500	1500	1500
SUPP. INCOME	0200	0200	0200
TOTAL INCOME	1700	1700	1700
MONTH BALANCE	0047	0047	0047
CUM. BALANCE	0047	0094	0141

Figure 4.3: Part of Analysis Screen from Budget.

Module 4.3.1: Initialisation

The size of this module should be sufficient to convince you of the complexity of the program. The use of the arrays and variables will be described during the course of the program commentary.

Module 4.3.1: Lines 10000 – 10340

```
10000 REMark ************************
10010 REMark initialise
10020 REMark ************************
10030   INK 0 : PAPER 4 : CLS
10040   DIM payment$(1,59,14)
10050   DIM payments(1,59,11)
10060   DIM monthly(1,59)
10070   DIM paytotals(1,11)
10080   DIM b_balance(1,11)
10090   DIM balance(1,11)
10100   DIM income_1(1,11)
10110   DIM income_2(1,11)
10120   DIM month$(11,9)
10130   DIM y_budget(1)
10140   DIM items(1)
10150   real=0
10160   RESTORE
```



```
10170   FOR i=0 TO 11
10180     READ month$(i)
10190     month$(i)=month$(i) & "         "
10200   NEXT i
10210   INPUT\\" Load from microdrive (Y/N): ";q$
10220   IF q$="y" OR q$="Y"
10230     recall
10240   ELSE
10250     REPeat check
10260       INPUT\\" Current month number: ";q
10270       IF q>=1 AND q<=12 THEN EXIT check
10280     END REPeat check
10290     curr_m=q-1
10300     year=curr_m+11
10305   test_data:GO TO 11000
10310     income
10320     new_headings
10330   END IF
10340   DATA "JANUARY","FEBRUARY","MARCH","APRIL",
        "MAY","JUNE","JULY","AUGUST","SEPTEMBER",
        "OCTOBER","NOVEMBER","DECEMBER"
```

Module 4.3.2: The program menu

A standard menu module.

Module 4.3.2: Lines 11000 – 11470

```
11000 REMark ************************
11010 REMark menu
11020 REMark ************************
11030   REPeat loop
11040     PAPER (4+real)
11050     CLS
11060     AT 1,16 : PRINT "BUDGET"
11070     PRINT\\"Commands available:"
11080     PRINT\\"   1) Display monthly analysis"
11090     PRINT "   2) Changes"
11100     PRINT "   3) New budget headings"
11110     PRINT "   4) Delete budget heading"
11120     PRINT "   5) Reset hypothetical data"
11130     PRINT "   6) Reset month"
11140     PRINT "   7) Store data"
11150     PRINT "   8) Change to ";
11160     IF real=0
11170       PRINT "hypothetical";
11180     ELSE
11190       PRINT "real";
11200     END IF
11210     PRINT " data"
11220     PRINT "   9) Stop"
11230     INPUT\\"Which do you require: ";z
11240     SELect ON z
11250       ON z=1 : display
11270       ON z=2 : changes
```

```
11290       ON z=3 : new_headings
11310       ON z=4 : remove
11330       ON z=5 : reset_data
11350       ON z=6 : reset_month
11370       ON z=7 : store
11390       ON z=8 : real=1-real
11410       ON z=9
11420         CLS
11430         AT 10,16 : PRINT "BUDGET"
11440         PRINT #0,"Program terminated"
11450         STOP
11460       END SELect
11470     END REPeat loop
```

Commentary

Line 11040: The variable REAL will be used to indicate whether the real or hypothetical data is being worked on at the moment. From the user's point of view, this is made plain by the fact that when working with real data the screen will be green, and when working with the hypothetical figures this will change to cyan.

Lines 11150 – 11210: Just a small touch, this, but a nice one nevertheless. Since the program can only change between the two states of dealing with real and hypothetical data, this particular option on the menu indicates which of the two will be accessed by choosing option 8, eg if the program is currently in the real data mode, the prompt will read:

Change to hypothetical data

and vice versa.

Module 4.3.3: Short functions

In the course of this program, we shall be calling on a number of trivial functions or procedures. Rather than dignify them all with a module of their own, we shall adopt the expedient of a single module consisting of four short sections. They will be briefly commented on here, but their real usefulness will only become apparent when they are used.

Module 4.3.3: Lines 25000 – 25260

```
25000 REMark ************************
25010 REMark short functions
25020 REMark ************************
25030   DEFine FuNction i_m
25040     LOCal i_m
25050     i_m=i-12*(i>11)
25060     RETurn i_m
25070   END DEFine i_m
25080   :
25090   DEFine PROCedure table (temp$)
```



```
25100     space : PRINT temp$; : space
25110     PRINT
25120   END DEFine
25130   :
25140   DEFine PROCedure table2 (temp)
25150     OVER 1
25160     PRINT FILL$(" ",16+5*(i-start));
25170     OVER 0
25180     form_print (temp) : space
25190     PRINT
25200   END DEFine table2
25210   :
25220   DEFine PROCedure space
25230     PAPER 0
25240     PRINT " ";
25250     PAPER real+4
25260   END DEFine space
```

Commentary

Lines 25030 – 25070: This function is intended to cope with the problem which arises when the start of the year, as the program sees it, does not correspond to the start of the calendar year. If the program is currently starting from August and looking 12 months ahead, problems will be encountered in program loops which scan the year, when the transition is made from December to January, a move from month 5 to 6 as far as the program is concerned but from 12 to 1 (or rather 11 to 0) as far as any data goes which is stored in arrays from January to December. The simple answer adopted here is that all loops in this program completely ignore the transition from December to January. If they are to start in December, they count 12, 13, 14, etc. This function is given the job of providing a variable to replace the loop variable, which is assumed to be I, with another which will count 12, 1, 2, etc.

Lines 25090 – 25120: A formatting aid. When it comes to printing out a table, this procedure will accept a heading and print an inverse space immediately before and after it, thus facilitating building up columns. To do this it calls up SPACE, described below.

Lines 25140 – 25200: In the absence of a TAB facility in SuperBASIC, a good simulation can be arrived at with OVER. This module will print a line of spaces on the current printing line, with OVER set. This has the effect of moving the print position to the right without corrupting anything which is already on the screen. The module assumes that it is being called from a loop with the loop variable I, and the position on the screen will depend on the value of two variables, I and START.

Lines 25220 – 25260: These lines print an inverse or black space used in laying out tables.

Testing

1) If you set the value of I directly by entering:

i = x [ENTER] (where X is a number between 0 and 23)

you should find that PRINT I__M will print out the value of I, less 12 if I was greater than 11.

2) Type:

real = 0 [ENTER]
table ("Name") [ENTER]

you should see the word echoed back with an inverse space before and after (this is also a sufficient test of SPACE).

3) Type:

start = 1[ENTER]
i = 2[ENTER]
print:table2 (123)[ENTER]

and you should see '0123' printed near the centre of the current printing line.

Module 4.3.4: Entering figures for income

This is a straightforward module which stores income under two headings, main and additional, in the arrays INCOME__1 and INCOME__2.

Module 4.3.4: Lines 20000 – 20160

```
20000 REMark ************************
20010 DEFine PROCedure income
20020 REMark ************************
20030   CLS
20040   AT 1,13 : PRINT "CHANGE INCOME"
20050   PRINT\" Input '\' to leave unchanged"\\
20060   FOR i=curr_m TO year
20070     INPUT\(month$(i_m) & ": Main (" &
          income_1(real,i_m) & "): ");q$
20080     IF q$<>"\"
20090       income_1(real,i_m)="0" & q$
20100     END IF
20110     INPUT (month$(i_m) & ": Additional ("
          & income_2(real,i_m) & "): ");q$
20120     IF q$<>"\"
20130       income_2(real,i_m)="0" & q$
20140     END IF
20150   NEXT i
20160 END DEFine income
```


Commentary

Line 20060: CURR__M is the variable which holds the number within the calendar year of the current month. YEAR is simply CURR__M plus 11.

Lines 20070 – 20140: The module is later used when resetting the program to start at a new month, so provision is made for the input of \ (a key conveniently near ENTER) to leave a figure unchanged. Otherwise, the user is prompted with the month name and asked to supply the two income figures for that month. Note the use of I__M to translate the I loop, which may run from 11 to 22, into a value in the range 0 – 11. Apart from I__M, the position in which the data will be stored is determined by the value of REAL, which dictates whether the real or hypothetical halves of the arrays are to be accessed.

Testing

Run the program, answering N to the prompt to load from microdrive, and you should be prompted for 24 income figures. When you have finished entering them, the program will stop with a BAD NAME error as it tries to call a procedure you have not entered yet. Type:

for i = 0 to 11:print income__1(0,i),income__2(0,i):next i

You should see the income figures you have entered displayed.

Module 4.3.5: Entering payment headings

Just as the program needs to know the income figures when it starts up, it is hardly going to be much use without some information as to the payments to be made. This procedure can be called at any time when the main program has been entered but is always called first when the program is RUN and data not loaded from tape.

