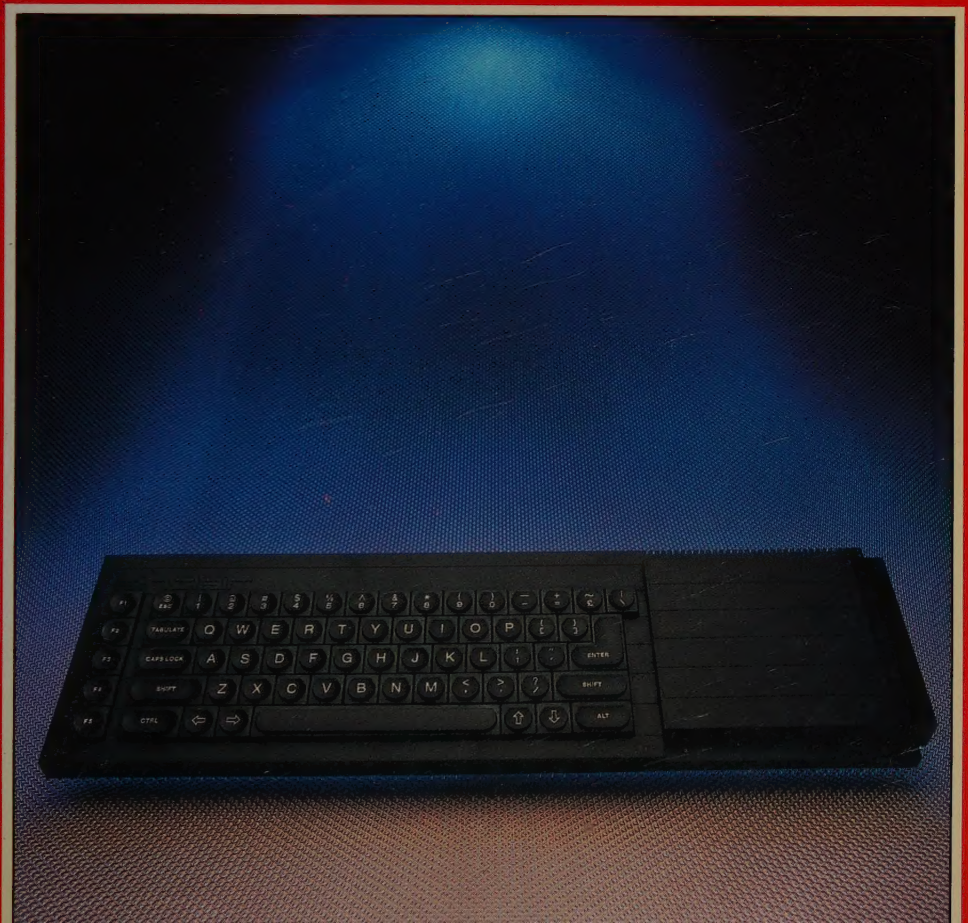
Tim Hartnell's

QL HANDBOOK



Interface Publications
LONDON · MELBOURNE

TIM HARTNELL'S QL HANDBOOK





TIM HARTNELL'S QL HANDBOOK

By Tim Hartnell with Andrew Nelson

INTERFACE PUBLICATIONS LTD., LONDON AND MELBOURNE

First published in the UK by:
Interface Publications Ltd,
9-11 Kensington High Street,
London W8 5NP

Copyright © Tim Hartnell, 1984
First printing November 1984

ISBN 0 947695 19 2

The programs in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. While every care has been taken, the publishers cannot be held responsible for any running mistakes which may occur.

ALL RIGHTS RESERVED

No use whatsoever may be made of the contents of this volume – programs and/or text – except for private study by the purchaser of this volume, without the prior written permission of the copyright holder. Reproduction in any form or for any purpose is forbidden.

Books published by Interface Publications are distributed in the UK by WHS Distributors, St. John's House, East Street, Leicester LE1 6NE (0533 551196) and in Australia and New Zealand by PITMAN PUBLISHING. Any queries regarding the contents of this volume should be directed by mail to Interface Publications, 9-11 Kensington High Street, London W8 5NP.

Interface Publications has exclusive rights to the names used for programs in this book. Portions of this book have appeared in the volume entitled "Exploring the Sinclair QL – An Introduction to SuperBASIC". This earlier book is now out of print. Sinclair, QL and ZX Microdrive are trademarks of Sinclair Research Ltd.

The copyright in SuperBASIC belongs to Sinclair Research Ltd. Copyright © 1983 in SuperBASIC Sinclair Research Ltd. Limited Copyright © 1984 Tim Hartnell. SINCLAIR QL, SuperBASIC etc. are trade marks of Sinclair Research Ltd.

Printed by Heffers Printers Ltd Cambridge England

Contents

Introduction	vii
One – First Steps on the QL	1
Two – Care and Feeding of Microdrives	9
Three – Concepts of SuperBASIC	13
Four – Identifiers	19
Five – Operators	25
Six – Strings	29
Seven – Arrays	35
Eight – First Steps in Structured Programming	43
Nine – Functions and Procedures	55
Ten – Graphics	59
Eleven – Coordinate Systems	73
Twelve – User-Defined Graphics	85
Thirteen – QLogo	89
Fourteen – Windows	107
Fifteen – The Sound of Music	111
Sixteen – Dealing with DATA	119
Seventeen – Extending your Vocabulary	123
Eighteen – Using Files	133
Nineteen – Financial Modelling	135
Twenty – Improving your Programming	143
Twenty-One – The FORTH Engine	157
Twenty-Two – Simulating Reality	197
Twenty-Three – Creating and Playing Adventures	207
Appendices:	245
One – Multi-tasking Demonstration	247
Two – Mathematical Functions	249
Three – How it all began	251
Four – Introduction to the 68008	253
Five – Error Messages	257
Six – Glossary of Computer Terms	259
Seven – Independent QL Users' Group	269
Eight – Making Connections	271
Nine – Further Reading	283

Contents

Introduction 1

One – First Steps on the QL 7

Two – Care and Feeding of Microdrives 13

Three – Concepts of SuperBASIC 19

Four – Identifiers 25

Five – Operators 29

Six – Strings 35

Seven – Arrays 45

Eight – First Steps in Structured Programming 49

Nine – Functions and Procedures 55

Ten – Graphics 59

Eleven – Coordinate Systems 75

Twelve – User-Defined Graphics 85

Thirteen – Clocks 91

Fourteen – Windows 107

Fifteen – The Sound of Music 115

Sixteen – Dealing with DATA 129

Seventeen – Extending your Vocabulary 133

Eighteen – Using Files 143

Nineteen – Financial Modelling 153

Twenty – Improving your Programming 163

Twenty-One – The EXCELL Engine 167

Twenty-Two – Simulating Reality 177

Twenty-Three – Creating and Playing Adventures 201

Appendices 245

One – Multi-tasking Demonstration 247

Two – Mathematical Functions 259

Three – How it all began 281

Four – Introduction to the 68008 283

Five – Error Messages 287

Six – Glossary of Computer Terms 289

Seven – Independent QL Users' Group 309

Eight – Making Connections 311

Nine – Further Reading 313

Introduction

The QL is an exciting one when looked at as a piece of 'state of the art' hardware. But the firmware, the built-in language, is in my opinion far more exciting and far more important.

SuperBASIC provides, for the first time on a popular micro-computer, a language of immense power and elegance. It is a language which encourages 'structured programming' (a term you'll soon understand, if you don't know exactly what it means at the moment); a language which helps you program cleanly, simply and well. The programming techniques you'll learn by working with SuperBASIC will help you no matter which other high-level computer languages you study in the future: Pascal, FORTH, Modula 2 or whatever.

In this book, I've tried to make SuperBASIC as easy to learn, and as simple to *appreciate* and *apply* as possible. I've made few, if any, assumptions about the level of programming proficiency you now possess. Therefore, even if the QL is the first computer you've ever programmed, you'll find a clear path from 'first steps' to 'structured programming' in this book. If you have worked with other machines, you'll find the programming skills you already possess can easily be adapted and extended to take advantage of the power and clarity of SuperBASIC.

The book is divided loosely into a number of sections. First we'll look at the fundamental components of SuperBASIC – the 'building bricks' from which your SuperBASIC programs will be constructed – and see how they work in isolation. The bricks will then be used to build 'structures', as we develop game and utility programs which, I feel, have a considerable degree of merit in their own right. That is, they are not just 'guess my number' standard programs which, although they may teach a thing or two about the use of a particular programming word or concept, are not in themselves particularly worthwhile or interesting programs. I hope the major programs in

this book have an intrinsic value which transcends their use purely as teaching devices.

As the major programs are built, I'll be taking slight detours down fascinating sidetracks, examining things like SuperBASIC's sound and graphics potential. We'll play music, define our own characters, and much more down some of these side roads.

Once you have the building bricks firmly under your belt, and are conversant with the ways structured programs can be made up from these fundamental elements, we'll introduce two other computer languages which can be emulated (after a fashion) in SuperBASIC. In the first, we'll construct a *FORTH Engine* which will allow you to study 'reverse Polish notation', as used in Forth (and on some stack-orientated calculators). This 'FORTH' is only the tip of the FORTH iceberg, but it does serve as an introduction to the language.

The second 'language' we'll look at is Logo. Now, as you know, the QL comes equipped with 'turtle graphics' as part of its vocabulary. However, these turtle graphics are only a small part of the rich language of Logo. With the program *QLogo,* I've emulated a much larger portion (though far from the whole) of 'real' Logo. It allows you to program the QL using the vocabulary of standard Logo's, producing a fascinating wealth of effects.

This *QLogo* has been included, as was the *FORTH Engine,* to not only give me an opportunity to demonstrate further aspects of SuperBASIC in action, but also to give you a worthwhile and enjoyable tool for further experimentation and learning.

Once we've forded the streams of these two mini-languages, we'll end the book by examining the production of simulation and adventure programs in some detail. Not only will these help you develop your own versions of these most popular of all computer games, but will give you ready-to-run programs of a reasonably high standard.

These categories which I've mentioned dovetail into each other. However, the sections are not as cleanly separated as this preamble may suggest. Rather, the major topics mentioned above are designed to give you an overall feel of the direction the book takes, rather than suggest that it is rigidly divided into mutually exclusive sections. As

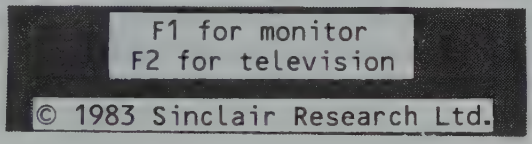
you'll see as you work through the book, the various topics overlap and interweave. This, if nothing else, indicates the harmonious nature of SuperBASIC, as its integral ideas permeate the whole of the language and its implementation.

Tim Hartnell,
London

Chapter One
First Steps on the QL

When you first turn your QL on, you'll see the screen fill very briefly with vertical red and green stripes. These are replaced with what I'd call 'hi-res rubbish' while the QL does a little internal checking.

Next, the screen clears to black, and the following message appears at the bottom of the screen:



If you press function key one (to indicate you have a monitor connected to the socket marked RGB) the screen will clear to show a rectangular area which fills about two-thirds of the screen. The left half of this rectangle is white, and the right half is red. If you press function key two (to tell the QL you have a TV attached to the UHF socket) the screen will clear completely to white.

The Most Important Keys

There are four keys with which you should familiarise yourself when you first start programming the QL.

The ENTER key is the one shaped like a 'backwards letter L', three up from the bottom on the right hand edge of the keyboard (just in from the edge of the Microdrive housing). You use this when you want the QL to do something. If you've entered a direct command into the machine, such as PRINT 6+ 7, you press ENTER to tell the

QL you want it to go ahead and do what you've told it (in this case, to print the number 13).

You also use ENTER after a command such as RUN (to start a program running) or NEW (to empty the computer of a program which has been in it). ENTER in this situation, as in the one mentioned in the previous paragraph, tells the QL you want it to actually carry out the latest command.

The third use for this key is when you are typing in a program. When you have finished typing a line on that black area below the rectangle on the screen, and you want the QL to accept that line and add it into the program, you press the ENTER key to get this message across.

There are two shift keys on the QL, one below ENTER on the right hand side, and the other above CTRL on the left hand side, the second key up from the bottom, on the second row in from the side. If you hold down one of the SHIFT keys, you'll get upper case (capital) letters when you type on the keyboard (or, if there is something else engraved on the key, such as the $ sign above the 4, you'll get that).

Instead of holding down SHIFT each time you want an upper case letter, you can engage the CAPSLOCK (above the left hand SHIFT key). All letters generated by the keyboard, once CAPSLOCK has been engaged, will be uppercase (unless you press SHIFT, in which case they will be lower case). Press CAPSLOCK again to disengage it.

If you want to erase a letter you have just typed, you hold down the CTRL (for 'control') key, which is below the left hand SHIFT, and then press the 'back arrow' key (to the left of the space bar). The CTRL key is used, often in combination with the ALT key and an alphabetic one, to trigger specific actions from the computer.

Lines and Statements

A statement in SuperBASIC tells the QL to do something, either immediately, or later on when the program containing that statement is RUN. If the statement starts with a line number, the

computer knows that this is part of a program, and will not execute it until the program is RUN.

If, however, you don't have a line number in front of the statement, the QL assumes you want it to act on that statement as soon as you press the ENTER key.

Therefore, if you type in PRINT "HELLO" then press the ENTER key, the word HELLO will be printed on the screen. If, however, you type in 10 PRINT "HELLO" and then press ENTER, the line will simply move to the upper section of the screen to show it has been accepted as a program line. You now have to type in RUN, then press ENTER again, for it to actually print HELLO on the screen for you.

A direct statement which is designed for immediate execution can be made up of more than one single statement. You used a colon (:) to join the separate sections together. Therefore PRINT "HELLO": PRINT "GOODBYE" can be entered as a direct statement, which will result – when you hit the ENTER key – in the words HELLO and GOODBYE appearing on the screen.

Keywords

SuperBASIC has a number of inbuilt keywords divided into three groups: inbuilt procedures, inbuilt functions and commands. You can add to them by defining procedures.

These are the inbuilt procedures:

ARC, ELLIPSE, LINE, POINT, SCALE, BEEP, INPUT, PAUSE, PRINT, BAUD, CLOSE, COPY, COPY_N, DELETE, DIR, FORMAT, OPEN, OPEN_IN, OPN_NEW, POKE, POKE_W, POKE_L, RANDOMISE, CLEAR, CONTINUE, EXEC, EXEC_W, LBYTES, LIST, LOAD, LRUN, MERGE, MRUN, NEW, RETRY, RUN, SAVE, SBYTES, SEXEC, STOP, AT, BLOCK, BORDER, CLS, CSIZE, CURSOR, FLASH, INK, OVER, PAN, RECOL, SCROLL, STRIP, UNDER and WINDOW.