Module 4.3.5: Lines 16000 – 16270

```
16000 REMark ************************
16010 DEFine PROCedure new_headings
16020 REMark ************************
16030   REPeat nh_loop
16040     CLS
16050     AT 1,6 : PRINT "INPUT OF NEW PAYMENTS"
16060     PRINT\\" Precede name with a '*' if
          you do"!"not wish it to be budgeted."
16070     PRINT\"Enter 'ZZZ' as name to quit."
16080     INPUT\"Name for new payment: ";q$
16090     IF q$="zzz" OR q$="ZZZ"
16100       update
16110       EXIT nh_loop
16120     END IF
16130     IF items(real)=60
```

```
16140       PRINT\\"        No room for more data"
16150       t$=INKEY$(-1)
16160       RETurn
16170     END IF
16180     CLS
16190     PRINT\" PAYMENTS FOR ";q$\\
16200     FOR i=curr_m TO year
16210       PRINT month$(i_m);
16220       INPUT payments(real,items(real),i_m)
16230     NEXT i
16240     payment$(real,items(real))=q$ &
          "                "
16250     items(real)=items(real)+1
16260     END REPeat nh_loop
16270 END DEFine new_headings
```

Commentary

Line 16060: The meaning of this prompt will not be obvious until you have some experience of the program. We have already noted that one of the functions of the program is to provide an average figure per month to cover all the bills which must be met during the course of a year. It is possible, however, to exempt a bill from this budgeting process. Take, for example, the situation where you intend to go on holiday at the end of the current 12-month period, at a cost of £500, knowing that in that month you will receive a holiday bonus, also of £500. The bonus is duly entered under additional income and the holiday as a payment. The result is that the average budget figure for each of the next 12 months is increased by £42, so that, by the end of the 12 months, the holiday will have been paid for. This is clearly not what you want, since there is no need to budget in advance for the holiday. In this case, the payment for the holiday is entered, but the payment name is preceded by a '*', as a signal to later parts of the program that the associated bill is not to be included when calculating the average monthly budget.

Line 16100: The next module you will enter, which performs the calculations on the bare figures you provide.

Line 16220: The array PAYMENTS will hold the amounts entered for each bill. There are three dimensions to the array:

1) The side of the array (real or hypothetical).
2) The number of the payment (0 – 59). The number of payments registered on each side of the array is held in the two element array ITEMS.
3) The month (0 – 11).

Line 16240: The name of the payment is held in PAYMENT$, which is padded with spaces to ensure standardised length of payment names when they are printed in later tables.

Testing

If the program has been initialised, you should be able to type:

new_headings [ENTER]

and be prompted to input a heading. Respond with:

TEST1

followed by a series of monthly payment figures, corresponding to the number of the month.

When you are prompted for the next payment name, enter 'zzz' and the procedure will terminate. Now type:

print payment$(0,0) [ENTER]
for i = 0 to 11:print payments (0,0,i):next i
print items(0)

You should see the payment name, the numbers from 1 to 12 and finally the number 1, indicating that ITEMS(0) records a single item on the real side of the arrays.

Module 4.3.6: Data files — store

A standard module, this is given at this early stage because, of all the programs in this book, Budget is probably the most tiresome to have to re-enter data manually for. Note that not all of the arrays dimensioned when the program is initialised are saved. The reason for this is that most of them are derived from the income and payments figures you have already entered. Rather than saving them, it is more economical simply to save the income and payments and recalculate the rest when the data is reloaded.

Module 4.3.6: Lines 26000 – 26260

```
26000 REMark ************************
26010 DEFine PROCedure store
26020 REMark ************************
26030   CLS
26040   AT 1,14 : PRINT "SAVE DATA"
26050   INPUT\\" Name of data file:";file$
26060   tfile$="mdv1_" & file$
26070   DELETE tfile$
26080   OPEN_NEW #8,"mdv1_" & file$
26090   PRINT #8,curr_m : PRINT #8,year
26100   FOR i=0 TO 1
26110     PRINT #8,items(i)
26120     IF items(i)<>0
26130       FOR j=0 TO items(i)-1
26140         PRINT #8,payment$(i,j)
26150         FOR k=0 TO 11
```

```
26160            PRINT #8,payments(i,j,k)
26170          NEXT k
26180        NEXT j
26190        FOR j=0 TO 11
26200          PRINT #8,income_1(i,j)
26210          PRINT #8,income_2(i,j)
26220        NEXT j
26230      END IF
26240    NEXT i
26250    CLOSE#8
26260 END DEFine store
```

Module 4.3.7: Data files — recall

A standard module. Note the call to the UPDATE module, which you have not entered yet, but which will fill out the arrays with figures derived from the income and payments stored on the microdrive.

Module 4.3.7: Lines 27000 – 27260

```
27000 REMark ************************
27010 DEFine PROCedure recall
27020 REMark ************************
27030   CLS
27040   AT 1,14 : PRINT "RECALL DATA"
27050   DIR mdv1_
27060   INPUT\\" Name of data file:";file$
27070   OPEN_IN #8,"mdv1_" & file$
27080   INPUT #8,curr_m,year
27090   FOR i=0 TO 1
27100     INPUT #8,items(i)
27110     IF items(i)<>0
27120       FOR j=0 TO items(i)-1
27130         INPUT #8,payment$(i,j)
27140         FOR k=0 TO 11
27150           INPUT #8,payments(i,j,k)
27160         NEXT k
27170       NEXT j
27180       FOR j=0 TO 11
27190         INPUT #8,income_1(i,j)
27200         INPUT #8,income_2(i,j)
27210       NEXT j
27220     END IF
27230   NEXT i
27240   CLOSE#8
27250   update
27260 END DEFine recall
```

Module 4.3.8: The calculations

All the important calculations for the program are carried out by this procedure. There is nothing complex about what is done, it is simply cumbersome, since the payments must be scanned in totally separate ways — once along the months from 0 to 11 and once in order of payments, from 0 to a possible 59. Be warned that, when the program holds the maximum number of payments, this procedure takes some considerable time to complete.

Module 4.3.8: Lines 14000 – 14290

```
14000 REMark ************************
14010 DEFine PROCedure update
14020 REMark ************************
14030   FLASH 1 : PRINT "CALCULATING" : FLASH 0
14040   IF items(real)=0 THEN RETurn
14050   y_budget(real)=0 : cum_budget=0
14060   cum=0
14070   FOR i=0 TO items(real)-1
14080     y_total=0
14090     IF payment$(real,i,1)<>"*"
14100       FOR j=0 TO 11
14110         y_total=y_total+payments(real,i,j)
14120       NEXT j
14130       monthly(real,i)=y_total/12
14140       y_budget(real)=y_budget(real)+
            y_total/12
14150     END IF
14160   NEXT i
14170   FOR i=curr_m TO year
14180     paytotals(real,i_m)=0
14190     FOR j=0 TO items(real)-1
14200       paytotals(real,i_m)=paytotals(real,
            i_m)+payments(real,j,i_m)
14210       IF payment$(real,j,1)<>"*"
14220         cum_budget=cum_budget+payments
              (real,j,i_m)
14230       END IF
14240     NEXT j
14250     b_balance(real,i_m)=y_budget(real)*
          (i-curr_m+1)-cum_budget
14260     cum=cum+income_1(real,i_m)+income_2
          (real,i_m)-paytotals(real,i_m)
14270     balance(real,i_m)=cum
14280   NEXT i
14290 END DEFine update
```

Commentary

Line 14030: As a reassurance that the program is not locked up, the word CALCULATING is flashed on the screen while this procedure is being carried out.

Lines 14070 – 14160: For each payment (the I loop), the payments made in each month (the J loop) are added together in the temporary variable

Y_TOTAL. One-twelfth of this total payment over the year is saved in the corresponding line of the array MONTHLY, which records what the monthly budget figure for each payment is. In addition, the same figure is added to Y_BUDGET, which will, by the end of the I loop, hold the total of the average monthly payments for all of the bills — ie the overall average monthly payment. Note that line 14090 excludes any bills whose names begin with '*' from this process.

Lines 14170 – 14280: These lines deal with figures which will apply to each month separately. For each month of the year, the J loop adds together all the payments to be made in that month and stores them in the array PAY-TOTALS. Within the same loop, CUM_BUDGET keeps track of the total of payments made on items which are included in the average budget. Each time the J loop ends, CUM_BUDGET will hold the total payments on budgeted items since the year (as represented by the I loop) began.

Line 14250: The array B_BALANCE is used to record how much, in any particular month, the actual payments on budgeted items are ahead or behind the amount put aside in the average monthly budget. For instance, if a large budgeted payment is to be made at the end of the year, one-twelfth of it will be put aside in each of the preceding months and a large surplus will build up in B_BALANCE. The contents of B_BALANCE for each month are calculated by taking the average monthly budget figure (Y_BUDGET), multiplying it by the number of months since the year started and then subtracting all the payments to date on budgeted items (CUM_BUDGET).

Lines 14260 – 14270: The cumulative balance, ie how much has been saved out of income since the beginning of the year, is stored in BALANCE. It is calculated by adding the income figures each month to the temporary variable CUM and subtracting the total payments.

Testing

It is not really practical to test this module until we have entered the next series, which allows the figures to be dumped out in the form of a table.

Module 4.3.9: Formatting a number

We have already looked, in the preceding programs in this chapter, at the problems of formatting a number. The current module is simpler than those which have gone before, because Budget works only with integer numbers up to 9999.

Module 4.3.9: Lines 24000 – 24130

```
24000 REMark ************************
24010 DEFine PROCedure form_print (nn)
24020 REMark ************************
```



```
24030   LOCal i,n
24040   n=INT(ABS(nn)+5E-2)
24050   n$=""
24060   FOR i=3 TO 0 STEP -1
24070     n$=n$ & INT(n/10^i)
24080     n=n-10^i*INT(n/10^i)
24090   NEXT i
24100   IF nn<0 THEN PAPER 2
24110   PRINT n$;
24120   PAPER real+4
24130 END DEFine four
```

Testing

Type:

real = 0[ENTER]
form_print 1[ENTER]

The result should be the printing of '0001' on a green background.

Module 4.3.10: Printing a heading for a table

In a moment, we shall be entering the main display module of the program, or rather two main display modules. Since both parts of the display need a heading for three-monthly columns, a separate module for this purpose is included.

Module 4.3.10: Lines 23000 – 23110

```
23000 REMark ************************
23010 DEFine PROCedure table_top
23020 REMark ************************
23030   LOCal i
23040   CLS
23050   PAPER 0 : INK 7
23060   PRINT "MONTHS          ";
23070   FOR i=start TO start+2
23080     PRINT month$(i_m, 1 TO 3);"  ";
23090   NEXT i
23100   PAPER real+4 : INK 0
23110 END DEFine table_top
```

Commentary

Lines 23070 – 23090: The I_M function is used to print out the first three letters of the names of three months, starting with the value held in START.

Testing

Provided that the program has been initialised, thus loading the month names into MONTH$, type:

table_top(0)

and you should see MONTHS, JAN, FEB and MAR spaced out across the top of the screen in inverse lettering.

Module 4.3.11: Displaying the payments

We have now entered all that we need in order to be able to print out an orderly table of the payments which the user has recorded.

Module 4.3.11: Lines 12000 – 12380

```
12000 REMark ************************
12010 DEFine PROCedure display
12020 REMark ************************
12030   CLS
12040   AT 1,13 : PRINT "BUDGET DISPLAY"
12050   REPeat check
12060     INPUT\\"Number of month to start: ";
          start
12070     IF start>=1 AND start<=12 THEN EXIT
          check
12080   END REPeat check
12090   start=start-1
12100   IF start-12*(start>=curr_m)>=curr_m-3
        THEN start=curr_m-3
12110   start=start+12*(start<0)
12120   INPUT\"Analysis only (Y/N):";q$
12130   IF q$="y" OR q$="Y" THEN analysis :
        RETurn
12140   FOR i=0 TO 45 STEP 15
12150     table_top
12160     PAPER 0 : INK 7
12170     PRINT " B    "
12180     PAPER 4+real : INK 0
12190     AT 1,0
12200     FOR j=0 TO 14
12210       IF i+j=items(real) THEN EXIT j
12220       space
12230       PRINT payment$(real,i+j);
12240       space
12250       FOR k=start TO start+2
12260         form_print payments(real,i+j,k-12
            *(k>11))
12270         space
12280       NEXT k
12290       form_print (monthly(real,i+j))
12300       space : PRINT
12310     END FOR j
12320     PAPER 0 : PRINT FILL$(" ",36);
12330     PAPER real+4
12340     t$=INKEY$(-1)
12350     IF i+j=items(real) THEN EXIT i
```



```
12360   END FOR i
12370   analysis
12380 END DEFine display
```

Commentary

Lines 12050 – 12110: The start month number for the three-monthly display is received into START. Since Budget works on a single 12-month period, trying to start a table in the last month, or the month before, would produce a nonsense, wrapping around to the beginning of the year. For this reason, if the START given is not a full three months before the current month, it is reduced — eg if the current month is July and the requested start is May, this will be reduced to April, so that April, May and June will be represented.

Lines 12120 – 12130: The next module will provide for the printing of an analysis of the figures displayed by the current table. These lines allow for the user to dispense with the display of individual bills and go straight to the analysis.

Lines 12140 – 12360: This loop allows the full 60 bills to be screened in four successive displays.

Lines 12150 – 12180: The top of the table is printed, and an extra heading on the right of the screen — B — under which will be placed the average monthly budget for each item.

Lines 12200 – 12310: These two embedded loops print the name of each payment on the left of the screen, then the figures for the three months, using SPACE to separate the items on each line.

Testing

If you have entered some data, type:

goto 11000

and call up option 1 on the menu. Specify a month and you should see an orderly presentation of any bills you have entered. The display will terminate with an error when you press a key, since the next module is called at the end of this one. Perform the test again, but this time use the menu to change to the hypothetical half of the array to ensure that the figures in that half are properly displayed. Now that you can display the results of changes easily, it would be wise to check modules like NEW_HEADINGS to make sure they work on both sides of the arrays.

Module 4.3.12: Displaying the analysis

As mentioned in the the commentary on the last module, once the figures for payments have been displayed, the program then goes on to display the

analysis which was carried out by the UPDATE module. The short functions we have already defined make the printing of the new table a very simple matter. A list of headings is printed using TABLE, then TABLE2 prints the columns of corresponding figures for the three months covered.

Module 4.3.12: Lines 13000 – 13270

```
13000 REMark ************************
13010 DEFine PROCedure analysis
13020 REMark ************************
13030   table_top
13040   AT 1,0
13050   table "MONTHLY TOTAL "
13060   table "BUDGET        "
13070   table "BUDGET BALANCE"
13080   table "MAIN INCOME   "
13090   table "SUPP. INCOME  "
13100   table "TOTAL INCOME  "
13110   table "MONTH BALANCE "
13120   table "CUM. BALANCE  "
13130   FOR i=start TO start+2
13140     AT 1,0
13150     table2 (paytotals(real,i_m))
13160     table2 (y_budget(real))
13170     table2 (b_balance(real,i_m))
13180     table2 (income_1(real,i_m))
13190     table2 (income_2(real,i_m))
13200     table2 (income_1(real,i_m)+income_2
          (real,i_m))
13210     table2 (income_1(real,i_m)+income_2
          (real,i_m)-paytotals(real,i_m))
13220     table2 (balance(real,i_m))
13230   NEXT i
13240   PAPER 0 : PRINT FILL$(" ",37);
13250   PAPER real+4
13260   t$=INKEY$(-1)
13270 END DEFine analysis
```

Testing

The test conducted on the previous module can now be continued through to the display of the analysis of the figures you have entered. Remember that this is your first real check of the working of the UPDATE module, so do ensure that figures displayed make some sense.

Module 4.3.13: Reset_data

So far, we have not really experimented much with the hypothetical capabilities of the program. When you *do* start to use those capabilities, you will sometimes find that you have built up such a body of changes that getting back to a semblance of the real situation will involve a lot of tire-

some deletion. This module relieves you of that need by simply copying the real half of the arrays over into the hypothetical half so that you can start your experiments with a clean slate.

Module 4.3.13: Lines 15000 – 15190

```
15000 REMark ************************
15010 DEFine PROCedure reset_data
15020 REMark ************************
15030   y_budget(1)=y_budget(0)
15040   FOR i=0 TO items(0)-1
15050     payment$(1,i)=payment$(0,i)
15060     monthly(1,i)=monthly(0,i)
15070     FOR j=0 TO 11
15080       payments(1,i,j)=payments(0,i,j)
15090     NEXT j
15100   NEXT i
15110   FOR i=0 TO 11

15120     paytotals(1,i)=paytotals(0,i)
15130     b_balance(1,i)=b_balance(0,i)
15140     income_1(1,i)=income_1(0,i)
15150     income_2(1,i)=income_2(0,i)
15160     balance(1,i)=balance(0,i)
15170   NEXT i
15180   items(1)=items(0)
15190 END DEFine reset_data
```

Testing

You will need to have made some entries on the hypothetical side of the arrays and checked that they have been registered. Call up menu option 5 and then display the hypothetical figures. They should now be identical with the real figures.

Module 4.3.14: Changing the month

The program will not be of much use if it is always going to be stuck on the same 12-month period, so this module makes provision for the start month to be altered as time passes. The array simply accepts payment and income figures for the new months which have come in at the end of the period. If the change were from May to July, figures for next May and next June would be requested for each payment heading and also for main and supplementary income.