The inbuilt functions are these:

BEEPING, INKEY$, KEYROW, ABS, ACOS, ACOT, ASIN, ATAN, COS, COT, EXP, INT, LN, LOG10, SINC, SQRT, TAN, PI, CHR$, CODE, PEEK, PEEK_W, PEEK_L and RND.

The QL's inbuilt commands are as follows:

FOR, REPeat, SELect, ON, IF THEN ELSE, DEFine PROCedure, DEFine FuNction, RETURN, DATA, END FOR, END IF, END REPeat, END DEFine, END SELect, EXIT and NEXT.

Note that when entering commands, you do not have to enter the word (or words) in full as above. So long as you enter the portion in upper case (such as REP for REPeat) the QL will supply the rest.

Program control commands and functions:

CLEAR
CONTINUE
DLINE list of ranges
EXEC name
EXEC_W name
LBYTES name, start address
LIST (#N,) list of ranges
LOAD name
LRUN name
MERGE name
MRUN name
NEW
RENUM (oldline (To oldline);(start)(,increment)
RETRY
RUN (line number)
SAVE name (list of ranges)
SBYTES name, start address, length
SEXEC name, start address, length, data space
STOP

Miscellaneous commands and functions:

CHR$ (character code)
CODE (string)
MODE 4/8/256/512
PEEK (address)
PEEK_L (address)

PEEK_W (address)
POKE address, byte
POKE_L address, word
POKE_W address, long word
RND [([m TO]n)]

Graphics Modes

We'll be looking at graphics modes in detail later on. However, as a brief introduction, it is useful to know there are *two* modes. MODE 256 (or MODE 8) is the lowest resolution mode, with (as you've probably realised) 256 pixels (points or 'picture elements', or dots) across the screen. The higher mode is MODE 512 (or MODE 4) which gives you double the resolution of MODE 256 (but halves the number of colours you can display).

The QL goes into the 256 setting if you press the function key to indicate you have a television attached and displays a square, purple flashing cursor. If you tell it you have a monitor attached, the computer goes into the higher mode, with a thin rectangular cursor, which is flashing red.

Colours

You can display eight colours (blue, red, magenta, green, cyan, yellow, white and black) on the 256 screen, while only four are available (black, red, green and white) with the 512 setting.

Many other colours can be created using the various 'stipple' combinations. There are four stipple patterns (vertical bars, horizontal bars, large foreground dots and small foreground dots) which allow an enormous range of colours to be created.

If you're using a black and white television or monitor, you'll see eight scales of grey in the 256 x 256 mode, and four scales of grey in the 512 mode. The number of characters on the screen is determined by your pixel setting. The default mode, when you're using a monitor, is 25 lines of 85 characters. The normal format, with a television, is 37 to 60 columns, depending on the software.

The Real-time Clock

The QL has a feature which is found in few other computers – a real-time clock. The DATE$ command is used to access the clock (as in PRINT DATE $).

You'll see, when you do this, that the clock gives a full calendar, as well as the time (on a 24-hour clock). It follows this format:

day-month-year hour-minutes-seconds

You set the on-board clock using the SDATE command (or sdate, as the QL allows you to enter your commands in upper case, lower case, or a combination), as follows:

```
sdate 1984,09,19,08,20,03
```

Type this enter, and follow it by pressing the ENTER key. Wait a few moments, and then type in PRINT DATE$, and you'll see something like this:

```
1984 Sep 19 08:20:56
```

Wait a few more moments, and PRINT DATE$ will give you a readout along the following lines:

```
1984 Sep 19 08:21:33
```

To set the date, and to extract parts of it, you can use the following program:

```
10 REMark Set DATE$ and use parts of it
20 CLS:CLS #0
30 CSIZE 3,1
40 SDATE 1986,1,1,1,0,0
50 PRINT DATE$
60  REPeat timing
70    G$=DATE$
80    AT 3,6
90    PRINT G$(16 TO)
100 END REPeat timing
```

Superclock

This program, originally published in the Independent QL Users' Club magazine *Quanta,* creates a window which fills the entire screen, and places the day, date and time on init a readily 'human-understandable' form. It makes a good display in a place where you need a program constantly running. It looks particularly effective with white INK on red PAPER.

```
100 REMark      SuperClock
110 REMark   From QUANTA 1/7
120 :
130 clock
140 :
150 DEFine PROCedure clock
160  WINDOW #1,512,256,0,0
170  CLS
180  CSIZE 3,1:OVER 0
190 :
200 REPeat tick
210  a$=DATE$
220  AT 6,7:PRINT DAY$;" ";a$(6 TO 11
);" ";a$(1 TO 4)
230  b$=a$(13 TO 14):c$="am"
240  IF b$>12 THEN b$=b$-12:c$="pm"
250  IF b$<10 THEN b$="0" & b$
260  a$(13 TO 14)=b$
270  AT 4,9:PRINT a$(13 TO);" ";c$
280  IF INKEY$<>"" THEN EXIT tick
290 END REPeat tick
300 :
310  WINDOW #1,422,192,40,18:CLS:CSIZ
E 1,0
320 END DEFine clock
```

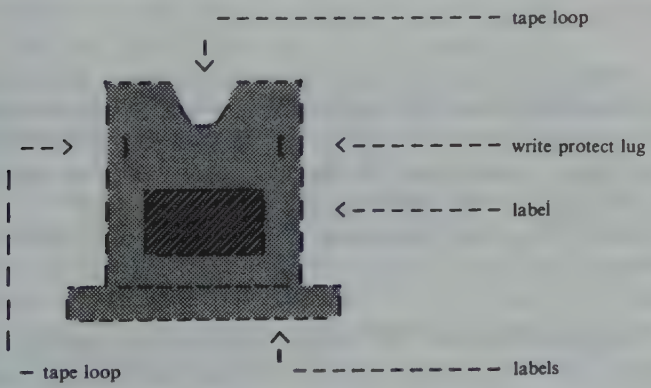
Chapter Two
The Care and Feeding of Microdrives



The QL uses microdrives to store programs, which would otherwise be lost when you turned off the power. It is essential to know a little about them now, so you can save the programs given in this book.

You should format new microdrive cartridges a few times in order to condition the tape. You do this with the command FORMAT which is used in the following way: FORMAT MDV1_name_of_cartridge. Note that if you FORMAT a used cartridge, all the original information will be wiped from it.

These are the main parts of a microdrive cartridge:



Because microdrives are the major form of memory on your QL, you should treat them with care. Always back up really important material, just in case something goes wrong with the cartridge you're

using. As each cartridge can hold around 100K, you could set aside a couple of cartridges just for backup. It should take you quite a while to fill the backup ones.

Inside the cartridge is a loop of tape which is around 200 inches long. It spins around inside the little plastic case at a speed of around 1.6 miles an hour. This may not sound very fast, but it means the tape is racing past the read head at nearly 101,000 feet per hour.

Even though the tape inside the cartridge is video tape, it is not invulnerable, and needs to be treated with care. Don't touch the tape. If a bit of tape appears at those points shown on the diagram marked 'tape loop', push it back carefully into the cartridge, but don't touch it with your hands to do so.

You can screw up the data on the cartridges by repeatedly sticking them in and out of the QL without using them, or by turning the computer off or on with them in place.

Generally, respect the fact that the tiny cartridges are pretty fragile beasties, and need to be treated with care.

You can hook up to six additional microdrives to the QL. The command CAT (used in the form CAT MDV1 or CAT "MDV2") is used to get a list of the files on the cartridge in the designated Microdrive.

The COPY command is used to copy information from one point in the system to another. You link the source channel or device to the destination one with the word TO, as in COPY MDV1 TO MDV2. If you want to wipe a particular file from a cartridge, you use the command DELETE in the following format: DELETE
command DELETE in the following format: DELETE MDV1_name_of_file.

To load a program into the QL from a microdrive, you use the command LOAD MDV1_name_of_file. If you want the program to start running automatically after loading (like the CHAIN command on some computers such as the BBC Micro), use LRUN instead of LOAD. You save programs with SAVE MDV1_name_of_file.

Let's try a few simple experiments to get you used to using the microdrives.

First, get a new cartridge (one which does not have any program on it). You *(format)* this cartridge, to get it ready for use, by typing in the following:

```
format mdv1_
```

Put the new cartridge in the microdrive slot nearest to the centre of the computer. Once you've typed in the above line, press the ENTER key, and wait while the red light comes on, and the tape whirls around inside the cartridge. In due course, a message like this will appear on your screen:

```
217/219 sectors
```

This means that 217 out of 219 sectors of the tape are suitable to use for storing programs. I suggest you go through the FORMAT process a few times. You'll probably find that more segments of the tape become available with subsequent formatting runs.

Once the tape has stopped whirring around, enter the following program, pressing the ENTER key after each line:

```
10 CLS
20 PRINT "This is a test program"
30 PRINT "on my dull grey QL"
```

You can now save this program on your microdrive by typing in the following, and pressing ENTER:

```
save mdv1_test
```

You can check what is on the cartridge with the following line (you must press the ENTER key after this line, and after each program line, but I will not bore you by pointing this out every time from now on, I'll just assume you'll do it):

```
dir mdv1_
```

A message like the following will appear on your screen:

```
216/219 sectors
test
```

Now that the program is on the microdrive cartridge, we can wipe it from the computer's memory, by typing in the word NEW. The program will vanish, which you can prove by entering LIST, which (when a program is present in the QL's memory) prints the listing of it on the screen.

To get the program from the microdrive into the computer, type the following:

```
load mdv1_test
```

There'll be a few moments whirring, and then the tape will stop spinning. Now try typing LIST. If you were successful, the listing of your program will appear. And that's all you need to know at this stage in order to be able to use the microdrives successfully.

Chapter Three
Concepts of SuperBASIC

SuperBASIC is a programming language developed by Sinclair Research which may appear at first – as its name suggests – as merely a 'souped-up' version of other BASICs. Certainly, it owes quite a bit to Spectrum BASIC (and you can, if you like, program the QL more or less as a Spectrum), but SuperBASIC is much, much more than this.

In fact, it is so far from BASIC, when used to the full, that Sinclair says they seriously intended calling it something else, but thought that people would be put off the machine if it meant having to learn a new language in order to program it.

The four fundamental components of SuperBASIC are:
- identifiers
- the ability to group and manipulate related elements
- coercion
- slicing

To help with the transition from other BASICs to programming in SuperBASIC, there are a number of familiar keywords from other BASICs within SuperBASIC which are not totally necessary. As well as easing your transition from your former BASIC to the QL BASIC, these 'less than vital' words also allow you to type in programs written for other machines with minimal changes. However, to make the most of the computer, and to use its language to the full, I suggest you get into the habit of replacing the superfluous functions with those from SuperBASIC.

GOTO can be replaced with IF, REPeat and the like, and GOSUB can be replaced with a call to a procedure (defined with DEFine PROCedure). ON . . . GOTO (which has not been provided with previous Sinclair machines) can be replaced in SuperBASIC with

SELect, as can ON . . . GOSUB.

Many other commands will be familiar to you, and perform in SuperBASIC just as they do in other BASICs. These include such things as PEEK and POKE, LIST, READ and DATA. Others, such as REMark and SQRT, fairly obviously take the place of the more familiar forms (REM and SQR).

There will be some commands which won't work at all, such as LLIST which is used on most computers to print a listing out on a printer. You accomplish this on the QL by opening a channel to the printer, and then using LIST, which is as happy to LIST to the printer as it was to LIST to the screen.

You'll find that you can get the QL to do anything other computers using BASIC can do (and, of course, a great deal more), although it may take a bit of fishing to find the command combination you need. Common commands not supported by the QL include the LLIST we've just discussed, LPRINT, VAL, STR$, IN, OUT, ON ERROR and IN INPUT.

As I pointed out, you get the QL to LIST to a printer by using LIST #, indicating the device number. LPRINTing is accomplished in much the same way, with PRINT #.

VAL and STR$ are not provided, because they are not needed, as you'll see shortly when we discuss a particularly useful feature of the QL called 'coercion'. As you probably know, VAL changes a number within a string to its numerical equivalent (that is, it changes "1" into 1, so it can be processed as a number). STR§ does the opposite, changing a number into a string (456 into "456"). These happen automatically when needed on the QL, with the computer forcing the element into the correct form required, so VAL and STR$ are not needed.

IN and OUT, commands provided on the Spectrum (as well as other computers) are not applicable to the Motorola 68008, the main processor inside the QL. (The second major processor is an Intel 8049, which is used to control the keyboard, generate the sound, and act as an RS-232-C receiver. There are also two custom-designed chips. One looks after the display and memory, and the other

controls the Microdrives, the local network, and RS-232-C transmission.)

Before we continue the 'heavy learning', here's a simple, but incredibly effective program, to enter and run on your QL. Just type it in just as it is, and sit back for an hour or two of entertainment. I won't explain what the various lines do at the moment. All I want to do is give you something worthwhile now, so you can see the power of the genie you now have at your command, and the kind of things it can do. (The program, by the way, is explained in the section on turtle graphics, as part of the material on *QLogo.*)

```
100 REMark Turtle graphics
110 PAPER 0
120 CLS:CLS #0
130 PENDOWN
140 REPeat logo
150  IF RND>.68 THEN CLS
160  note=RND
170  POINT 80,50
180  INK RND(1 TO 7)
190  turtle=RND(3 TO 70)
200  terrapin=RND(25 TO 340)
210   FOR k=1 TO turtle
220    MOVE k/.7
230    TURN terrapin
240    BEEP 3200,k/note
250   END FOR k
260 IF INKEY$<>"" THEN STOP
270 END REPeat logo
```

Type this program into your QL, pressing the ENTER key after each line, then type in the word RUN, and sit back and enjoy the effects. I suggest you try running it in a darkened room to enjoy it at its best. You can stop the program at any time just by holding down any key.

Numbers, Numbers, Numbers

If you started your computer life – as I did – with a ZX80, you'll remember that the highest number you could have was a paltry

32767 (and the lowest was -32767 or -32766 in some circumstances). In comparison with this, the QL has a range which is mind-boggling. Although the integer (whole number) limits are -32767 to +32767 (and variables to hold integers are designated as such by following the variable name with the percentage sign (such as A% for example), the floating point number range stretches off into a galaxy, far, far away.