Module 4.3.14: Lines 17000 – 17320

```
17000 REMark ************************
17010 DEFine PROCedure reset_month
17020 REMark ************************
```

```
17030   CLS
17040   AT 1,13 : PRINT "UPDATE MONTH"
17050   REPeat check
17060     INPUT\\"Number of new month (1-12): ";
          month2
17070     IF month2>=1 AND month2<=12 THEN EXIT
          check
17080   END REPeat check
17090   month2=month2-1
17100   IF month2=curr_m
17110     PRINT\\" Program already on that month"
17120     t$=INKEY$(-1)
17130     RETurn
17140   END IF
17150   IF month2<curr_m THEN month2=month2+12
17160   FOR i=curr_m TO month2-1
17170     CLS
17180     AT 1,13 : PRINT "UPDATE MONTH"
17190     PRINT\\"Input amounts for next ";month$
          (i_m);\\
17200     FOR j=0 TO items(real)-1
17210       PRINT payment$(0,j);" (";payments
            (0,j,i_m);"): ";
17220       INPUT payments(0,j,i_m)
17230     NEXT j
17240     INPUT\"Main Income: ";income_1(0,i_m)
17250     INPUT\"Additional income: ";income_2
          (0,i_m)
17260   NEXT i
17270   curr_m=month2-12*(month2>11)
17280   year=curr_m+11
17290   real=0
17300   update
17310   reset_data
17320 END DEFine reset_month
```

Commentary

Lines 17160 – 17260: These two loops shuttle through all the available payments, displaying the figures they were set for in what is now the previous year and inviting new figures.

Testing

Call up option 6 from the menu and specify the next month to the current month. You should be invited to give one complete set of payment income figures for the new end of Budget's year. You will also find, in using the display function, that the year start and end have been adjusted accordingly.

Module 4.3.15: Making changes

The first of a series of four modules which, together, allow changes to be made to existing income or payment headings. The purpose of the current module is merely to determine whether income or payments are to be changed.

Module 4.3.15: Lines 18000 – 18130

```
18000 REMark ************************
18010 DEFine PROCedure changes
18020 REMark ************************
18030   CLS
18040   AT 1,15 : PRINT "CHANGES"
18050   PRINT\\" Do you wish to change figures
        for:"
18060   PRINT\"  1) A payment"\"  2) Income"
18070   INPUT\" Which do you require: ";z2
18080   SELect ON z2
18090     ON z2=1 : change_payment
18100     ON z2=2 : income
18110   END SELect
18120   IF z2=1 OR z2=2 THEN update
18130 END DEFine changes
```

Testing

The module can be tested as far as income goes since it uses the income entry module entered earlier. Call up changes and specify income. You should be invited to give new main and additional figures for each of the coming 12 months.

Module 4.3.16: Checking a payment name

Several previous programs, like Nnumber and Accountant, have needed provision to check that items requested by the user are in fact present in the file. Budget includes a separate module to make this check, returning the results of the search to the next module in the form of the variable FOUND.

Module 4.3.16: Lines 22000 – 22150

```
22000 REMark ************************
22010 DEFine PROCedure find (q$)
22020 REMark ************************
22030   found=0
22040   q$=q$ & FILL$(" ",14-LEN(q$))
22050   FOR place=0 TO items(real)-1
22060     IF q$=payment$(real,place)
22070       found=1
22080       EXIT place
22090     END IF
```

```
22100   NEXT place
22110     PRINT\" Item not found, please check
          with"\" monthly display."
22120     t$=INKEY$(-1)
22130     RETurn
22140   END FOR place
22150 END DEFine find
```

Module 4.3.17: Changing the figures for a payment

This module is similar to the one used to change income. Monthly figures for the specified payment heading are displayed and the user is invited either to confirm or change them.

Module 4.3.17: Lines 19000 – 19150

```
19000 REMark ************************
19010 DEFine PROCedure change_payment
19020 REMark ************************
19030   INPUT\\"Payment to be changed: ";q$
19040   find (q$)
19050   IF NOT found THEN RETurn
19060   CLS
19070   PRINT\" ";payment$(real,place)
19080   PRINT\" Input new amount or '\' to
        leave."\\
19090   FOR i=curr_m TO year
19100     INPUT (month$(i_m) & "(" & payments
          (real,place,i_m) & "): ");q$
19110     IF q$<>"\"
19120       payments(real,place,i_m)="0" & q$
19130     END IF
19140   NEXT i
19150 END DEFine change_payment
```

Testing

Just as you were able to change income figures, you should now be able to call up menu option 2, specify a payment heading and re-enter the figures for that payment.

Module 4.3.18: Deleting items

The final module of the main program allows an entire payment heading, its name and the 12-monthly figures associated with it, to be removed from the file.

Module 4.3.18: Lines 21000 – 21160

```
21000 REMark ************************
21010 DEFine PROCedure remove
21020 REMark ************************
21030   CLS
21040   AT 1,14 : PRINT "DELETIONS"
```



```
21050   INPUT\\" Name of payment to be deleted: "
        ;q$
21060   find (q$)
21070   IF NOT found THEN RETurn
21080   items(real)=items(real)-1
21090   FOR j=place TO items(real)-1
21100     payment$(real,j)=payment$(real,j+1)
21110     FOR k=0 TO 11
21120       payments(real,j,k)=payments(real,
          j+1,k)
21130     NEXT k
21140   NEXT j
21150   update
21160 END DEFine remove
```

Testing

You should now be able to delete payment headings by calling up menu option 4. If this facility works correctly, the program is ready for use.

APPENDIX

Instructions for Use of Checksum Generator Tables

The following short program is designed to act as a check that the program you have entered is the same in every important respect as those from which the programs in this book were listed out. It does this by reading the program file from microdrive and successively adding and subtracting the values of the characters which make up each line of the program, excluding spaces. The values produced are known as 'checksums' and, if a program line is copied correctly from one program to another and examined by the same checksum generator, it should produce the same checksum — if there is an error it should be indicated by a difference in the checksums. It is possible to make changes in a line which will result in the same checksum being calculated for it, but it is unlikely that this will happen often.

If you wish to check your program against the tables, you will have to load the Checksum Generator into memory and then place a cartridge bearing the program to be checked in drive 1. Run the Checksum Generator and supply the program name. You will also be asked to supply the start and finish lines for the table. This will allow you to select either a specific module(s) or to enter 1 and 99999 (an impossibly high line number) to generate a table for the whole program.

The tables are laid out in modules, with the start line of each module marked. Line numbers are not included for the tables since this would make them impossibly large. To read the table, find the module you want, then read each line of the table from left to right. Since the tables in the book and the tables you will generate will be of the same format, you should have no difficulty in comparing them.

If the tables are exactly alike, it is highly probable that the program is entered correctly in every respect. If you come across lines where the checksums differ between your table and that contained in the book, go to the offending line and compare it very carefully with what is contained in the listing in the book — you will probably find that there is an error. Note that differences in spacing will not result in differences in checksums — spaces are ignored. If you cannot find *any* difference, there is always a small possibility that last-minute changes during the production of the book have somehow not been recorded in the checksum table, so that the

checksum in the book refers to a line which has been subsequently changed. We make every effort to ensure that this does not happen but it is wise to bear in mind this possibility if you cannot find a difference and the program runs correctly.

Checksum Generator

```
1000 REMark ************************
1010 REMark control
1020 REMark ************************
1030   INK 7 : PAPER 3 : CLS
1040   count=0
1050   INPUT "Name of program: ";program$
1060   INPUT\"Start Line: ";start
1070   INPUT\"Finish line: ";finish
1080   program$="mdv1_" & program$
1090   INPUT\"Printer output (Y/N): ";hard$
1100   hard=hard$="Y" OR hard$="y"
1110   IF hard
1120     OPEN #7,ser1
1130   ELSE
1140     OPEN #7,scr
1150     INK#7,7 : PAPER#7,3
1160     CLS#7
1170   END IF
1180   PRINT#7,"CHECKSUM TABLE FOR ";program$
1190   PRINT#7,\"Start line is ";start
1200   PRINT#7,\"Finish line is ";finish\\
1210   OPEN #8,program$
1220   :
1230   :
1240   REPeat lines
1250     INPUT #8,line$
1260     line_number=line$
1270     IF line_number>finish THEN EXIT lines
1280     IF line_number>=start
1290       analyse
1300       check_print
1310     END IF
1320     IF EOF(#8) THEN EXIT lines
1330   END REPeat lines
1340   :
1350   :
1360   PRINT#7 : CLOSE #7
1370   CLOSE #8
2000 REMark ************************
2010 DEFine PROCedure check_print
2020 REMark ************************
2030   IF line_number/1000=INT(line_number/1000)
2040     PRINT#7,\\line_number;
         " ************************"          \\
2050     count=0
2060   END IF
```



```
2070   number$="       " & checksum
2080   number$=number$(LEN(number$)-4 TO)
2090   PRINT#7,!number$;
2100   count=count+1
2110   IF hard AND count=6
2120     count=0
2130     PRINT#7
2140   END IF
2150 END DEFine check_print
3000 REMark ************************
3010 DEFine PROCedure analyse
3020 REMark ************************
3030   checksum=0
3040   FOR i=1 TO LEN(line$)
3050     IF line$(i)<>" "
3060       IF i/2=INT(i/2)
3070         checksum=checksum+CODE(line$(i))
3080       ELSE
3090         checksum=checksum-CODE(line$(i))
3100       END IF
3110     END IF
3120   NEXT i
3130 END DEFine analyse
```