The floating point number range stretches from -10 raised to the -615th power right through to 10 raised to the 615th power. You can have plus or minus 10 raised to the -615th power significant decimal digits. The QL defaults to this number of significant decimal digits on power up.

String literals can be up to 32768 characters long (and we'll be discussing strings in detail a little later on).

All previous Sinclair computers have had a limit of 1 (although a Ø could be POKEd) to 9999 for the line numbers which preceded statements in a program. The range for the QL is from 1 to 32767.

Running Programs

The QL automatically clears all variables (numerical to zero, and string to the empty string) when you type in RUN and press ENTER. You can preserve the values by starting the program with the command GOTO 1 (whether or not you have a line numbered 1).

If you wish to start program execution at a specific line, you can use GOTO n (where n is the program line) to preserve the variables, or RUN n, when variables will be cleared, but execution will begin at the line number specified (or the next highest number, if that line number does not exist).

If you break into program execution for some reason – such as to check the value of a variable – you can continue execution from the break point with the command CONTINUE.

Using the Screen Editor

You use the EDIT command, fairly obviously, when you want to edit a program line. You call up the editor by typing in a command of the following form:

EDIT nnn (where nnn is the line number)

If you leave off the line number, the computer will produce the lowest numbered line in the program. As you modify the line, it will be highlighted, to indicate that the QL's internal syntax error checking procedure has not yet been carried out on that line.

Once you press ENTER, to indicate that you've finished working with the line, the QL will check the syntax of the line. If it is error-free, then the line will enter the program, replacing the original ('pre-EDIT') version of that line. The QL is a hard taskmaster, and will not let you move off the line until it has passed the syntax-checking process.

If you are editing a line which is not completely on the screen, you will see that the line will automatically scroll, as you move the cursor along it, so that the part of the line you wish to work on is in view on the screen.

The material you are typing is added before the old material, and the old material moves along to make room for it.

The pre-EDIT version might be:

```
190 A=INT(M/Q):PRINT A
```

You move the cursor to just after the colon and add your new material as follows:

```
B=5*A
```

The whole line would then read:

```
190 A=INT(M/Q):B=5*A:PRINT A
```

The old material scrolls sideways to make room for the new.

If you're working on a line, in EDIT mode, and you change your mind, you can restore the original line unaltered simply by pressing the ESC key.

Chapter Four
Identifiers

You'll recall, at the start of the book, I said I'd be discussing the 'building blocks' of SuperBASIC, the fundamental concepts and components of the language, before showing how these could be combined to form SuperBASIC programs. The first such building block is the 'identifier'.

An identifier can be up to 255 characters long, but must start with a letter (or an ampersand in some cases, when it is assumed to be a system identifier; we will not be discussing this special case here). You can follow the starting character with any combination of letters and numbers and – most importantly – underscore characters can be used. If it ends with a $ symbol, then the identifier is understood to stand for a string. If it ends with %, it stands for an integer variable.

The underscore is particularly useful for allowing you to produce identifiers which unambiguously label a loop, function or whatever. For example, all the following are valid identifiers:

```
result
answer
the_final_result
end_of_game_flag
effect_2a
```

Make sure you do not confuse the underscore character (_) with the hyphen (-). This is easy to do, especially as you get them both from the same key, next to the zero key on the top row. Without holding down SHIFT, this key will give the hyphen. Hold down SHIFT, and the key will give the underscore character.

Note that the QL (in common with the Spectrum) does not

differentiate between upper and lower case for identifiers. This means that the following are all identical, so far as the QL is concerned:

answer_3e
ANSWER_3E
Answer_3e
Answer_3E
aNswER_3e

The identifier is a fundamental concept in SuperBASIC. You will find that as you become accustomed to working in the language, you will use identifiers in every program you write. Once you recognise the additional clarity given to your programs when you use such things as procedures (rather than GOTOs or GOSUBs), aided by the explicit identifiers, you will realise what a powerful programming tool SuperBASIC really is.

Note that although, in some circumstances, an identifier appears to behave as if it was a variable, it is not a variable. An identifier is the tag the QL places on a specific object it has stored. The QL has other pieces of information 'locked to' the identifier name. These include the nature of that which is identified (is it a string? a floating point number?), the specific example of the class of things identified which it actually has stored ("Joshua" or 778.45), and where the object is located in its memory.

Using identifier A, SuperBASIC can manipulate the values locked to that identifier, along with any values in related identifiers B, C and so on which are part of the definition of identifier A.

Coercion

I said earlier that VAL and STR$ were not needed on the QL, because of a feature called 'coercion', another of our building blocks.

In the vast majority of other BASICs, including those provided on the ZX81 and the Spectrum, providing the wrong type of input can lead to a system crash.

Consider the following program:

```
10 REMark Coercion Demonstration
20 CLS:CLS #0
30 PRINT "Enter a number"
40 INPUT a
50 PRINT "Now enter another one"
60 INPUT b$
70 PRINT "The total is ";a+b$
```

If you ran this on most computers, the program would crash in the final line. You are trying to effect an addition between a number and a string.

If, however, you ran that program on the QL, it would add the two together without any problems, provided that you pressed a number key when asked to enter the second number.

Say you press the "2" key. Although your input would be stored inside the QL as "2" (as opposed to 2, as a number), the computer would convert it (or, 'it would be coerced') into 2, so it could be added.

The QL will perform the coercion if it is possible to do so. Consider the following program, which would crash on most computers, but on the QL shows coercion acting in the opposite way to the first example:

```
10 REMark Coercion Demo II
20 CLS:CLS #0
30 PRINT "What is your first name?"
40 INPUT a$
50 PRINT "How old are you?"
60 INPUT b
70 LET c$=a$ & b
80 PRINT c$
```

Notice how there is an ampersand (&) in line 50, rather than a plus sign (+). This tells the QL that you want to work with strings, rather than with numbers. (Note also that the LET in line 50 is optional.)

When you run this on the QL, you'll see the following:

```
What is your first name?
Tim
How old are you?
167
Tim167
```

The last line is the result of adding two strings together. Of course, your age – in this case – was accepted and stored by the QL as a number, then coerced into a string so it could be added to the other string (your name). The word 'concatenation' is used to describe the addition of strings.

Coercion will work in all the following cases:

```
LET a="345" + 96
        (Gives answer 441)
LET a="13" + "56" + "99"
        (Gives answer 168)
LET a$=245 & 245
        (Gives answer "245245")
LET a$="345" & 96
        (Gives answer "34596")
```

When you are adding two floating point numbers together, and assigning them to a floating point variable (any combination of letters and numbers, starting with a letter, such as A8976 or BNGS5 or BN77BN), no coercion is needed, and none takes place.

Similarly, if you are assigning the result of adding two floating point numbers to an integer variable (any combination of letters and numbers, starting with a letter and ending with a % symbol, as A786%, B% or BHJHJ%), the two floating numbers are not coerced but the result of the addition is stored as an integer.

The use of the ampersand and the plus sign must be followed carefully. What do you think the output of these two programs would be when run on your QL?

PROGRAM A:

```
10 REMark Coerce A Demo
20 CLS:CLS #0
```

```
30 a$="6754.65"
40 b=152.76
50 c$=a$ + b
60 PRINT c$
```

PROGRAM B:

```
10 REMark Coerce B Demo
20 CLS:CLS #0
30 a$="6754.65"
40 b=152.76
50 c$=a$ & b
60 PRINT c$
```

Program A would return "6907.41", and program B would return "6754.65152.76". In the first program, the QL coerces the "6754.65" to the floating point number 6754.65 and, leaving B as a floating point number 152,76, adds them numerically, then converts the floating point answer back into a string.

In the second program, B, the first string stays a string, and the 152.76 is coerced into "152,76". Then, the two strings are concatenated. The + sign tells the QL you want to treat the variables as numbers and the & sign indicates you want the variables treated as strings.

Chapter Five
Operators

We'll now add 'operators' to our store of QL building blocks. Operators actually *do* things, like the plus sign between two numbers (in effect) *adds* the two numbers. The QL supports an impressive range of operators:

=	equal	numerical	– logical equal
		string	– type 2 comparison
==	equivalence	numerical	– "almost equal"
		string	– type 3 comparison
+		numerical	– addition
-		numerical	– subtraction
/		numerical	– division
*		numerical	– multiplication
<	less than	numerical	– less than
		string	– type 2 comparison
>	greater than	numerical	– greater than
		string	– type 2 comparison
<=	less than or equal	numerical	– less than or equal
		string	– type 2 comparison
>=	greater than or equal	numerical	– greater than or equal
		string	– type 2 comparison
<>	not equal	numerical	– not equal to
		string	– not equal type 3 comparison
&	ampersand	string	– concatenation
&&		bitwise	– AND
||		bitwise	– OR
^^		bitwise	– Exclusive OR

~ ~	bitwise	– NOT
OR	logical	– OR
AND	logical	– AND
XOR	logical	– Exclusive OR
NOT	logical	– NOT
MOD	numerical	– modulus
DIV	intager	– divide
^	floating	– raise to the power
(^)	integer	– raise to the power
-	unary minus	
+	unary plus (NOP)	

Imagine your QL comes across the following program fragment:

```
10 CLS:CLS #0
20 a=16 + 8*2
30 PRINT a
```

When you run this, the answer 32 will be given (16 plus 8 times 2). If, however, line 10 was changed as follows . . .

```
10 CLS:CLS #0
20 a=(16+8)*2
30 PRINT a
```

. . . the QL would produce an answer of 48 (twice the sum of 16 plus 8).

The difference, of course, was the brackets. In the second version of line 10, the brackets made sure that the 16 and 8 were added before the multiplication took place. In the first example, the multiplication took place first.

We say there is an 'order of precedence' which determines the order in which arithmetic operations will be carried out.

Here's the order of precedence on the QL (from highest to lowest):

unary plus or minus

string concatenation
exponentiation
multiply and divide (modulus and integer divide)
add and subtract
logical comparison
NOT
AND
OR and XOR

You can include logical operators (<, >, = and the like) within an expression. If you do this, and the expression is 'true' (such as A=3:PRINT A=A) then the computer will return a one. If the expression is false (such as PRINT 3<>3) then the QL will return a zero.

Chapter Six
Strings

The QL – like all computers running a version of BASIC – can deal with two kinds of information, *numeric* (or numbers) and *string*. A string variable or identifier is differentiated from a numeric one by the fact that the string variable ends in a dollar sign. As you'll see in the next chapter on arrays, a string array also uses a dollar sign to indicate that it deals with strings, rather than with numbers.

We first came across strings when discussing coercion. There we saw that the QL, in many circumstances, can handle numeric and string information in an intelligent way, changing the nature of the material to suit the demands of the circumstances.

In this chapter, we'll look at other ways strings can be used.

CHR$ and CODE

This pair of words perform complimentary functions. CHR$ (said aloud as 'char-dollar' or 'character-string') turns a number into its equivalent character.

That is, PRINT CHR$(46) will print a full stop. The CODE of the ASCII character "." is 46. If you want to get the CODE of a character, you just tell your QL to PRINT CODE "n", with your character in place of the n.

Type in the following and press ENTER:

```
PRINT CODE("B")
```

The computer will return the number 66. PRINT CODE "BATTLESHIP" will also return 66 as CODE takes just the *first* element of the string. Enter the following program, but before you

run it, try to predict what the answer will be:

```
10 TEST$="BANANA"
20 CLS:CLS #0
30 PRINT CODE(TEST$)
```

Once again, of course, you got 66, as the QL evaluated TESTS as though it was "BANANA" (to which it was assigned in line 10). Note that you don't need quote marks around the name of a string (GATE$, WORD$, A$ or whatever) in order to get the computer to return the code.

Every number, letter, sign and symbol the QL can print out has a CODE assigned to it. Here are some of the CODEs the QL uses for its defined character set:

```
  32   ! 33   " 34   # 35
$ 36   % 37   & 38   ' 39   ( 40
) 41   * 42   + 43   , 44   - 45
. 46   / 47   0 48   1 49   2 50
3 51   4 52   5 53   6 54   7 55
8 56   9 57   : 58   ; 59   < 60
= 61   > 62   ? 63   @ 64   A 65
B 66   C 67   D 68   E 69   F 70
G 71   H 72   I 73   J 74   K 75
L 76   M 77   N 78   O 79   P 80
Q 81   R 82   S 83   T 84   U 85
V 86   W 87   X 88   Y 89   Z 90
[ 91   \ 92   ] 93   ^ 94   _ 95
` 96   a 97   b 98   c 99   d 100
e 101   f 102   g 103   h 104   i 105
j 106   k 107   l 108   m 109   n 110
o 111   p 112   q 113   r 114   s 115
t 116   u 117   v 118   w 119   x 120
y 121   z 122   { 123   | 124   } 125
```

Slicing

Strings can be *sliced.* Run through the following program:

```
10 REMark String Slicing
20 CLS:CLS #0
30 a$="casino gambler"
40 PRINT a$
50 PRINT a$(1)
60 PRINT a$(2 TO 4)
70 PRINT a$(5 TO)
80 PRINT a$(1 TO 9)
```

The output this will produce is as follows:

```
casino gambler
c
asi
no gambler
casino ga
```

The first one – casino gambler – is no surprise. After all, we assigned a$ to the string "casino gambler" in line 30, so we would expect it to be printed out in full by line 40.

But what's happening in line 50? Sinclair string-handling allows us to treat parts of the string separately. We specify the section of string we want to use by following the string with a number (or numbers) in brackets.

Line 60, a$ (2 TO 4), produces "asi" as its output. The string-slicing mechanism looks in this case to the first number in brackets (in this example it is 2) and then extracts the portion of the string from this character number TO the next number (in this case 4) specified. If there is no second number within the brackets (as occurs in line 70 of the program) the computer just goes right to the end of the string, producing – as we can see in the output – "no gambler". Note that the space between the two words counts as a character.

How Long is a Piece of String?

The length of a string is returned by the LEN function, which you can demonstrate in action with the next program:

```
10 REMark LEN demo
20 CLS:CLS#0
30 REPeat cycle
40  INPUT "What is your name? ";name$
50  IF name$<>"" THEN EXIT cycle
60 END REPeat cycle
70 PRINT \\"Well, ";name$;", your name is"
80 PRINT LEN(name$);" characters long"
90 PRINT \\"'sausage' is ";LEN("sausage")
100 PRINT "   characters long"
```

NOTE: Throughout this book, we have enclosed strings within double quotes (''). This is not essential on the QL, which will accept a single quote (') just as happily. We have avoided this because few, if any, other BASICs use it, and we prefer to keep the single quote for when an apostrophe is needed within a string. However, you can use single quotes if you prefer.