CHECKSUM TABLES

Anaclock

```
1000 ************************
  -1   -91    -3   335   214     3
  47   202     9  -130   116   134
 246   241   181   179   -67   -66
 -18    49    59

2000 ************************
  -2  -154    -4    44   -65   214
  -1    14    48    47

3000 ************************
  -3  -200    -5    16   -38  -314
 129    26   -50  -302  -105    17
  37    52   110   118   -30    49
  48

4000 ************************
  -4  -165    -6    10   -45   128
 210   230   203   223     4   -96
   9    52    51

5000 ************************
  -5  -264    -7  -128   123     8
  22   116  -125   -75    58   -57
  77   110   -91    50    49

6000 ************************
  -6  -398    -8    -1   -44   -44
 234   212   231   209   238   216
  51   -60    -2   -61  -253   139
 151  -243   -87
```

Clock

```
1000 ************************
  -1   -91    -3     5    49    -9
-127   108   126    65   -61    -3
  56    55

2000 ************************
  -2  -154    -4   181   584  -116
  -1    14    48    47

3000 ************************
  -3  -200    -5    68    15   -21
-515   128    25   -51  -292  -106
  16    36    51   109   117    65
 -32    47    57

4000 ************************
  -4  -144    -6   158   -94   -69
-182   161  -101    61   -95    31
  53    52

5000 ************************
  -5  -264    -7    38   -81   122
   7   -55    27   -78  -184   -18
 -90    51    50

6000 ************************
  -6  -143    -8  -112   -60   107
  64    44   -65   -67   115   -37
 -58   113   -61    41   -62   107
 -65  -134   -55   119   -58   -45
 -59   -24
```

Timer

```
1000 ************************
 -43   -91   -45     5    49    -5
  67   -10    49    48

2000 ************************
 -44  -154   -46   611    20   121
  89    14    48    47

3000 ************************
 -45  -147   -47   -32    67    64
-344   199   212    72    86   -61
 152    92    62    89   101  -104
  87     1   -62    68   -81  -155
 -66    56    -6   -38   167   207
 134    34    29    30    27    53
  52
```

```
4000 ************************
  -46  -256   -48   -48   168  -475
  -85   -55    50    27   -26   128
   82    44  -105    10   -86    48
   47

5000 ************************
  -47  -217   -49    66    66    31
   86   153    83   -15   140   -11
 -207  -145   -27   135  -128   -31
  132  -131   -17    83  -145    63
  -86   168    -3    94   -67    47
   46

6000 ************************
  -48  -190   -58    30   124     0
   46    45

7000 ************************
  -49  -230   -51  -150   168   204
  -94   -37   -31     9    32   -67
   54    58    26   -58   -26    12
  -89     5    29   -54    51    50

8000 ************************
  -50  -319   -52    63   -34   -67
  -72   -91  -731   -81    -6    57
  -88    78   -37    26    -6    81
   52    46   -18    14   -66   -19
  113  -631   -11  -299    92    61
   57  -142    51    50

9000 ************************
  -51  -236   -53   -24    61    83
  -43    28    92    78   -65    38
  -62    47    46

10000 ***********************
  -49   148   -47   -36  -119   -66
  -13   158   -57   -69    73   149
 -207    17   105   158  -307    49
  -19  -281  -189   -86   -10  -261
   28   -66   -85    96   -92    10
 -106    98  -184  -123   -26  -105
 -104

11000 ***********************
   -6   225    -4  -119   -24  -102
   34  -217  -225   -17  -267    50
 -144  -114  -141  -153    55
```

Event

```
1000 ************************
   -1   -91    -3     5    49    68
 -127     3  -228     2   -61   204
   -4    55    54

2000 ************************
   -2  -154    -4   611    19   102
   15    49    48

3000 ************************
   -3  -200   -47   -16    67  -223
 -678   -93    63    10   -32  -195
   12    24  -129  -205   231   -73
 -161  -214   262   -94   -26   236
  133    30   -46   205   -84    10
   40  -141    47   -55   130  -154
  128   -29    50    49

4000 ************************
   -4  -158    -6   121  -144    78
  -85   105  -111   148  -122    57
   57  -344   181    -1  -148    52
   51   -89    19    55    54

5000 ************************
  -47  -264   -49    68    16    79
  -11    68   -52   -98    54    53

6000 ************************
  -48  -161   -50    79  -117   -78
   54    65    48     5    53    52
```

Designer

```
1000 ************************
  -43  -158   -45   199   -74    55
  243  -358     6    52  -147    95
  113   -91    39    53    52

2000 ************************
  -44  -154   -46   120   153   -55
  119   -15   121    12    57    56

3000 ************************
  -45  -242   -47   -22     2   157
  -56   126   154   -59   134    84
  -82    63   -74   -71    50    49

4000 ************************
  -46  -262   -48  -127  -108  -139
 -120  -149  -119  -146  -105  -123
  148   149    55  -111   -40   -43
   51  -482  -458  -417  -343  -449
 -432  -185    53   -92    48    47

5000 ************************
  -47  -291   -49   -24     0    80
 -361   -80     5     1    58     4
  -74  -118    50    49

6000 ************************
  -48  -296   -50   -25    93    93
  117   -25   -94    85    85  -356
  -75     9     6    52    89   -80
 -128    44    54

7000 ************************
  -49  -257   -51   -26    92    92
   95   -39    51    50   -31   -32
 -126  -127   -35   -36   -37    77
 -364   -83    11     9    55   -33
  -77    51   -86    46    45

8000 ************************
   -8  -166   -10   -50   113    71
  -30    47   -31     3    77    76
  -19   -22    51   -26     7   -10
  -13    12   105     5    54   -22
   10    45    42    42    39    39
   47    47    44    54    58     7
   47    46

9000 ************************
   -9  -129   -11    -4    61   179
  -89  -108    45    -1    58    49

10000 ***********************
  -49   198   -47    88   -52  -106
   90  -106  -103    92   -19  -111
   48  -105  -104

11000 ***********************
  -48   123   -46  -119  -136   -38
  -54   -18   -31   -23  -153  -152
 -143   -84  -296  -160   -98    52
   52  -105   -38  -111    36    21
 -297  -148    49   -27  -104    61
  -39  -112   -99   -74    21  -150
 -200  -106   -50   -78   -56  -109
 -108

12000 ***********************
  -47   199   -45  -240   129   132
 -107    37    30    36    13   -24
   25  -103  -102

13000 ***********************
  -46   194   -44  -117  -211  -126
 -204    16    23   -72  -240  -161
   38    49    49  -119   -18    25
  -97   -96
```



```
14000 ***********************
  -45   113   -43  -116  -230  -290
 -124  -178   -69  -229  -161    41
   49    49  -119   -18   -57   -97
  -96
```

3-D Graph

```
1000 ************************
  -43    91   -45   429    78  -164
  187    39   -63  -102    88  -241
   -4   -22    -8   -11    84   196
  100    66    79    12   -82  -128
 -282   144   -97     3    -9    50
   60

2000 ************************
  -44  -139   -46  -196    85   -13
  -59   -22   -16   -66   -14     9
    4   -59   -56    32    51    50

3000 ************************
  -45  -169   -47   -81    29    20
   49    48

4000 ************************
  -46  -141   -48   118   174    -6
    3    -8  -104   275   136   137
   98   -98    89   111   -43    34
 -103    -9    36    55    54

5000 ************************
  -47   -28   -49    50   -46    11
   47  -106   -43    44  -268   -60
  -62   -64   -66    49    85    21
   46  -187   -50    54    27    15
   51    73   -23    48    -5   -42
   56     2   -37    53    -2   -68
   50    -6   -71
```

Screen

```
1000 ************************
  -43  -193   -45    28   124  -124
  -35    55    13   123   -26    59
   60

2000 ************************
  -44  -141   -46   114    43  -103
   28    49    48

3000 ************************
  -45  -239   -47   287   -11   -93
  -78    48    47

4000 ************************
  -46  -156   -48   159   133    84
  -46     9  -110    10    55    54

10000 ***********************
  -49   142   -47   -60  -106   -65
  -27  -100   -99

11000 ***********************
  -48   149   -46  -213  -185   -91
    1   -61    58   -17  -107  -106

12000 ***********************
  -47   116   -45  -660  -212  -222
   51   -51   -48    93   -80   366
   26  -132   -90   -49   238   280
  -75   -51
```

Characters

```
10000 ***********************
  -49    80   -47  -265    23    13
  -54   -36   -39   -83  -108  -107
```

```
11000 ***********************
  -48   104   -46  -223  -256   -32
  -65   -99   -98

12000 ***********************
  -47   -98   -45     3   -13  -334
 -200  -155   300    54  -158  -162
    6  -183  -269    50    50   -16
 -376  -103  -192  -143   -48   -37
   51   -39   -55   -80   -99   -98

13000 ***********************
  -46    81   -44    13  -208  -134
  -30   -15   -45   -98    21  -338
 -164  -150  -144  -143  -200  -176
 -228  -177  -181   -87   -60   -84
   -5  -164  -119  -136   206  -127
  -48   -67   -81   -80   -83    68
 -140   439  -103    77  -148   436
 -110  -145  -183   -44   -38   -69
  -26   -86   -49  -151   -40     2
   13  -186    -3   -75  -203   -62
 -100  -109  -108

14000 ***********************
  -45    97   -43   -19  -205   185
 -174   187  -172   -92  -154    14
  -80  -146   -75   -99   -98

15000 ***********************
  -44    76   -42  -156  -156   147
 -198   332  -181    58    47  -204
  -98  -100   -99

16000 ***********************
  -43   105   -41  -155  -155  -380
   56    56   -62   -92  -102

17000 ***********************
  -42   111   -40  -154  -154   180
 -255   -42  -254   -95    50    50
  -18   -62   -97   -96

18000 ***********************
  -41   103   -39  -137  -152  -152
  181    62   -95    61    50   -46
  -17   -70   -96   -95

19000 ***********************
  -40   140   -38  -136  -151  -151
  182    63   -94    62    51   -45
  -16   -33   -95   -94

20000 ***********************
  -50   237   -48  -146  -161  -161
  -24     6  -104    52    41   -55
  -26    64  -105  -104

21000 ***********************
  -49    93   -47  -161   -50  -160
  194    51   -97    52   -83  -107
 -106

22000 ***********************
  -48   192   -46  -119  -213  -128
 -206    14  -144    26  -107  -106

23000 ***********************
  -47   111   -45  -118  -232  -292
 -126  -126   -56    20    10

24000 ***********************
  -46    86   -44   -66  -162   -17
  -47  -219   -24    55   -90
```

Sound Demo

```
1000 ************************
   -1  -141    -3   -60    45   -64
  -65    47   281   339  -159   155
   72    86   -14  -116  -117    97
  109    14    19   -25   206   -62
   82    48   -25    73  -305    42
```

```
  129    83  -101   -17    -6  -105
  -63   -65   -55   -56    56  -101
   27  -104    23  -107    89  -110
  100  -102   126  -105   198  -108
   40  -111   -18    26     6    39
   61    60

2000 ************************
  -44  -199   -46    66   330   -17
 -109   270   450    40   -23    56
   55

1000 ************************
  -43  -133   -45     5   106   -74
   36    50    49

2000 ************************
   -2  -154    -4   195    77   114
  163    -5  -103  -103    22    56
   55

3000 ************************
   -3  -257    -5    78    91    28
   51    71   -75   -50   106    55
   54  -146   166   -76    76    -2
  111    51   130   -84   138    58
 -154  -111    75    54   -35   -16
   62   -27    -1    59  -356     1
   56   -33    -3    53   -25     4
   61   -28     0    50   -31    -4
   55   -34   -24    63  -344   -13
   80    59    54    53   -97   100
  184   -70   168    81    61    56
   55  -220   -87   -11   -75    61
   60

4000 ************************
   -4  -144    -6    98   -44   -75
   52   191  -130   -27    51   -60
  -97    46    51    58

5000 ************************
   -5   -22    -7   -34   -33   -26
   47    46

6000 ************************
   -6    84    -8    36    23    32
   20    47    47    17    48
```

Unifile

```
10000 ************************
  -49   -39   -47  -384  -264  -151
 -204   308   -56   -47  -176  -183
  -54  -107  -219  -229   208   -60
 -167    51  -260   -66  -160  -162
   -6  -100  -107  -102  -101

11000 ************************
  -48   -46   -46   112  -528  -262
 -290   -73   133  -175  -201  -138
 -104  -256  -109  -128  -120  -104
 -102   206   179   187    30  -123
 -116   -45   -77    53  -102  -101

12000 ************************
  -47   207   -45   -21    80   -84
 -115   -62  -104  -222  -135  -242
   72   -48   -44   336  -198  -102
  -89  -377  -372  -386  -476    46
 -107   177    21  -122   -70   -76
   29  -107  -106

13000 ************************
  -46   177   -44  -211  -158  -194
   19  -118  -259   -99    10  -123
  -26  -106    49   -47    29   -98
  -97

14000 ************************
  -45   127   -43    26  -153   106
 -159   -45  -171   -98   -53  -252
 -106  -211  -147   -66   -98   -97

15000 ************************
  -44   171   -42  -119  -114   113
 -163  -121   -20   -97    33  -100
 -595    36   -26  -179   -84  -135
 -169  -163   -36  -107    93     7
 -127  -388  -118    10   -60    98
 -109    46   -29  -281  -105    -5
 -132   109  -113    -1  -134  -231
  -94  -383    57  -381   -85    51
 -365  -399  -259  -161  -134   -32
  288   262  -110    -2  -103   262
 -129  -174  -110    -5   -68    33
   16  -101  -100

16000 ************************
  -43   179   -41   -27  -125  -222
 -111     1   -84  -372    57  -381
  -26   -31  -268   197   165  -146
 -108   256   288    28   -73  -207
  -48    65  -101   -81  -187    56
  -74  -180   -18  -170     5   -99
  -98

17000 ************************
  -42   129   -40    69  -299   260
  -40   -93   -92

18000 ************************
  -41  -214   -39   -98  -178   111
   79   -92   -91

19000 ************************
  -40   -69   -38   -68  -194    61
  -36   -91   -90

20000 ************************
  -50   190   -40  -121  -215  -130
 -208    12    19    19    60  -122
  -42    44  -109    13    65  -117
  -37    50  -114   -12   -23  -159
 -232  -117  -195    47  -405  -154
   39  -129    14  -108  -107

21000 ************************
  -49   189   -47  -120  -372  -233
 -293  -127  -181   -44    -6  -189
 -147  -194    46  -107   -48     1
 -181   -98  -198    42  -111    -6
  -17  -153    52  -228  -203    50
 -413   -91    42  -126   -65  -105
 -104
```

Nnumber

```
10000 ************************
  -49   -39   -47  -126  -308  -183
   -3   245   -61   -47  -198  -112
 -107

11000 ************************
  -48   -46   -46   -22  -118  -139
  -96  -323   -52  -415  -229  -161
  -50   -25   -97   -98   -10  -116
  -95   148  -134   -43  -110   -17
 -228  -204  -107  -119  -205  -229
 -202   -47   -23   -79   -83  -128
   19  -214   -38   -22   -50  -109
 -108

12000 ************************
  -47   109   -45  -182  -236   -61
  -60   -98   -97
```



```
13000 ************************
  -46   207   -44   -16  -116    35
 -145   -25   -27  -194  -131   -16
 -107   -49  -220   -80   -11  -102
 -141   148  -256  -131   -16  -107
 -106   -37  -112    83   246   -62
 -398     1  -114  -108   -77    34
 -101  -100

14000 ************************
  -45   232    -1   -25  -115  -163
 -207   -89  -276  -193   -80  -129
 -325    49  -177   102  -124  -101
   64   -95  -105

15000 ************************
  -44   210   -42   100  -114    39
   96    10    -8  -119  -129   -28
 -105    81  -120   -95   -96  -110
 -179  -110  -202   112    20  -145
  -54  -108    42    40   -96   -95

16000 ************************
  -43   202   -41   -41   -52   -22
  -99     8  -143  -107    -2  -122
  -41  -103     4  -116   -19   -99
   56    71    25  -102  -101

17000 ************************
  -42   113   -40    99  -141   -86
  -83   -92    58   -95   -89   -90
 -102   -60   -97   -96

18000 ************************
  -41   237   -39   -51    81  -110
  -15  -119  -205    63  -271   196
  -51    -1  -100   -74  -116  -199
 -233   103    49   -80   -67   -41
   82  -100   -93   -98   -76    87
 -106    56  -144  -103    56  -216
  -37   -18   -98    24    58  -102
 -101

19000 ************************
  -40   129   -38  -134   -83   -80
  -89    59   -97   -37   -99   -98

20000 ************************
  -50   118   -40  -145   -19   -62
 -118    37  -146  -169   -55  -215
 -149    -3  -171  -151  -222   -64
  245   -12  -114   -37    -4  -178
 -110   197  -277   -28   -25   -98
   39  -117   -68  -112    48   -47
  -79   -53  -103  -102

21000 ************************
  -49   191   -47  -120  -214  -129
 -207    13    20  -151  -144   -92
 -233    -2  -168  -146    49    49
  -25  -209   -38  -172  -102    45
   45  -108  -122    21  -101  -100

22000 ************************
  -48   110   -46  -119  -233  -293
 -127  -101  -148  -130   -89  -233
    1  -168  -143    49    49   -25
 -209   -24  -172  -113    45    45
 -108  -122   -61  -101  -100
```