Playing with the String-Man

We now come to our first game program in the book, to show a little more of string manipulation. Written by Scott Vincent, STRING-MAN is a version of the old favourite HANGMAN. The computer chooses a word (from those listed in lines 400 to 420) and then challenges you to guess it. Notice how LEN, which weve just looked at, is used in several places in the program (such as lines 290 and 200).

All you need to do with this program is enter it, and follow the instructions on the screen. You do *not* need to press ENTER after you select a letter. You'll notice in the program listing that some lines have a gap after the line number (such as 40, 50 and 60) before the program line itself begins. This is to make the program easier to read, as it breaks the listing up into a number of indented sections. The QL tolerates spaces well, and you enter these just by pressing the space bar as many times as necessary, before you start typing the line itself in. Alternitively, you can simply leave the spaces out, although this diminishes the clarity of the program.

Once you've played STRING-MAN several times in its present form, change the words in lines 400 to 420 to those of your choice to make the game more challenging.

```
10 REMark     STRING-MAN
20 REMark By Scott Vincent
30 REMark ********************
40   RESTORE
50   RANDOMISE
60   BORDER 2,3
70  CLS:CLS #0
80 AT 1,15:PRINT "STRING-MAN"
90 AT 2,15:PRINT "----------"
100   n=INT(RND*12)+1
110  FOR 1=1 TO n
120   READ B$
130 END FOR 1
140 a$=""
150 FOR 1=1 TO LEN(B$)
160   a$=a$&"_"
170 END FOR 1
180 REMark ********************
190 AT 15,1:PRINT"Incorrect Guesses:"
:X=15:Y=20
200 AT 5,20-LEN(a$)/2:PRINT a$
210 IF a$=B$ THEN GO TO 360
220 AT 10,5:INPUT"Enter Guess: ";
230 i$=INKEY$:IF i$<"A" OR i$>"z" THE
N GO TO 230
240 PRINT i$
250    PAUSE 64
260 IF CODE(i$)>CODE("Z") THEN i$=CHR
$(CODE(i$)-32)
270 AT 10,18:PRINT " "
280 V=0
290 FOR 1=1 TO LEN(B$)
300   IF B$(1)=i$ THEN a$(1)=i$:V=1
310 END FOR 1
320 IF V=1 THEN GO TO 200
330 AT X,Y:PRINT i$;",":Y=Y+2
340 IF Y=36 THEN Y=2:X=X+1
350 GO TO 200
360 FOR 1=100 TO 1 STEP -1
370   BEEP 30,1
380 END FOR 1
```

```
390 REMark ******************
400 DATA "MICROCOMPUTER","SINCLAIR","
GAMES","INNOVATIVE"
410 DATA "ENORMOUS","ARTIFICIAL","SAT
ISFACTION","SHADOW"
420 DATA "PASTURE","EMPIRE","BISCUITS
","URGENCY"
```

Chapter Seven
Arrays

Arrays form another, essential building block to add to our programming arsenal. An array is used when you want to set up a list of objects (numbers or strings) which you can access by referring to their *position* within the list.

You set up the array using the DIM statement.

An array is a set of memory spaces reserved within the QL. Any *element* of that array is referred to by giving the *name* of the array and a *subscript,* which indicates the position of the element within the array the element. To produce an array on the QL which will hold four numbers, you enter the line (assuming we are calling the array HOLD):

```
DIM hold(3)
```

If you wanted an array to hold 45 numeric elements, you'd use a statement like:

```
DIM name_of_array(44)
```

You can see that the number in brackets is one less than the number of elements we want to hold. This is because the first element of the array is referred to as element O.

Let's assume that you have set up an array of four elements with the statement:

```
DIM a(3)
```

When you dimension a numeric array like this, each element contains a zero. Therefore, the following program . . .

```
10 DIM a(3)
20 CLS:CLS#0
30 PRINT a(0)
40 PRINT a(1)
50 PRINT a(2)
60 PRINT a(3)
```

. . . would produce . . .

```
0
0
0
0
```

You could fill it as follows:

```
10 DIM a(3)
20 CLS:CLS#0
30 a(0)=10
40 a(1)=9.97
50 a(2)=898273
60 a(3)=65.8781
```

To refer to, say, the element holding 898273 in that array (in effect, to refer to 898273), all you have to do is refer to A(2). A program can manipulate A(2) as easily as it can any other variable.

The arrays we've looked at are *one-dimensional* arrays, which can be used as the one above demonstrated, for holding a list of numbers.

The QL, in common with most other computers, allows you to have arrays of more than one dimension. These arrays are referred to as *multi-dimensional* arrays. There is no limit to how many dimensions you can have in an array on the QL (apart from the limitation imposed by memory restraints).

When you dimension on array, the QL reserves space for every element of that array. The memory is reserved, whether or not every element is actually used and multi-dimensional arrays consume memory at a prodigious rate, although you've got quite a bit to play with on the QL, so you can feed an awful lot into it before you strike problems.

Anyway, a multi-dimensional array is defined by adding a second (or third, or whatever) number after the first, in the DIM line as follows:

```
DIM hat(3,2)
```

This sets up a four by three array. The contents of our one-dimensional arrays could be thought of as a list of numbers as follows:

```
89788
 2314
12314
45621
```

By contrast, a two-dimensional array (such as HAT above) must be thought of as a table. When it is first dimensioned, all the elements of the array are – as you now know – set to zero. Here, then, are the contents of the elements of HAT, just after dimensioning:

```
   ! 0 1 2 3
---------
0  ! 0 0 0 0
1  ! 0 0 0 0
2  ! 0 0 0 0
```

Each element within that array is referred to by giving *two* subscripts. For example, the element in the bottom right hand corner of that table is HAT(2,3).

String Arrays

The arrays we've been looking at, including COAT, HAT and the rest, have been numeric arrays, designed to hold numbers.

The QL also provides string arrays. These must be DIMensioned in a slightly different way. If we used a statement like the following . . .

```
DIM name$(6)
```

. . . it might be thought probable that this would create a string array in which we could store six complete strings. However, this is not so. Unlike most other non-Sinclair BASICs (for the following peculiarity is shared by both ZX81 and Spectrum BASIC), the QL only sets up a *single character* array with a DIM statement like the preceding one.

We could only use it as follows:

```
10 DIM gee$(4)
20 RESTORE
30 FOR h=1 TO 4
40  READ gee$(h)
50 END FOR h
60 DATA "h","e","l","l"
```

To store full words, we have to give the array a second dimension, with a number which is the length of the maximum length string we wish to store in the array. To show this, the next program sets up an array which can hold three strings of up to ten letters each:

```
10 DIM whiz$(3,9)
20 RESTORE
30 FOR g=1 TO 3
40  READ whiz$(g)
50 END FOR g
60 DATA "jeremy rus","t hartnell","li
z north"
```

Although it is necessary to add a second number to the DIMension statement when setting up an array for string use, it is not necessary to use the second element when specifying a string. The following program fragment should make that clear:

```
10 DIM golly$(2,5)
20 CLS:CLS #0
30 golly$(1)="hello"
40 golly$(2)="there"
50 PRINT golly$(1)
60 PRINT golly$(2)
```

The output of this program would be . . .

```
hello
there
```

. . . rather than . . .

```
h
t
```

. . . as you might have expected.

Slicing String Arrays

We saw in the previous chapter how slicing works with single strings. We'll now apply that knowledge to the elements of string arrays. Examine the following program, and before you run it on your QL, try to predict what the effect of running it will be:

```
10 DIM gate$(5,9)
20 CLS:CLS #0
30 gate$(3)="bill"
40 gate$(5)="swing"
50 gate$(0)="back"
60 gate$(1)="logic"
70 gate$(2)="rusty"
80 PRINT gate$(0)(1 TO 3)
90 PRINT gate$(1)(1 TO 4)
100 PRINT gate$(2)(4 TO)
110 PRINT gate$(3)(1)
120 PRINT gate$(4)
130 PRINT gate$(5)(5 TO)
```

Once you've made your prediction, enter and run the program on the QL to see if you were right. If you were, this shows you've been very quick to pick up the concepts of string-slicing in SuperBASIC.

Here's the result of running it:

```
bac
logi
ty
b


g
```

BAC – this is GATES$(0) (1 TO 3)
LOGI – this is GATES$(1) (1 TO 4)
TY – this is GATES$(2) (4 TO)
B – this is GATES$(3) (TO 1)
– this is GATES$(4) which was not assigned
G – this is GATES$(5) (5 TO)

You can assign slices of one string to another . . .

```
a$="sausage"(3 TO 5)
```

. . . will set A$ equal to "USA".

You can also change only *a part* of a string, using slicing as this program shows:

```
10 first$="Sinclair"
20 first$(3 TO 5)="BBC"
30 PRINT first$
```

When you run this program, your QL will produce this result:

```
SiBBCair
```

This shows that elements 3, 4 and 5 of the original string ("NCL") have been replaced with another string ("BBC").

Array Substitution

By now, it has probably occured to you that it might be possible to start swapping parts of string arrays. This can be done, as our next program shows:

```
10 DIM word$(2,4)
20 word$(0)="lazy"
30 word$(1)="trek"
40 word$(2)="cast"
50 word$(1)(3 TO)=word$(2)(1 TO 2)
60 word$(0)=word$(1)(4) & word$(0)
70 PRINT word$(0)
80 PRINT word$(1)
90 PRINT word$(2)
```

The output of that inspiring program is as follows:

```
alaz
trca
cast
```

What has happened to our original words$ Except for "cast", the original words in lines 20, 30 and 40 have been replaced. Following through lines 50 and 60 should show what has happened.

Line 5Ø sets the last two letters of the word$(1) equal to the first and second letters of word$(2). Line 6Ø sets word$(Ø) equal to the fourth letter of word$(1) and then tries to add *all* of the original word$(Ø) on the end of it. But there is not enough room, as the original array was only dimensioned for a word of four letters. In SuperBASIC, a string is *truncated* (that is, cut off) if there is no room for it within the array.

Chapter Eight
First Steps in Structured Programming

One of the most outstanding features of SuperBASIC is the encouragement it gives to you to construct your programs in a clean, structured manner. Although the older, less-elegant branching mechanisms exist (such as GOTO, GOSUB and ON ... GOTO/ON ... GOSUB) in SuperBASIC, they do so mainly to give a degree of compatibility with other BASICs.

I suggested earlier that, if you were familiar with another BASIC (especially if it is Spectrum BASIC), you could start by programming the QL more or less as if it was a Spectrum, and gradually introduce elements of QL BASIC into your work.

The control, looping and branching sections of your programs are the ones which will gain most from SuperBASIC.

The Structures

The QL vocabulary is particularly rich in this area, offering you the following:

IF	(THEN)	ELSE	END IF
SELect ON	(ON var) =		END SELect
FOR	(NEXT)	(EXIT)	(END FOR)
REPeat	(NEXT)	(EXIT)	END REPeat

Our old friends GOTO and GOSUB are available on the QL, although their use is not recommended:

GOTO
GOSUB RETURN
ON exp GOTO/GOSUB list

As well as these, the QL offers procedures, with local variables:

DEFine PROCedure/FuNction name [(list of parameters)]
LOCAL
RETURN [value]
END DEFine

Compatibility

We'll start by looking at the program flow controls with which you've probably had some experience, GOTO, GOSUB/RETURN, ON . . . GOTO and ON . . . GOSUB/RETURN.

Here's a simple program to show it in action:

```
10 PRINT "This is the QL ";
20 GO TO 10
```

As you would expect, this fills the screen with the following, until you break into the program:

```
This is the QL This is the QL This is
 the QL This is the QL This is the QL
 This is the QL This is the QL This i
s the QL This is the QL This is the Q
L This is the QL This is the QL This
```

In this program, the QL executes line 10, then moves onto line 20 where it finds the instruction GOTO 10. There are no conditions placed on this transfer of control from line 20 to line 10. It occurs every time. This is an unconditional branch.

GOSUB works in much the same way, except that it works in tandem with another word – RETURN – which directs action back to the line *after* the one which contained the word GOSUB, as this brief program illustrates:

```
5  CLS:CLS #0
10 PRINT "This is line 10"
15 PAUSE 55
20 GO SUB 50
30 PRINT \"This is line 30"
40 STOP
50 PRINT \"This is the subroutine at
50"
60 PAUSE 55
70 PRINT \"I am now RETurning"
80 PAUSE 55
90 RETurn
```

When you run this, you'll see the following on the QL screen:

```
This is line 10

This is the subroutine at 50

I am now RETurning

This is line 30
```

The program first tells you it is at line 10, then hits the GOSUB 50 in line 20. The section of program it goes to, which lies between the line control was sent to (50 in this case), and the RETurn (90 in our program) is called a *subroutine.*

On reaching the subroutine, the QL tells you this with the THIS IS A SUBROUTINE message. After a pause it tells you I AM NOW RETURNING then after another short pause the program comes to the RETurn statement in line 90, which terminates the subroutine, and sends action to the line *after* the one which sent it to the subroutine, that is, line 30.

The 'destination line number' for a GOTO or GOSUB can be the result of some arithmetic work (such as X=A/B:GOTO X) or of some expression (like GOSUB 8*X). Of course, if you use expressions like this, you cannot renumber your QL program with RENUM as this would send the computer to the wrong place.

The QL has a variation of GOTO and GOSUB which has not been

provided with earlier Sinclair BASICs, the ON . . . GOTO and ON . . . . GOSUB.

Here's a program to show ON . . . GOSUB in action:

```
10 REMark ON...GOSUB Demo
20 CLS:CLS #0
30 REPeat looping
40  x=RND(1 TO 4)
50  ON x GO SUB 80,100,120,140
60  PAUSE 36
70 END REPeat looping
80 PRINT \"That was one"
90 RETurn
100 PRINT \"That was two"
110 RETurn
120 PRINT \"That was three"
130 RETurn
140 PRINT \"That was four"
150 RETurn
```

The QL will print up THAT WAS ONE if the random number generated in line 40 was a 1, THAT WAS TWO if it was a 2, and so on. You find out which of the GOSUB destinations the computer will go to by counting along the line numbers in the ON . . . GOSUB line. The first one is 80, which means the QL will GOSUB 80 if X equals 1. The second line number is 100, so the GOSUB destination when X is 2 is 100 . . . and so on.

ON . . . GOTO works in much the same way, except that – of course – there is no RETURN to send action back to the line after the ON . . . GOTO statement.

Redundant

GOTO, GOSUB, ON . . . GOTO and ON . . . GOSUB are generally not needed on the QL, because of the rich control structures provided. However, we have used them in some programs in this book simply because it makes it a little easier to follow what is going on. Generally, you'll find the more complex programs, do *not* use

GOTO and GOSUB, as I feel that you'll fully understand the 'new' control structures by the time you'll get to them. However, shorter programs, sometimes contain a GOTO or two, simply because – as I said before – it is easier to follow through the action of the program.

The IF Statement

The IF statement lies at the heart of your QL's ability to make decisions. In fact, the principal difference between a simple calculating device and a computer is the latter's ability to make decisions (IF some condition is found to be true, THEN take this branch in the path) and THEN ACT on that decision.

So, the QL (in common with other computers) can test conditions, and use the results of that test to direct the next step the computer will carry out.

There are three versions of the IF statement on the QL. The simplest form, and the one which you are most likely familiar with, is the IF/THEN. IF the milk has been split, THEN cry over it. IF your BBC Micro is broken, THEN buy a ZX8Ø. And so on.

IF . . . test . . . THEN . . . do something . . .

If the result of the test is found to be true, then the QL will carry out the steps following the THEN. The IF and the THEN form a single statement on the QL for this use of IF. Here are some possible program lines containing IF/THEN:

```
IF name$= "tim" THEN PRINT "hello"

IF a+b>c THEN LET d=b

IF guess_5=answer THEN GO SUB 1000
```

In the second way IF can be employed on the QL, the THEN does not have to be within the same statement as the IF. You need to use a third statement, END IF, to use it in this second way, when a number of 'THEN changes' are made.

If you look at the next small program segment, you'll see this in action:

```
10 IF a$="end" THEN
20  PRINT "Thanks for playing"
30  LET high_score=score
40  LET game_count=game_count+1
50 END IF
```

As you can see, all the statements between the initial IF/THEN (line 10) and the END IF (line 50) are evaluated only if the condition being tested in line 10 is found to be true.

The third version of IF/THEN allows for two outcomes. In the second one, you either carried out the steps within the IF/END IF section, or you did not.

Here's IF/END IF with the addition of ELSE, which allows the QL to choose between two alternative actions. If you follow it through you should quickly understand how to apply it:

```
10 IF score>high_score THEN
20   PRINT "New high score"
30  ELSE
40   PRINT "High score stands"
50 END IF
```

You can see from this program that IF the score achieved is greater than the high score, THEN the program prints out the message "New high score". If the condition being tested in line 10 is not true, then the ELSE is triggered, and the QL prints "You didn't beat high score".

The Optional THEN

In contrast to the Spectrum, but in common with certain other computers (such as the BBC Micro and the Acorn Atom), the word THEN is optional. The QL will understand it is meant to be in place if you replace it with a colon in the first form as follows:

```
IF guess=answer:PRINT "That's right"
```

In the longer forms, you can leave it out completely:

```
10 IF name$="Tim"
20   PRINT "That's a good name"
30   LET score=2*score
40 END IF
```

Although you can leave THEN out, I prefer to always include it, because it helps make it clear what is happening within a program.

Nesting

You can nest IF/END IFs as deeply as you like (presuming, of course, you don't use up all the QL's memory in the process).

Here's a program which nests IF/END IFs two deep. As with nested FOR/NEXT loops (when each NEXT relates to the closest preceding FOR), each END IF relates to the closest preceding IF:

```
10 IF guess=answer THEN
20  IF name$="Tim" THEN
30   PRINT "Well done, Tim"
40    ELSE
50   PRINT "You got it right!"
60  END IF
70 END IF
```

Repetition

There are two fundamental ways of controlling the repetition of sections of code on the QL. These use either REPeat/END REPeat or FOR/END FOR in conjunction with NEXT and EXIT.

The structure is as follows for REPeat/END REPeat:

```
REPeat label
 the stuff you want done
 more stuff for the QL to do
END REPeat label
```

For FOR/END FOR we set up the program as follows:

```
FOR label=range_start TO range_end
 the stuff you want done
 more stuff for the QL to do
END FOR label
```

As I said a little earlier, these are used with two other statements in QL BASIC which are NEXT label and EXIT label. When the QL comes across the NEXT in your program, unless the FOR range has been exceeded when it will go to the line which follows the NEXT, it goes to the line which comes after the relevant FOR or REPeat.

If the QL comes across the word EXIT, it will send its attention to the line which follows the END FOR or the END REPeat chosen by the EXIT line. Note that if you're using EXIT in a loop, you need to use END FOR or END REPeat to end it.

The best thing about these constructions is that they allow you to set up complex forms of what are – to some extent – FOR/NEXT, DO/WHILE or REPEAT/UNTIL constructions, but with the ability to leap out of the middle of them without causing a system crash (as happens on some computers after you have left FOR/NEXT loops prematurely a few times) or without committing offences against the structured programming ethic.

This latter condition may seem irrelevant at this stage, but as you develop your programming skills with the QL, you'll find you are more and more conceiving your programs in structured terms. Once your ideas are being formed in this way, you will discover you develop a marked resistance to writing code which works, but which has been stuck together with the electronic equivalents of plasters.

EXIT, along with REPeat and END REPeat, allows you to construct elegant 'endless loops' as follows. Look at this next program fragment:

```
10 REPeat key_press
20  a$=INKEY$
30  IF a$="q" THEN EXIT key_press
40 END REPeat key_press
50 :
60 REMark ...rest of program...
```

The program will cycle tirelessly from line 10 through to line 40 (that is, from the REPeat key_press through to END REPeat key_press) *until* it finds (in line 30) that you are touching the "q" key (that is, that INKEY$ equals "q"). Once it finds this is so, it EXITs, using the balance of line 30.

Here's another one to make REPeat/END REPeat/EXIT clear:

```
10 REPeat game
20  score=score+result_of_shot
30  IF score>hi_score:EXIT game
40 END REPeat game
50 :
60 REMark ...rest of program...
```

SELECTing A Direction

SELect and END SELect are provided within the QL's vocabulary to give the machine another mechanism with which to determine what it should do next. The value of a variable is tested, and the decision is made based on the value found.

In contrast to IF/THEN (where you must either carry out an action or skip over it) and IF/THEN/ELSE (where you can carry out one of two conditions, or bypass them both) SELect/END SELect allows the QL to choose from a large number of alternative responses.

You'll see that you can also add on ON REMAINDER statement which represents a course of action the QL can take if all the tests prove negative (none of the specified variable values which lead to particular actions have been reached).

Here's how a program can look:

```
100 SELect ON level
110  ON level=1000
120   PRINT "You have not done well"
130   bonus_points=-200
140  ON level=2000
150   PRINT "You are now a captain"
```

```
160   bonus_points=780
170  ON level=3000
180   PRINT "You are now commander"
190   bonus_points=1560
200  ON level=REMAINDER
210   PRINT "You are now emperor"
220   bonus_points=3500
230 END SELect
```

You can see how this program will print up "You have not done well" and set the variable bonus_ points to 200 if level_reached is 1000; "You are now a captain" with bonus_points to 780 if level_reached is 2000 and so on. If the level_reached is neither 1000, 2000 or 3000, then the ON REMAINDER section is triggered, with the bonus_ points being set to 3500 and the message "You are now emperor" being printed up.

You do not have to include the full information which you see in lines 110, 140, 170 and 200 in the preceding program. This next version of the program performs in exactly the same way:

```
100 SELect ON level
110 =1000
120   PRINT "You have not done well"
130   bonus_points=-200
140 =2000
150   PRINT "You are now a captain"
160   bonus_points=780
170 =3000
180   PRINT "You are now commander"
190   bonus_points=1560
200 =REMAINDER
210   PRINT "You are now emperor"
220   bonus_points=3500
230 END SELect
```

Replace IF/THEN

You can also use SELect to provide a cleaner statement than that which you could obtain with an IF/THEN. To show what I mean, here are some IF/THEN statements:

```
10 IF result>500 AND result<1000 THEN
 PRINT "Not very good"
20 IF result>999 AND result<2000 THEN
 PRINT "Barely acceptable"
30 IF result>1999 AND result<3000 THE
N PRINT "You're improving"
```

You can use SELect to produce a cleaner, clearer program as follows:

Although this program is no shorter than the IF/THEN version, it certainly makes it very clear which values the variable result should hold to produce particular effects. In contrast to this, the IF/THEN version demands that you actually work them out (that is, you have to say that *<2000* means 'up to, but not equal to, *2000*' and so on).

```
10 SELect ON result
20  ON result=501 TO 999:PRINT "Not v
ery good"
30  ON result=1000 TO 1999:PRINT "Bar
ely acceptable"
40  ON result=2000 TO 2999:PRINT "You
're improving"
50 END SELect
```

A shorter version is as follows:

```
10 SELect ON result
20  =501 TO 999:PRINT "Not very good"
30  =1000 TO 1999:PRINT "Barely accep
table"
40 =2000 TO 2999:PRINT "You're improv
ing"
50 END SELect
```

Chapter Nine
Functions and Procedures

Functions and procedures are vital elements in your QL programming lexicon. Once you have set them up – using DEFine FUNction and DEFine PROCedure – you trigger them in a program simply by printing their names.

A function acts on a number (or variable) to return a value, so when its name is included within a program, this number or variable must immediately follow it. The answer produced by the function is obtained by use of the RETurn statement. (This is *not* the same as the RETurn statement which terminates a subroutine. With functions, you should think of it as the programming element which returns an answer.)

Here is a typical function being defined at the start of a QL program:

```
10 DEFine FuNction rud_half(x,y)
20  LOCal point_five
30  point_five=INT((x+y)/2)
40  RETurn point_five
50 END DEFine
```

If you want to trigger this function in the course of a program's execution, you include a line such as the following:

```
100 :
110 PRINT rud_half(99,12)
120 :
```

As you can see, this allows you to add arithmetic manipulation words to the QL's vocabulary (rather like FORTH) so that, for the rest of the program, you need only refer to the name you have given to it (rud_half) in order to trigger it.

Note the use of the word LOCAL in line 20 of the function definition section. This ensures that the value given to the variable point five will not get confused with use of the variable name at other points in the program. You could have, for example, X, Y and Z as LOCAL variables which would act quite independently of X, Y and Z in the wider program (where they would be 'global' variables). It is suggested that you do not use action within a function or procedure to assign values to non-local variables, as this can lead to hard-to-locate bugs in your programs.

Procedures

Procedures are very similar to functions, except that you can program actions into them which are not only mathematical. Procedures can modify screen colours, produce sounds, scroll the screen, manipulate strings, or whatever. As a general rule, you can assume that if an action can be carried out legally in SuperBASIC, you can trigger it from within a procedure.

Here's a sample program showing how to define a procedure:

```
10 DEFine PROCedure multi_name(name_c
ount)
20  LOCal ego_trip
30  FOR ego_trip=1 TO name_count
40   PRINT "...your name..."
50   PRINT "---------------"
60   PRINT "***************"
70  END FOR ego_trip
80 END DEFine
```

You trigger this simply by including the name of the procedure within your program:

```
100 :
110 multi_name(6)
120 :
```

You can see how the number of times you want your name to be printed (100, in line 110) follows the procedure 'call' within

brackets. You must include this number (or a variable name) if the procedure itself demands a number. If there is no number within the procedure definition (as in the following example) you do not need to include a set of brackets.

```
10 DEFine PROCedure truncate
20  LOCal b$
30  b$=a$(1 TO (LEN(a$)/2))
40  PRINT b$
50 END DEFine truncate
```

This is called as follows:

```
100 :
110 a$="your name"
120 truncate
130 :
```

As you can see, procedures allow you to add words to the QL's programming vocabulary, just as you can with the FORTH language. Remember that, unlike functions, which are restricted to mathematical manipulations, procedures can be asked to do anything the QL can do.

Chapter Ten
Graphics

The QL is provided with a rich store of words in its vocabulary for controlling the screen, windows, borders, colour and the like. The screen vocabulary includes:

AT	BLOCK
BORDER	CLS
CSIZE	CURSOR
FLASH	INK
OVER	PAN
PAPER	PRINT
RECOL	SCROLL
STRIP	UNDER

All of these (with the exception of PRINT) are followed by numbers which tell the QL exactly what you want to do. In most cases, the number can have one, two or three values (such as Ø, 1 or 2 for SCROLL). RECOL can take any one of eight values (zero through to seven).

Colour

You can control three kinds of colour on your QL. The first one is the background colour (referred to in the following simply as 'colour'). The second can be a 'contrast' colour and the third is the colour produced by using the QL's 'stipple' capabilities. Colour can be controlled by single, double or triple arguments.

Colour Control

Here's how the colour is held within the colour byte. The bottom three bits look after the main colour (colour), the next three is the

contrast colour (XOR of the main colour) and the top two bits are set according to the stipple pattern which you have nominated:

STIPPLE		CONTRAST (XOR)				MAIN	
7	6	5	4	3	2	1	0

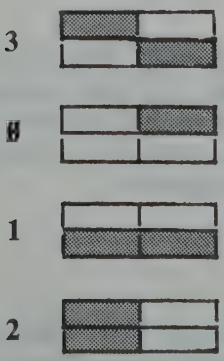
If you want a solid (i.e. non-stipple) display, you simply specify only the top three bits.

Colour control is referred to as single, double or triple. Setting only the top three bits, for solid colour, is a single setting (with just one argument following the relevant statement, such as PAPER 3). If you select the double setting, you are telling the QL that you want a stipple setting (two numbers follow the statement, such as PAPER 1,5). The triple setting allows you to select the main colour, the contrast and the stipple.

Stipples

Stipples are reproduced fairly poorly on domestic TV sets. To see them properly, you need a monitor. However, as you'll discover on your own TV, stipple effects can be used and certainly produce colours which are different from their non-stipple relatives.

There are four stipple patterns, with the first one shown here (numbered as three) being the stipple you'll get if you do not specify a particular pattern.



Modes

As we pointed out earlier, there are two degrees of resolution on the QL. The mode setting (either 256 or 512) determines not only the resolution of the picture, but the number of colours which can be obtained. You'll notice that there is some 'bleeding' between pixels (picture elements) in the higher resolution mode when viewed on a TV. You really need a monitor to see it properly. Note that when you use a MODE statement in a program, all windows and channels feeding to those windows are 'closed'.

MODE 256: In this mode, the screen is 256 pixels wide and 256 deep. In this mode you can use eight colours.

Ø – black
1 – blue
2 – red
3 – magenta
4 – green
5 – cyan
6 – yellow
7 – white

As on the Spectrum, these give various shades of grey on a black and white TV or monitor, ranging from black fairly evenly through to the white.