MultiQ

```
10000 ************************
  -49   -39   -47   -29  -160  -392
 -262    89   245   -55   -57    82
 -122  -196    40  -210  -160   106
   30   -22  -175   -64   -65  -169
 -109  -104  -103

11000 ************************
  -48   -46   -46   112  -379  -262
 -182   -73   133    80  -186  -170
 -234  -201  -138  -101  -126  -106
  -63  -117  -112  -110   198   210
   83   202   209   194    41  -115
  -58   -48   -80    50  -105  -104

12000 ************************
  -47   217   -45   -21  -117  -137
  -86   -70  -111    54    25   -91
  -98   -29  -106  -212   196    97
 -171  -256   -85    41  -105  -104

13000 ************************
  -46   208   -44   -20  -116   -64
  -46  -109   -93   -21  -109  -105
 -190  -111  -203  -186     5   -70
 -111    54   -68  -112    97   -23
  -68   243  -109    -7   -35   -46
 -111   -84    32  -104  -103

14000 ************************
  -45    93   -43   -43   -54   -24
 -101     6  -145  -189    28  -124
  -43  -105    34  -118   -21  -101
   54   101    23  -104  -103

15000 ************************
  -44   106   -42    97  -143   -88
  -85    55   -98  -113  -125  -111
  -66   -67   -99   -98

16000 ************************
  -43   123   -41   -51  -152    33
  -48    56   -44   -92  -102

17000 ************************
  -42   210   -40  -113  -144   -18
 -295   231  -155    31     0    -8
  -18   -63  -113  -416   -41  -154
  -94    58    87  -233    45    31
 -103  -113    93   -61  -156   -84
  -97   297  -236   -49   147   -14
 -176  -307  -173   -98  -255   -37
  116    50   -78    36   -98   -97

18000 ************************
  -41   125   -39  -270    -6    -3
  -25   -76   -50   -67  -127  -130
 -102  -101  -154    92  -154   -65
    2   -95    -6   224    47  -102
  253    55   -99    65   -58    72
 -106    49   -51   -99   -98

19000 ************************
  -40   217   -38  -111  -137    81
 -230   -85   -16   -93  -164   -89
 -317  -360   249  -155  -146   -96
   49   -90  -100

20000 ************************
  -50    85   -40   -60    72  -119
  -24  -128  -214  -398  -309  -322
  -60   -10   -84  -126  -209  -243
   61   -14   -90   -77   -51    72
 -110   -70  -100   -83    77  -105
   11  -197   -50   -31  -111    11
   56  -104  -103

21000 ************************
  -49   120   -47  -111  -142   -91
  -88    50  -106   -63   -56  -107
 -106

22000 ************************
  -48   192   -46  -119  -213  -128
 -206    14    21   -82  -170  -183
 -161  -211    48  -230   -96    51
   51  -153   -66    43  -125    18
 -104  -105
```

```
23000 **********
  -47   111   -45  -118  -232  -292
 -126  -180   -79  -159  -180  -161
 -208    48  -230  -141    51    51
 -153   -52    42  -125   -64
```

Banker

```
1000 **********
   -1   -11    -3   337   214   111
   27    89    69    84   170   -49
  -94    -2  -302   -16    52    51


2000 **********
   -2    -4    -4   -28   335    67
  -28    77    90   197    11   145
   50    83   124    25    58    44
   68    52    61  -281  -170  -219
 -239  -118    25    29    66   -77
   87    -6   229    -4    55    54


3000 **********
   -3  -291    -5   -44    67   -29
  -24   -12    56    54   -15    54
   70   -26   -21    58   -17    58
   -3   -83    61   -51    24   -24
  146   164   -22   -62   -35   108
    7   118    60   -18   100   -97
  -21   -35   164   -30    16    62
  -93  -114   -60  -252  -101    46
   14    57   -21     1    17    69
 -147    -5   -30    57    63    20
 -188  -183   166  -125    35    65
 -116    54    53


4000 **********
   -4  -139    -6    92  -165    65
  146    75     6   147    54   168
   54     9    55  -100    37   148
  -27    42    34   153   273   175
    6  -276  -153   -23   -56    96
   61  -125    -4  -102    35    52
   51


5000 **********
   -5  -176    -7   210    72    29
 -102  -185  -105    51     0    53
   52


6000 **********
   -6  -228    -8   461   111   -37
   31    20    -7   -16    29    56
 -139    81    56    31   -83    62
  -56    80   -13   137   150    15
   50    53  -102   -72    50   -53
 -122    55   -66  -169    46  -216
 -118    68   -73   156   -32    29
   36   163   -67    51  -127    86
  -71   -91    61   -94    35   -12
 -116    46   -54    50    49


7000 **********
   -7  -127    -9   -41   -45   -38
   -6   -29    47    46   -36   -41
   86    -2   129    51    50    31
   44    43


8000 **********
   -8    76   -10   235   129   122
  122   217    -3  -109   112   101
   48   227   -58    66    61     5
   43    42


9000 **********
   -9  -249   -11    62   156    71
  149   -71   -78    51   148   119
   97    79   -36   -74  -106    62
  -81    41    51
```

```
10000 **********
  -49   109   -47  -120  -234  -294
 -128  -182  -107  -196  -175  -153
 -138   -20    15    47  -121    22
 -100   -99
```

Accountant

```
1000 **********
   -1   -11    -3   332   214    88
 -138  -306    49    48


2000 **********
   -2    -4    -4   -33   330    67
  -30   128    61   125   -24  -162
   77    12   148    85   -32  -179
   36    67    52    61    59    12
  -98   -95  -236   -84    70   -71
   65   166   -10    33    32    58
   57


3000 **********
   -3  -152    -5  -137   -54   -51
  101   -82    54   -10    21    23
   54    53


4000 **********
   -4  -245    -6   -39    66    13
   29  -128   -59   -99   -62    30
  102   -20   -11    48    48    25
   20    15   -68    55    54


5000 **********
   -5  -172    -7   -43   -33   249
   51   151   -77  -110    48   -21
   56   -31   114    -2   107   -16
   51   101   137    64     4    53
   52


6000 **********
   -6  -150    -8  -140   160    48
   78  -126    -5  -171    47   -22
  129    61   146  -134  -160   -84
  110   106    53    32   -98    95
  153    58    21    47    46


7000 **********
   -7  -142    -9    88  -168    62
  326    70   107   -89  -138   -49
   33    80    52     3   324   -93
  108    54   153   121   102   199
    3  -279  -136   -26   -59    93
   58  -108    -7  -249   251   139
 -362   -49    50    49   141  -102
   35    52    51


8000 **********
   -8  -179   -10  -198   -81   -41
  -37   120    -9   -48    55    23
   53    67   -26    -3  -107    97
   41    62    -2    51    50


9000 **********
   -9  -151   -11    62   -71   139
  150  -258  -312   224  -250  -166
   59    75  -258   107  -242    47
 -286   105    79    83     4    52
   41  -104    87    30    76    63
  159  -160    65    53  -102   -22
  -64    -6   -99    47    -6  -139
 -116   -20  -214    42   -88    46
   45


10000 **********
  -49   -89   -47  -292  -106  -179
 -179  -274   -54    52  -169  -158
 -105  -284  -119    45   -63  -101
 -100


11000 **********
  -48   192   -46  -119  -213  -128
 -206    14    21  -178  -105  -220
 -225  -158    47   -48  -120    23
  -99   -98
```



```
12000 **********
  -47   111   -45  -118  -232  -292
 -126  -180  -178   -91  -220  -222
 -155    47   -48  -120   -59
```

Budget

```
10000 **********
  -49   -39   -47  -514  -248  -171
 -226  -239  -246  -242  -153  -151
 -192  -140  -182   -60  -122  -213
  -16  -158    41    57   301   -63
  -54    85   205  -291    29    46
 -105   -42   -66  -111   -37  -105
 -104


11000 **********
  -48   -46   -46   -22  -140  -117
 -195    37  -288   126  -128   -88
 -139     3   -49  -248   -31  -202
  -48  -174  -112  -216  -175   -96
 -139   252   224   144   148   193
  270   226   189    66  -124   192
 -253   -45   -77   -81  -102  -101


12000 **********
  -47   226   -45  -118  -291    89
 -165  -287    33  -177    50  -185
 -155   232  -207  -175  -381  -252
   17    33  -223   294  -149  -401
 -147  -161   220  -144    53    71
 -287  -287  -335  -148  -130   296
 -281   -15    56   -99  -189


13000 **********
  -46    69   -44  -175    30    26
 -187  -278  -132  -112   -66     0
 -234  -159    29   -68   -88   -74
 -209  -207   -71   -72   -77    47
 -338  -144  -126  -101   -98   -97


14000 **********
  -45   121   -43  -429    61    -9
 -117  -185  -182   146  -217  -396
   49  -143  -178  -103    52  -122
  -72  -185  -190   137  -216  -186
   50    -1  -165  -122    53   -47
 -186  -185


15000 **********
  -44   169   -42   -23  -151  -117
 -100  -208   168    58    47  -214
 -185  -184     3     6  -181    54
  -99     2  -184  -183


16000 **********
  -43   116   -41    92  -113   275
  185   -18  -319   318   -67    92
 -184     0   -61  -123   -22   -99
 -118    52  -128   -84  -263    58
 -213  -185    34   -54   -95   -94


17000 **********
  -42   245   -40  -113  -192    94
 -325    20    38   -98    93   205
 -125   -24  -181    83  -159  -118
 -189  -422  -112  -347   -31    52
  -68  -112    54   -43   -89   -58
  -68  -114    69  -188   -99


18000 **********
  -41   201   -39  -112  -153   -61
 -285    -9   -76    48    -9   -72
  147    28   -96   -95


19000 **********
  -40    48   -38  -161  -121   162
 -188  -181  -148  -114    57     8
 -115  -188    55  -123   -93   -92
```

```
20000 **********
  -50    84   -48  -121  -234    15
 -127   -81     6   -76  -113  -165
   -1   -82  -189    46   -86  -182
 -181


21000 **********
  -49   122   -47  -120  -126    49
 -128   155  -182  -163  -185  -221
 -183    47    47   -68   -48  -181
 -188


22000 **********
  -48   180   -46  -120   -80  -188
  -61  -127    93  -181    41    36
 -131   -38  -288   -69  -181  -188


23000 **********
  -47   228   -45  -224  -117  -381
 -196  -155  -185    54    27    53
 -184  -183


24000 **********
  -46    80   -44  -289  -198  -176
 -173  -271   -51    55   117  -256
 -146   -82  -181  -188


25000 **********
  -45   283   -43  -198  -235  -243
   96    49   -95   249  -112  -122
  -76  -181   173    28  -228    29
 -193  -114   -19  -184   199  -288
 -129  -143    29   -98   -97


26000 **********

  -44   196   -42  -115  -289  -124
 -282    18    25  -322  -166   -24
  128   -97  -129  -212  -269    56
   56  -287  -195  -195    49  -185
   58  -118    25   -97   -96


27000 **********

  -43   115   -41  -114  -228  -288
 -122  -176  -123  -155   -21   128
  -97  -126  -212  -266    56    56
 -287  -181  -192    49  -185    58
 -118   -63   -56   -96   -95
```

Checksum

```
1000 **********
  -43   -95   -45   478    93   119
 -168    85   119  -212    53   -19
   43     4    92   289    88    55
  384   263    83    63    57    56
 -157    45    98    98   -21    75
  128    63  -197   -96    56    55
  187    38    52    51


2000 **********
  -44  -263   -46  -283   172    91
   54   -78   -79   187   166   139
   95    93    57   -92    51    58


3000 **********
  -45  -229   -47     4   112  -143
 -159    94    -4    98    68    59
  -96   -56
```

INDEX

&
&& 69, 76

A
Accountant 158
ADATE 6
Anaclock 1

B
Banker 145
BEEP 32, 87, 95
Binary 69, 72, 84
Binary search 106, 112, 126
Budget 171

C
Channel 154
Character memory 72, 73
Characters 71
Checksum Generator 194
Circle defining 7
CIRCLE command 47
Clock 14
CLOSE 55
Command lines 12
Control module 12
CSIZE 80

D
DATA 61, 91, 92
DATE 6
DATE$ 5, 6, 9, 21
Deleting items from arrays 115, 167
Designer 41
Directory 56

E
ENTER 24
Event 33
EXIT 13

F
Field 99, 116
FILL 11, 18
Formatting a number 151

G
GOTO 12, 13

I
Indicator characters 162
Initialisation 3, 42
INKEY$ 23
INPUT 24
Inserting into arrays 109, 147, 161
INSTR 94, 113
Internal clock 6
Inverting characters 81

L
LBYTES 85
Loading data from microdrive 55

M
Menu 24
MERGE 39

Microdrive 3
Mirroring a character 82
Mode 12, 13, 15, 71
Modular programming 12
Multiq 129
Music 90

N

Negative numbers 152
Nnumber 117
Note values 91

O

OPEN_IN 56
OPEN_NEW 55
OVER 43, 46, 176

P

PEEK_L 75
Pixel 71
Pixel colours 65
Pixels
  spacing of 16
Pointer array 103
Printer 35, 63, 66, 67, 154
Procedures 4
  passing parameters to 11
Program format x

R

RAD 8, 47
RAM 73
RECOL 71
Record 99
REPEAT 5, 13
RESPR 75, 86
RESTORE 61
RETRY 6
ROM 73
Rotation of a shape 47
Rounding errors 151
RS232C 38

S

Saving the program 2
SBYTES 85
SCALE 53
Scientific notation 150, 151
Screen 63
Screen memory 64
Screen protection 32
SCR_ 154
SDATE 21
SER1 38, 154
Setting the time 5, 21, 34
Shifting a character left 83
Sound Demo 87
Storing data on microdrive 54
String array 20
String comparison 100

T

TAB 176
3D Graph 56
Timer 19
Transferring the character set 74
Turning a character 83
Turtle graphics 56

U

UNDER 155
Unifile 98
User defined cursor 43, 77

W

WINDOW 80

Commodore 64 Disk Companion
David Lawrence & Mark England — ISBN 0 946408 49 1 — £7.95

The Working Commodore 64
David Lawrence — ISBN 0 946408 02 5 — £5.95

Commodore 64 Machine Code Master
David Lawrence & Mark England — ISBN 0 946408 05 X — £6.95

Machine Code Games Routines for the Commodore 64
Paul Roper — ISBN 0 946408 47 5 — £6.95

Programming for Education on the Commodore 64
John Scriven & Patrick Hall — ISBN 0 946408 27 0 — £5.95

Writing Strategy Games on your Commodore 64
John White — ISBN 0 946408 54 8 — £6.95

ELECTRON BOOKS

Graphic Art for the Electron Computer
Boris Allan — ISBN 0 946408 20 3 — £5.95

The Working Electron
John Scriven — ISBN 0 946408 52 1 — £5.95

Programming for Education on the Electron Computer
John Scriven & Patrick Hall — ISBN 0 946408 21 1 — £5.95

BBC COMPUTER BOOKS

Functional Forth for the BBC Computer
Boris Allan — ISBN 0 946408 04 1 — £5.95

Graphic Art for the BBC Computer
Boris Allan — ISBN 0 946408 08 4 — £5.95

DIY Robotics and Sensors for the BBC Computer
John Billingsley — ISBN 0 946408 13 0 — £6.95

Artificial Intelligence on the BBC and Electron
Keith & Steven Brain — ISBN 0 946408 36 X — £6.95

Essential Maths on the BBC and Electron Computer
Czes Kosniowski — ISBN 0 946408 34 3 — £5.95

Programming for Education on the BBC Computer
John Scriven & Patrick Hall — ISBN 0 946408 10 6 — £5.95

Making Music on the BBC Computer
Ian Waugh — ISBN 0 946408 26 2 — £5.95

DRAGON BOOKS

Advanced Sound & Graphics for the Dragon
Keith & Steven Brain — ISBN 0 946408 06 8 — £5.95

Artificial Intelligence on the Dragon Computer
Keith & Steven Brain — ISBN 0 946408 33 5 — £6.95

Dragon 32 Games Master
Keith & Steven Brain — ISBN 0 946408 03 3 — £5.95

The Working Dragon
David Lawrence — ISBN 0 946408 01 7 — £5.95

The Dragon Trainer
Brian Lloyd — ISBN 0 946408 09 2 — £5.95

ATARI BOOKS

Atari Adventures
Tony Bridge — ISBN 0 946408 18 1 — £5.95

Writing Strategy Games on your Atari Computer
John White — ISBN 0 946408 22 X — £5.95

SINCLAIR QL BOOKS

Artificial Intelligence on the Sinclair QL
Keith & Steven Brain — ISBN 0 946408 41 6 — £6.95

Introduction to Simulation Techniques on the Sinclair QL
John Cochrane — ISBN 0 946408 45 9 — £6.95

Developing Applications for the Sinclair QL
Mike Grace — ISBN 0 946408 63 7 — £6.95

Mathematics on the Sinclair QL
Czes Kosniowski — ISBN 0 946408 43 2 — £6.95

Quill, Easel, Archive and Abacus on the Sinclair QL
Alison McCallum-Varey — ISBN 0 946408 55 6 — £6.95

Inside the Sinclair QL
Jeff Naylor & Diane Rogers — ISBN 0 946408 40 8 — £6.95

GENERAL BOOKS

Home Applications on your Micro
Mike Grace — ISBN 0 946408 50 5 — £6.95

Sunshine also publishes

POPULAR COMPUTING WEEKLY

The first weekly magazine for home computer users. Each copy contains Top 10 charts of the best-selling software and books and up-to-the-minute details of the latest games. Other features in the magazine include regular hardware and software reviews, programming hints, computer swap, adventure corner and pages of listings for the Spectrum, Dragon, BBC, VIC 20 and 64, ZX 81 and other popular micros. Only 40p a week, a year's subscription costs £19.95 (£9.98 for six months) in the UK and £37.40 (£18.70 for six months) overseas.

DRAGON USER

The monthly magazine for all users of Dragon microcomputers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news related to the Dragon. A year's subscription (12 issues) costs £10 in the UK and £16 overseas.

MICRO ADVENTURER

The monthly magazine for everyone interested in Adventure games, war gaming and simulation/role-playing games. Includes reviews of all the latest software, lists of all the software available and programming advice. A year's subscription (12 issues) costs £10 in the UK and £16 overseas.

COMMODORE HORIZONS

The monthly magazine for all users of Commodore computers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news. A year's subscription costs £10 in the UK and £16 overseas.

For further information contact:
Sunshine
12–13 Little Newport Street
London WC2R 3LD
01-437 4343

Telex: 296275

Printed in England by Commercial Colour Press, London E7.

NOTES

NOTES

NOTES

NOTES