MODE 512: This mode sets the QL so that the screen is 512 pixels wide, and 256 deep. There are only four colours you can use in this mode:

Ø, 1 – black
2, 3 – red
4, 5 – green
6, 7 – white

Although this mode setting reduces the 'raw' colours which can be produced, the stipples allow you to create an enormous range of colours. Here's a program to demonstrate the range of colours available on the QL (and do a few unusual things to the writing on the screen!):

```
100 REMark Colour demo
110 MODE 256
120 CSIZE 3,1
130 FOR foreground=7 TO 0 STEP -1
140  FOR contrast=0 TO 7
150   FOR stipple=0 TO 3
160 PAPER foreground,contrast,stipple
170 CLS:AT 2,5
180 PRINT "Foreground ";foreground\\"
          Contrast ";contrast\\"
        Stipple ";stipple
190    PAUSE 20
200   END FOR stipple
210  END FOR contrast
220 END FOR foreground
```

As with any program, if you want to stop this before it is over, simply press CTRL and the space bar.

PRINT and Related Words

One of the most commonly-used words in BASIC programming is PRINT. As in other BASICs, SuperBASIC uses PRINT to output information to the screen. However, it also allows you to send information to other locations such as the printer.

The screen is the 'default output device', which means that, if you do not specify a device, PRINT will print on the screen. If, however, you have previously specified a device, such as the printer, PRINT will send its information there, until you override this device instruction with another one.

The simplest use is as follows:

```
PRINT "Hi there, superstar"
```

This prints the inspiring message "Hi there, superstar" directly on the QL screen.

The QL output includes use of 'separators', which allow you to format PRINT output, even if you're not completely sure what the length of individual items will be.

There are four separators:

! – this can be used as an 'intelligent space'. It will put a space between the different items you print out to the screen, or if the next item would be split at the end of a line, will start a new line for it. If the space falls at the start of a line, the ! will suppress the leading space:

```
100 REMark Intelligent separator
110 RESTORE
120 CLS
130 number=4
140 DIM a$(4,60),b$(4,60)
150 FOR items=1 TO number
160  READ a$(items),b$(items)
170  PRINT a$(items)!b$(items)
180 END FOR items
190 DATA "a pig in a poke and a fish"
,"a new way to play"
200 DATA "my electric car","only goes
to Cambridge"
210 DATA "two bits of shine","half pa
st nine"
220 DATA "testing, testing","bombs aw
ay"
```

, – the comma is the usual way of spacing output on the screen, producing neat columns:

```
100 REMark Comma spacing
110 CLS
120 grand_total=20
```

```
130 FOR count=1 TO grand_total
140  PRINT count,
150 END FOR count
```

– the backslash sends the QL to a new line:

```
110 DIM a$(4,20)
120 CLS:RESTORE
130 grand_total=4
140 FOR count=1 TO grand_total
150  READ a$(count)
160  PRINT a$(count)\"       yes"
170 END FOR count
180 DATA "testing a word"
190 DATA "proving a point"
200 DATA "eating your supper"
210 DATA "slaying a dragon"
```

; – finally the semicolon ensures that output follows immediately after the preceding output:

```
100 REMark Semi-colon
110 CLS
120 FOR numbers=33 TO 191
130  PRINT numbers;"-";CHR$(numbers);
"....";
140 END FOR numbers
```

Running these programs will show you the effects that the various separators create.

INK and PAPER

As on the Spectrum, INK is the colour in which the QL prints, and PAPER is the background upon which it is printed. You need to use CLS (clear the screen) after a PAPER command in order to set all the screen to that background colour. The format is simple. You just follow the word with the number which refers to the colour you want:

```
PAPER 3:CLS
```

This will set the background colour to magenta in MODE 256 and to red in MODE 512.

```
INK 1
```

This will set the INK colour to blue in MODE 256 and to black in MODE 512.

A good colour combination, and the one I use by choice, is:

```
PAPER 1:INK 7:CLS
```

This gives you white writing on a blue background.

Screen Clearing

I mentioned above the CLS needs to be used to set the new PAPER colour all over the screen. If CLS does not immediately follow a PAPER command, then CLS simply clears the screen of whatever happens to be on it.

You can follow the CLS with a number (from 1 to 4) which modifies this 'global' clearance as follows:

1 – this clears all of the top of the screen, except for the cursor line

2 – much the same as 1, except that it clears the bottom of the screen, but still leaves the cursor line

3 – this allows you to just clear all of the cursor line

4 – by following CLS with 4 you can clear only the right hand end of the cursor line, starting with and including the character cell the cursor is currently occupying

You have to follow the CLS with a hash symbol (#) before the number to get these to work. If you use a zero instead of a number, the command will clear the 'working area' below the screen windows.

BORDER

The BORDER command on the QL is much more flexible than its counterpart on the Spectrum. The BORDER is the area, fairly obviously, which surrounds the PAPER area. You can control the width and the colour of the BORDER, and can have a number of BORDERs, of different widths and colours, all inside each other.

This next program produces a wild, flashing BORDER, which works its way towards the centre of the screen, leaving a fairly small PAPER area in the middle:

```
100 FOR W=85 TO 2 STEP -2
110  BORDER W,RND(0 TO 7)
120  PAUSE 5
130 END FOR W
```

Lively Graphics

Of course, you can do far more with screen output than just change INK, PAPER and BORDER colours. Let's have a look at a few of the additional graphic possibilities on the QL.

FLASH: The PRINT output can be 'steady' (the normal state) or 'flashing', in which the QL swaps the PAPER and INK colours over and over again. This program shows FLASH in action:

```
100 REMark FLASH demo
110 PAPER 0:INK 6
120 CLS::CLS #0
130 PRINT \\"  I'm not flashing"
140  PAUSE 50
150 FLASH 1
160 PRINT \\\"  But I am!"
170  PAUSE 70
180 FLASH 0
190 PRINT \\\"  However, I am not"
```

As you can see, FLASH 1 turns the flash on, and FLASH 0 turns it off. You can only use FLASH when

the QL is in the lowest resolution mode (MODE 256 or MODE 8).

RECOL: This command changes the colour of whatever is on the screen, without wiping any of the information which appears on it. It works by changing individual dots on the screen following your wishes. That is, you can use RECOL to turn every green pixel red, or white, or whatever you choose. RECOL is followed by *eight* numbers (preceded, possibly, by a channel number).

The first number after RECOL specifies the new colour for black, with the following numbers specifying the new colours for blue, red, magenta, green, cyan, yellow and white.

Here's a program to show it in action:

```
100 REMark RECOL demonstration
110 PAPER 0:CLS:CLS #0
120 FOR J=1 TO 740
130  INK RND(0 TO 7)
140  PRINT CHR$(255);
150 END FOR J
160    PAUSE 40
170 BEEP 32000,25
180 RECOL 3,4,5,6,7,0,1,2
190    PAUSE 40
200 BEEP 32000,50
210 RECOL 4,5,6,7,0,1,2,3
220    PAUSE 40
230 BEEP 32000,100
240 FOR J=1 TO 5
250  RECOL RND(0 TO 7),RND(0 TO 7),RN
D(0 TO 7),RND(0 TO 7),RND(0 TO 7),RND
(0 TO 7),RND(0 TO 7),RND(0 TO 7)
260  BEEP 3000,RND(1 TO 10)
270 END FOR J
```

OVER: OVER is followed by –1, 0 or 1. The number determines the way in which the QL will 'overprint', as you'll see

from the following programs. Run the first one, which uses OVER -1 in both lines 160 and 200:

```
100 REMark OVER -1 demo
110 PAPER 0:INK 7
120 CLS:CLS #0
130 AT 3,0
140 PRINT "Let's get this show on the
 road"
150 PAUSE 40
160 OVER -1
170 AT 3,0
180 PRINT "XLet's get this show on th
e road"
190 PAUSE 40
200 OVER -1
210 AT 3,0
220 PRINT "XLet's get this show on th
e road"
```

Note, note the difference when the -1 in line 200 is changed to a 1, as follows:

```
200 OVER 1
```

Once you've done that, change lines 160 and 200 as follows:

```
160 OVER 0

200 OVER -1
```

UNDER: This has nothing to do with being an 'inverse OVER', but refers to the ability of the QL to *underline* its output on demand using the current INK colour:

```
10 REMark UNDER demo
20 INK 7:PAPER 1:CLS
30 PRINT \"This is standard"
40 UNDER 1
50 PRINT \\"This is underscored"
60 UNDER 0
70 PRINT \\"And this is not"
```

Like FLASH, UNDER can either be followed by 1 (on) or 0 (off).

PAN and SCROLL

You can use these to produce quite startling effects. In contrast to ZX81 SCROLL, which moved a whole line up the screen, SCROLL on the QL moves in pixels, so some very smooth scrolling is possible. PAN, which can be thought of as a 'sideways scroll', moves screen contents either right or left the number of pixels you define.

PAN: If you simply follow the command PAN with a positive number, it will pan the whole of the window you're using at that time the number of pixels specified by the number to the left.

This means that PAN 24 will move the current printed material 24 pixels to the left, with some extra PAPER coming in from the right to fill. An instruction to PAN –65 will move the current screen contents 65 pixels to the right, again with PAPER coming in from the left to fill.

You can follow PAN with two numbers. The first, as above, controls the pixel movement (with the sign of that number controlling the direction). The second number can be either 3 or 4. If it is 3 (as in PAN –25, 3), the QL will PAN just the current cursor line (in this case 25 pixels to the right). If the second number is 4 (as in PAN –25, 4) your computer will PAN the cursor line from (and including) the cursor position.

SCROLL: This moves the contents of the window up or down. In a similar manner to PAN, you can specify the direction and number of pixels over which the screen will SCROLL. As well, PAPER comes in to replace the empty window area.

If SCROLL is followed by a single number, it will move the contents of the window up (if the number is positive, as in SCROLL 19 to move it up to 19 pixels) or down (if the number is negative).

If you want only part of the screen to SCROLL, you add a second number after the first. Adding a 1 (so the line reads SCROLL 12,1) tells the QL to SCROLL just the

top of the window, and a 2 (so it reads SCROLL 12,2) lets the computer know you want to SCROLL just the bottom of the window. In both cases, the cursor line is not included in the SCROLLed material.

This next demonstration program shows both PAN and SCROLL in use. Note that if part of the information on the screen is either PANned or SCROLLed off, it is not restored when the rest of the line moves away from the edge. You'll see this in the following program when the asterisk in the second row is lost in a PAN, and the bottom half of the second row of letters vanishes due to SCROLL:

```
100 REMark PAN demo
110 PAPER 6:INK 2
120 CSIZE 3,1
130 CLS:CLS #0
140 AT 3,3
150 PRINT "Here we go!     *"
160 PRINT "       Here we go!     *"
170 FOR j=1 TO 3
180  PAUSE 20
190  PAN -40
200  PAUSE 20
210  PAN 80
220 END FOR j
230 PAUSE 20
240 PAN -80
250 FOR k=1 TO 3
260  FOR j=1 TO 9
270   SCROLL j
280  END FOR j
290  FOR j=1 TO 5
300   SCROLL -j
310  END FOR j
320 END FOR k
```

Sizing Things Up

QL SuperBASIC comes with the marvellous statement CSIZE (which stands for character size). This allows you to actually change

the size that each character – within certain limits – will appear on the screen.

There are four width sizes, and two height sizes. You can see them in action with this program:

```
100 REMark CSIZE demo
110 PAPER 0:INK 6:CLS:CLS #0
120 REPeat cycle
130 CSIZE 0,0
140 REPeat loop
150  INPUT \\"Enter height (0 or 1)";
h
160  IF h=1 OR h=0 THEN EXIT loop
170 END REPeat loop
180 REPeat loop_two
190  INPUT "Now width (0 to 3)";w
200  IF w>=0 AND w<=3 THEN EXIT loop_
two
210 END REPeat loop_two
220 CSIZE w,h
230 PRINT \"This is how it looks with
"
240 PRINT " height ";h;", and ";w;" w
idth"
250 FOR j=1 TO 1000:END FOR j
260 END REPeat cycle
```

As you can see from this program, the width can be between 0 and 3, and the height either 0 or 1. The numbers control the character width as follows:

WIDTH CODE	SIZE pixels	MODE 256	MODE 512
0	6	42	85
1	8	32	64
2	12	21	42
3	16	16	32

And this is how the height is controlled:

HEIGHT CODE	SIZE pixels	ROWS
0	10	25
1	20	12

Terrapin Tricks

Our discussion of QL graphics must include a demonstration of turtle graphics. A discussion, and some extensions of the graphics, comes a little later in the book in our material on *QLogo*. For now, just enter and run the following program. It is very similar to the one given close to the beginning of the book, so if you saved that one on microdrive, I suggest you load it back in now and enjoy it again. If you didn't save it, here's the listing again:

```
100 REMark Turtle graphics
110 PAPER 0
120 CLS:CLS #0
130 PENDOWN
140 REPeat logo
150  IF RND>.68 THEN CLS
160  note=RND
170  POINT 80,50
180  INK RND(1 TO 7)
190  turtle=RND(3 TO 70)
200  terrapin=RND(25 TO 340)
210   FOR k=1 TO turtle
220    MOVE k/.7
230    TURN terrapin
240    BEEP 3200,k/note
250   END FOR k
260 END REPeat logo
```

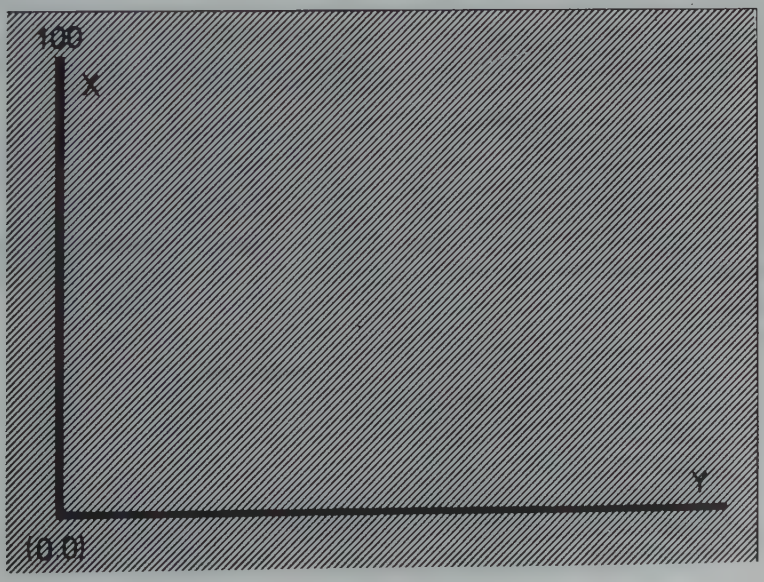
Chapter Eleven
Coordinate Systems

The Graphics System

The 'graphics coordinate system' puts the origin (Ø,Ø) in the bottom left hand corner of the scale, and determines that the height of the screen will be 1ØØ units.

The window shape determines how far along the x scale you can go. However, the scale of units is one to one with those on the y axis.

Five of the graphics procedures are MOVE (to move the graphics cursor), POINT (to plot a point), ARC (which draws an arc of a circle), LINE (to draw a line) and ELLIPSE (which is used to produce an ellipse).



Although, as we've said, a scale of 100 units is the standard when working on the graphics coordinate system, you can modify this to suit your own needs, as we will see shortly.

The POINT command allows you to place a dot on the screen in a position which is specified in relation to the graphics origin (bottom left hand corner). It is generally followed by two numbers, the first being the x coordinate, and the second the y coordinate.

Try the following program, entering numbers of your choice between 0 and 99. Terminate the program by entering -99 when asked for the first co-ordinate:

```
100 REMark POINT demo
110 INK 6:PAPER 0
120 CLS:CLS #0
130 REPeat loop
140  AT 0,0
150  INPUT "first co-ordinate? ";x
160  IF x=-99 THEN EXIT loop
170  INPUT "second co-ordinate? ";y
180  AT 0,0
190  PRINT x;" ";y;"
                    "
200  POINT x,y
210  FOR j=1 TO 100:END FOR j
220 END REPeat loop
230 STOP
```

Now you've done some of the work of entering co-ordinates for POINT, you can let the QL do it. In this program the screen is filled, within the red border, by a continuously evolving symmetrical design:

```
100 REMark POINTs abounding
110 PAPER 0:MODE 8
120 CLS:CLS #0
130 BORDER 2,2
140 REPeat delay
150  IF INKEY$="" THEN EXIT delay
160 END REPeat delay
```

```
170 REPeat design
180  IF INKEY$<>"" THEN EXIT design
190   IF RND>.5 THEN
200    INK 7
210     ELSE
220    INK 0
230   END IF
240  x=RND(66 TO 166)
250  y=RND(100)
260  POINT x,y
270  POINT 166-x,y
280  POINT 166-x,100-y
290  POINT x,100-y
300 END REPeat design
```

You can stop the program at any time by holding down a key.

LINE

You can use LINE to draw a straight line (surprise!) or one which spins through a designated angle starting at the current cursor position. You follow LINE with two numbers which specify the end point of the line. If you add a third number, the line will turn through this number of degrees as it is drawn. You can also draw a number of lines by using TO with the coordinates (as LINE 0,0 TO 10,10).

Our first LINE program generates lines which start at the origin, and terminate at a randomly-determined point. The program draws in the line, and tells you the co-ordinates of the end point:

```
100 REMark Line drawing
110 MODE 8:SCALE 100,0,0
120 PAPER 6:INK 2:CLS:CLS #0
130 REPeat lines
140  a=RND(10 TO 100)
150  b=RND(10 TO 100)
160  AT 0,5
170  PRINT "LINE 0,0 TO ";a;",";b
180  LINE 0,0 TO a,b
190  PAUSE 40
```

```
200  CLS
210  IF INKEY$<>"" THEN EXIT lines
220 END REPeat lines
```

The next program allows you to enter the starting co-ordinates (in line 150), pressing the ENTER key between each number. You then do the same for line 180. The INPUT lines vanish from the screen, and a line, as specified by you will appear on the screen. You'll also see the co-ordinates you've chosen printed at the top of the screen.

```
100 REMark LINE demo
110 PAPER 0:INK 6
120 CLS:CLS #0
130 REPeat loop
140  AT 0,0
150  INPUT "First co-ordinates ";x,y
160  INPUT "Second co-ordinates ";a,b
170  AT 0,0
180  PRINT x;",";y;" to ";a;",";b;"
                                    "
190  LINE x,y TO a,b
200  FOR j=1 TO 2000:END FOR j
210 END REPeat loop
```

Now we'll produce a LINE version of our last POINTs program. This program draws random rectangles on the screen, extremely quickly, in random colours. If you look closely at the listing, you'll see that, in effect, it works by 'joining the points' created in the *POINTs abounding* program:

```
100 REMark LINEs abounding
110 PAPER 0:MODE 8
120 CLS:CLS #0
130 REPeat delay
140  IF INKEY$="" THEN EXIT delay
150 END REPeat delay
160 REPeat design
170  IF INKEY$<>"" THEN EXIT design
180 INK RND(1 TO 7)
190  x=RND(66 TO 166)
200  y=RND(100)
```

```
210  LINE x,y TO 166-x,y
220  LINE 166-x,y TO 166-x,100-y
230  LINE 166-x,100-y TO x,100-y
240  LINE x,100-y TO x,y
250 END REPeat design
```

Finally, in our investigation of LINE, we'll concentrate on a program which I've called *Star Attack.* This produces a remarkably effective 'coloured broken glass' effect, which is constantly changing You may well wish to experiment with this program, to write a routine to add synchronised sound to the visual effect.

```
100 REMark Star attack
110 PAPER 0
120 CLS:CLS #0
130 a=RND(1 TO 70):b=a+30
140 REPeat star
150  INK RND(1 TO 7)
160  IF RND>.7 THEN a=RND(1 TO 70):b=
a+30
170  x=65+RND(a TO b):y=RND(a TO b)
180  FOR j=1 TO RND(10 TO 20)
190   LINE x/2,y/2 TO RND(x+15),RND(y
+15)
200  END FOR j
210  IF RND>.6 THEN CLS
220 END REPeat star
```

CIRCLE and FILL

Form LINE we move to some other great effects possible on the QL. The CIRCLE command is followed by three numbers. The first two specify the position of the centre of the circle (across followed by up) and the third number controls the radius. FILL, like FLASH, is followed by either a 1 or a 0, with 0 meaning 'off' and 1 meaning 'on'. Run the *Pulsing Bullet Holes* program which follows, and you'll soon understand how effective CIRCLE and FILL can be when combined in this way:

```
100 REMark Pulsing Bullet-holes
110 PAPER 0
120 CLS:CLS #0
130 a=RND(12 TO 74):b=RND(12 TO 88):c
=RND(3 TO 12)
140 REPeat spheres
150  c=c+RND(-2 TO 2)
160  IF RND>.76 THEN a=RND(12 TO 134)
:b=RND(12 TO 88):c=RND(1 TO 15)
170  INK RND(1 TO 7)
180  FILL RND(0 TO 1)
190  CIRCLE a,b,c
200  IF RND>.98 THEN CLS
210 END REPeat spheres
```

Our next program, *Orbits,* is based on *Pulsing Bullet Holes* with several changes. The most significant one occurs in line 190 where – as you can see – there are two additional numbers after the three which followed CIRCLE in the preceding program. The final two numbers control what is called the 'eccentricity' of the shape drawn. A circle has an eccentricity of 1 so, in effect, the command CIRCLE and the alternative command ELLIPSE are identical in their effect in SuperBASIC.

```
100 REMark Orbits
110 PAPER 0
120 CLS:CLS #0
130 a=RND(12 TO 44):b=RND(12 TO 108):
c=RND(1 TO 8):d=RND(2 TO 4):e=RND(2 T
O 4)
140 REPeat cycles
150  c=c+RND(-2 TO 2)
160 IF RND>.76 THEN a=RND(12 TO 144):
b=RND(12 TO 88):c=RND(1 TO 8):d=RND(2
 TO 4):e=RND(2 TO 4)
170  INK RND(1 TO 7)
180  FILL RND(0 TO 1):IF RND>.4 THEN
FILL 0
190  CIRCLE a,b,c,d,e
200  IF RND>.98 THEN CLS
210 END REPeat cycles
```

ARC

From CIRCLE and FILL we move to ARC, which is a curved line forming part of a circle. As LINE draws a straight line from x,y TO a,b so ARC draws a semicircle from point x,y TO a,b moving in an anticlockwise direction.

Our demonstration program shows ARCs being drawn in overlapping patterns, with much changing of colour. The values of the start and end points (a, b, c and d) are shown at the bottom of the screen. The screen clears every so often. The overall effect is very striking indeed, as you'll see when you run it.

```
100 REMark ARC in action
110 PAPER 0
120 CLS:CLS #0
130 REPeat oval
140  INK 6
150  a=RND(3 TO 145):b=RND(1 TO 145)
160  c=RND(1 TO 45):d=RND(1 TO 130)
170  AT 19,5
180  PRINT "a ";a;"   b ";b;"   c ";c
;"   d ";d;"    "
190  FOR j=1 TO 2 STEP .15
200   INK RND(1 TO 7)
210   ARC a,b TO c,d+j,1
220   ARC c,d+j TO a,b, 1
230  END FOR j
240  IF RND>.84 THEN CLS
250 END REPeat oval
```

Once you've run *ARC in action* in its present form, modify it by adding 115 as below:

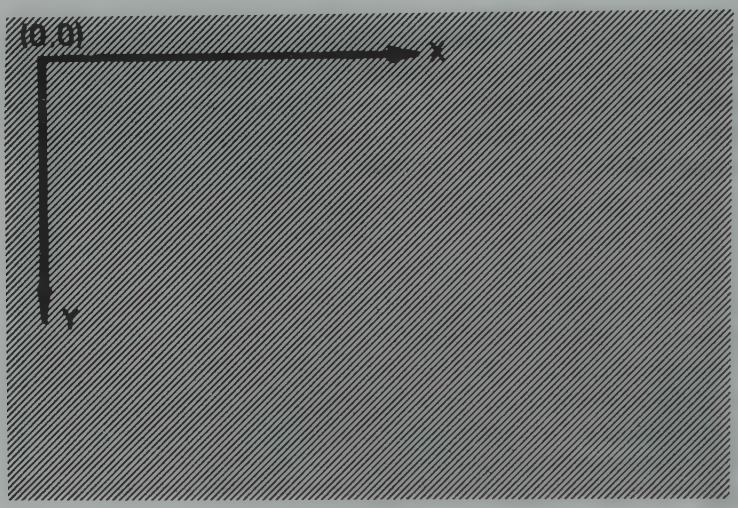
```
115 FILL 1
```

You'll see from this just how quickly FILL works. Now try deleting 115 and adding the following:

```
135  FILL RND(0 TO 1)
```

The Pixel System

Whereas the graphics coordinate system which we've been using starts in the bottom left hand corner, the pixel coordinate system starts in the top right hand corner of the screen, and is used to specify such things as the positions and sizes of windows.



This system, instead of using an arbitrary unit which is one-hundredth the height of the screen, refers to the positions on the screen in pixels, assuming always that the QL is in MODE 512. It doesn't matter, however, if you are in MODE 256, as the QL automatically accommodates the mode that you are working in.

Certain commands, such as WINDOW, are specified relative to the default window origin, while others, such as BLOCK, are specified relative to the current window origin.

BLOCK

This is the first command we will look at which uses the pixel system. The command fills a rectangle on the screen in the size and shape you determine. BLOCK is followed by five numbers (plus an optional one to determine the channel in which it will appear). The first two

numbers control the width of the block in pixels, the next two determine its height, and final argument controls the colour of the block.

BLOCK produces some extremely rapid effects, as our next program demonstrates:

```
100 REMark Close encounters of the
                  BLOCK kind
110 PAPER 0
120 CLS #0
130 REPeat demo
140  CLS
150  z=RND(1 TO 4)
160   SELect ON z
170    =1
180     one
190    =2
200     two
210    =3
220     three
230    =4
240     four
250   END SELect
260  PAUSE 20
270 END REPeat demo
280 :
290 DEFine PROCedure one
300  BLOCK 100,75,90,12,2
310  BEEP 11000,75
320 END DEFine one
330 :
340 DEFine PROCedure two
350  BLOCK 100,75,240,12,3
360  BEEP 4000,103
370 END DEFine two
380 :
390 DEFine PROCedure three
400  BLOCK 100,75,90,115,4
410  BEEP 1000,124
420 END DEFine three
```

```
430 :
440 DEFine PROCedure four
450  BLOCK 100,75,240,115,1
460  BEEP 1900,162
470 END DEFine four
```

Once you've run it for a while in this form, delete line 260 to see how frantically and quickly BLOCK can work.

SCALE

The pixel system allows you to determine the scale of the screen. If you do not specify any scale, the screen will be 100 units in height, and the origin will be in the bottom left hand corner. However, both the vertical height of the screen and the origin can be controlled. The command you need is scale. The default parameters of this command are 100,0,0 where 100 determines the number of units tall the screen will be, and the 0,0 places the origin.

Our *Scaling Demonstration,* which follows, draws a 'windmill' with the centre of the blades at the origin. The program starts with SCALE set to 100,0,0 (using the values scal, a and b set in line 140). By putting the centre of the windmill's blades at the origin, you can see exactly where the origin is each time the scale changes. When you run this program, you'll see the origin gradually move across and up the screen. The windmill gets smaller each time it is drawn, as the first value following SCALE increases. The values of scal, a and b are shown near the top left hand corner of the screen:

```
100 REMark Scaling Demonstration
110 PAPER 1:INK 7
120 PENDOWN
130 CLS:CLS #0
140 scal=100:a=0:b=0
150 FOR demo=1 TO 40
160  SCALE scal,a,b
170  CLS
180  AT 1,1:INK 7
190  PRINT "Scale ";scal;"  Origin ";
a,b
```

```
200  POINT 0,0
210   FOR j=1 TO 9
220   TURN 40
230   INK RND(4 TO 7)
240    FOR k=1 TO 3
250     MOVE 35
260     TURN 120
270    END FOR k
280   END FOR j
290 scal=scal+10:a=a-10:b=b-10
300 BEEP 10000,5*demo:PAUSE 25
310 END FOR demo
```

Chapter Twelve
User-Defined Graphics

In this chapter we look at a short program in SuperBASIC to create user-defined graphics on the QL. It was written by Derek Wilson, and first appeared in *Quanta*.

First of all, here is the program:

```
100 REMark User-defined graphics
110 REMark       Derek V Wilson
120 :
130 RESTORE
140 a=RESPR(64)
150 :
160 FOR i=0 TO 46 STEP 2
170  READ x
180  POKE_W a+i,x
190 END FOR i
200 :
210 DEFine FuNction udg(chan,start,nu
m)
220 LOCal space
230 space=RESPR(2 + 9*num)
240 CALL a,chan,space,start,num
250 RETurn space+2
260 END DEFine
270 :
280 DATA 8302,48,-15620,40,-11839
290 DATA -19986,52,27676,8246,-30720
300 DATA 27926,8256,28672,-27703,9282
310 DATA 5251,21316,5444,1,28709
320 DATA 20035,20085,28922,20085
330 :
340 chars=udg(1,128,16)
```

```
350 :
360 FOR i=0 TO 8
370  POKE(chars + i),124
380 END FOR i
390 :
400 CLS
410 PRINT CHR$(128)
```

The DATA statements in lines 280 onwards contain machine code which is POKEd into memory at the location reserved by the RESPR statement in line 140. This sets up the required character set, using parameters passed to it defining the code of the first character required, the number of characters for which space has been reserved, the address of that space, and the channel for which these characters are being defined. As well, you'll see the program provides a function which returns the start address at which bit patterns may be POKEd, given a simplified version of the data.

The program is used simply by entering it, then defining a character set by a function call such as you can see in line 340. This reserves space for sixteen characters.

Characters are defined as a bit pattern, which *must* occupy *only* bits 6 to 2 inclusive of each byte, bits being number 0 to 7, with bit 7 the most significant and bit 6 being the leftmost element. Each character requires nine consecutive bytes to define it, with the first byte corresponding to the top of the character. Therefore, the routine from lines 360 onwards will set up character 128 as a solid block, similar to the cursor. CSIZE and all other character control statements work in exactly the same way with user-defined characters as they do with characters from the original set.

Here are some other characters you might like to try. Change the lines from 350 so they read as follows, then simply put the data for your character in line 420.

First of all, this turns the solid block into a 7:

```
350 :
360 FOR i=0 TO 8
365  READ z
370  POKE(chars + i),z
```

```
380 END FOR i
390 :
400 CLS
410 PRINT \\"   ";CHR$(128);"    ";CHR$
(128):STOP
420 DATA 0,124,4,8,16,32,64,0,0
```

This is a right bracket:

```
420 DATA 0,48,8,8,8,48,0,0,0
```

A sort of space invader thing:

```
420 DATA 0,124,124,108,40,108,0,0,0
```

A spinning top:

```
420 DATA 0,16,56,68,56,16,0,0,0
```

A little tower:

```
420 DATA 0,24,60,36,60,36,60,0,0
```

And finally, a bomb or 'falling man' (depending on how you look at it):

```
420 DATA 0,108,40,16,124,56,16,0,0
```

You can have a lot of fun experimenting with graphics on the QL using this routine.

Chapter Thirteen
QLogo

Logo is a computer language which is very different from BASIC. BASIC was created as a language which was extremely easy to program with, even though it was not a language which could easily be extended. Logo was also written to be easy to use, but it was created so that it could be easily extended. As well, it was designed to be a language which would 'teach learning'. Proponents of the language say that, to some extent, this aim has been realised.

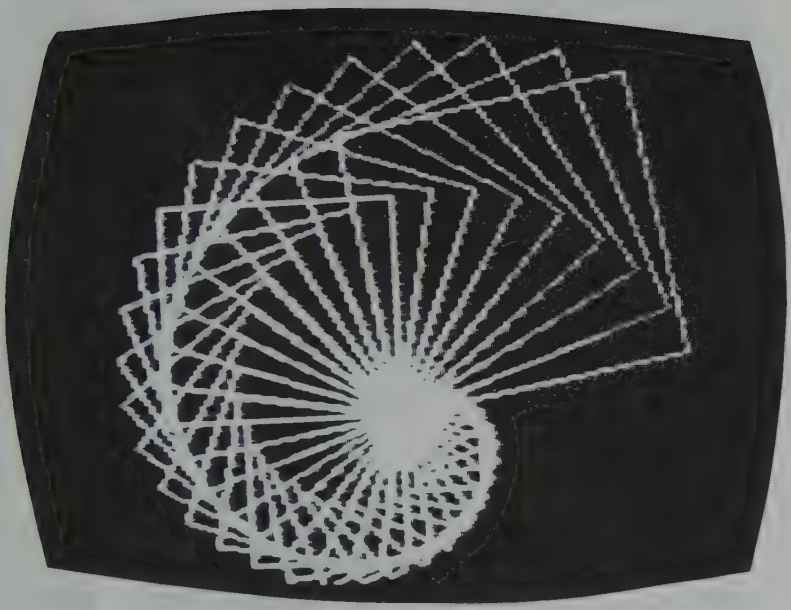
The *turtle graphics* provided on the QL can be combined to produce many of the commands which make up the standard logo language. In this chapter, we'll work on this idea, producing a language which I've called *QLogo.*



Logo proper was pioneered by Dr Simon Papert when he was a Professor of Mathematics and Education at the Massachusetts Institute of Technology in America. He moved from there to France to head the World Computer Centre. Papert said he believed that the first computer language a person learns colours the way the individual both programmed, and thought about programming, for the rest of his or her life. He said that a start in Logo paved the way for much better programming than a start in BASIC. Of course, at this stage BASICs of the quality of SuperBASIC (or BBC BASIC, which also contains many fine control structures) had not been developed. Otherwise, the story may have been different.

Anyway, Papert said he'd learned from observations made by Jean Piaget that children were able to acquire quite complex skills – such as being able to talk and walk – without formal training. As well, the informal environment which encouraged such learning was markedly absent from normal classrooms. Papert set out to create a computer language which would go some of the way towards remedying this deficiency. According to Papert, most instruction in computer programming puts the learner almost in the position of being programmed by the computer. Logo, by contrast, puts the child firmly in charge.





It does this by allowing the programmer to create new shapes and actions – such as one which draws a triangle – and gets the computer to execute this on demand simply by entering the command TRIANGLE. Ordinary BASIC has no such way of creating new commands and functions. They can be emulated, within a listing, in SuperBASIC by creating *procedures* which use the turtle graphics commands to create new shapes.

The commands available in SuperBASIC for turtle graphics are PENUP, PENDOWN, MOVE, TURN and TURN TO. These can be combined with other SuperBASIC options to emulate many 'true Logo' commands, such as HOME, CLEARSCREEN, SETPOS, SUM and REMAIN.

Enter the following program to give your QL a set of fifteen *QLogo* commands. Once you have the program in place, return to the book and I'll explain the commands so you can start combining them into your own *QLogo* programs.

```
100 REMark QLogo
110 initialise
120 :
130 REMark ** Program goes here **
140 :
150 :
160 :
170 :
180 :
190 :
200 :
210 :
220 :
230 :
240 :
250 :
260 :
270 :
280 :
290 :
300 :
310 :
320 :
330 :
340 :
350 :
360 :
370 DEFine PROCedure HOME
380  POINT 80,50
390  setheading 0
400 END DEFine HOME
410 :
420 DEFine PROCedure clearscreen
430  HOME
440  CLS
450 END DEFine clearscreen
460 :
470 DEFine PROCedure forward(x)
480  MOVE x
490 END DEFine forward
500 :
```

```
510 DEFine PROCedure left(x)
520  TURN(x)
530 END DEFine left
540 :
550 DEFine PROCedure right(x)
560  TURN (360-x)
570 END DEFine right
580 :
590 DEFine PROCedure backK(x)
600  TURN 180
610  MOVE(x)
620 END DEFine backK
630 :
640 DEFine PROCedure pencolour(x)
650  INK x
660  pencol=x
670 END DEFine pencolour
680 :
690 DEFine PROCedure penerase
700  INK backgrnd
710 END DEFine penerase
720 :
730 DEFine PROCedure background(x)
740  PAPER x
750  backgrnd=x
760  clearscreen
770 END DEFine background
780 :
790 DEFine PROCedure setheading(x)
800  TURNTO 450-x
810 END DEFine setheading
820 :
830 DEFine PROCedure setpos(x,y)
840  POINT x,y
850 END DEFine setpos
860 :
870 DEFine PROCedure sum (x,y)
880  result=INT(x+y)
890  PRINT result
900 END DEFine sum
910 :
```

```
920 DEFine PROCedure random(x)
930  result=RND(0 TO x-1)
940 END DEFine random
950 :
960 DEFine PROCedure product (x,y)
970  result=INT(x*y)
980  PRINT result
990 END DEFine product
1000 :
1010 DEFine PROCedure remain(x,y)
1020  result= x MOD y
1030  PRINT result
1040 END DEFine remain
1050 :
1060 DEFine PROCedure initialise
1070  backgrnd=0:pencol=6
1080  PAPER backgrnd:INK pencol
1090  MODE 8:CLS:CLS #0
1100  HOME
1110 END DEFine initialise
```



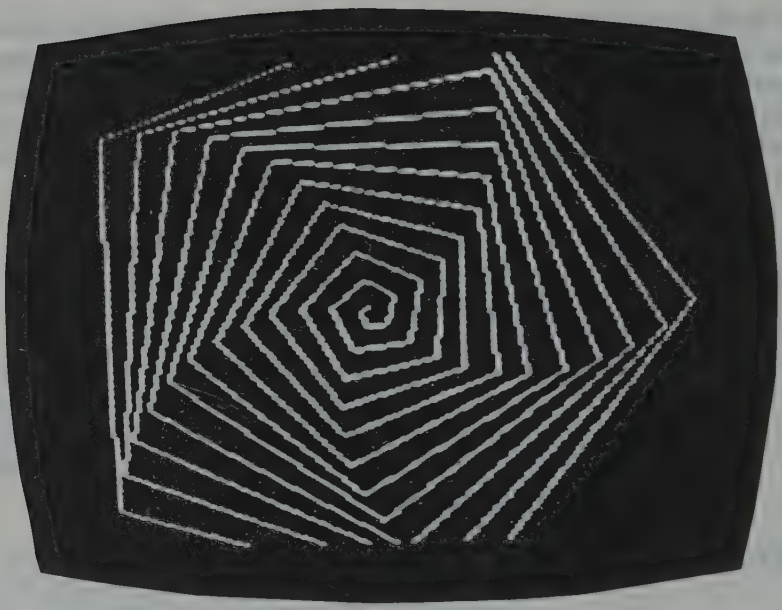
As well as the fifteen commands added by *QLogo,* you also use the words PENUP and PENDOWN, as these are the usual commands used in standard Logo vocabularies. (If you're used to running a version of Logo which has slightly different names for some things, such as RML Logo, by all means substitute the names you know in the procedure definition lines in the program.)

FORWARD X – this moves the invisible turtle (which is imagined to be drawing a line behind it as it moves) X steps forward

HOME – this puts the turtle in the middle of the screen, facing upwards (RML Logo uses the command CENTRE, which puts the turtle in the centre of the screen, facing right, so the CENTRE command is equal to HOME:RIGHT 90)

CLEARSCREEN – this clears the screen, and performs a HOME as well (Tandy Logo uses the command CLEAR and in RML Logo you use CLEAN to get the drawing off the screen, WIPE to clear text from the screen, and CENTRE to get the turtle back in the middle of the screen. As with HOME, a RIGHT 90 is needed to ensure those familiar with RML Logo can use *QLogo* easily)





LEFT X – turns the turtle through X degrees to the left

RIGHT X – turns the turtle through X degrees to the right

BACK X – this allows the turtle to move in reverse for X steps (RML Logo calls this BACKWARD or BK)

PENCOLOUR X – selects a colour in which the turtle will draw, choosing from the standard QL MODE 8 palette (in RML Logo this command is PENCIL)

PENERASE – by setting this command, you get the turtle to 'draw in the backgrond colour', so it can go back over its tracks, rubbing out a previously-drawn line (Tandy Logo has no equivalent of this command, while RML Logo calls it ERASER)

BACKGROUND X – whereas PENCOLOUR selects an INK colour, BACKGROUND chooses a PAPER colour

(again from the MODE 8 palette) and performs a CLEARSCREEN (this command is not available in RML)

SETHEADING X – turns the turtle so it is facing the direction specified (with zero degrees being straight up the screen, and 90 being to the right)

SETPOS X,Y – moves the turtle to the position specified by the co-ordinates X and Y, putting a dot at that position. (Tandy Logo draws a line from the initial position to the new one; Acornsoft's Logo calls this command SETXY, while in RML Logo it is SETX, SETY)

SUM X,Y – this adds the numbers X and Y together, printing out the result of the addition (RML does not provide this)

RANDOM X – produces a random integer between zero and X-1 (the RML command does not produce integers)



PRODUCT X,Y – multiplies X and Y together (not provided in RML Logo)

REMAIN X,Y – gives the result of an X MOD Y operation

You use *QLogo* by entering your program in the blank lines from 14Ø through to 36Ø (and you can easily renumber the whole thing in larger steps if you want more room in this space). You can call on the defined procedures to create your own commands. As well, or instead, you can add to the *QLogo* provided, to give 'prepackaged' routines, which can also be called on. Here are some such prepackaged routines which go at the very end of the *QLogo* listing, to give you SQUARE, POLYGON, TRIANGLE, STAIRS and SPINNER:



```
1120 :
1130 REMark 'Prepackaged' routines
1140 :
1150 DEFine PROCedure square(x)
1160  FOR j=1 TO 4
1170   forward x
1180   right 90
1190  END FOR j
```

```
1200 END DEFine square
1210 :
1220 DEFine PROCedure polygon(w,x,y,z
)
1230 random 4
1240 pencolour result+4
1250  FOR k=1 TO w
1260   FOR j=1 TO x
1270    forward y
1280    right z
1290   END FOR j
1300  END FOR k
1310 END DEFine polygon
1320 :
1330 DEFine PROCedure triangle (x)
1340  FOR j=1 TO 3
1350   forward x
1360   right 120
1370  END FOR j
1380 END DEFine triangle
1390 :
1400 DEFine PROCedure stairs(x,y)
1410  FOR j=1 TO x
1420   forward y
1430   right 90
1440   forward y
1450   left 90
1460  END FOR j
1470 END DEFine stairs
1480 :
1490 DEFine PROCedure spinner(x,y,z)
1500  POINT x,y
1510  FOR j=1 TO 30
1520   forward z
1530   right 12
1540  END FOR j
1550 END DEFine spinner
1560 :
```

Here's a program you can insert in the 'blank lines' section from 140 to 350 to show these prepackaged routines in action:

```
140 square 30
150 PAUSE 60
160 clearscreen
170 :
180 random 8
190 w=result+2
200 random 12
210 x=result+6
220 random 25
230 y=result+6
240 random 50
250 z=result+27
260 polygon w,x,y,z
270 PAUSE 60
280 clearscreen
290 :
300 random 8:x=result+14
310 random 8:y=result+16
320 random 3:z=result+1
330 triangle (y+z)
340 stairs x,y
350 spinner x,y,z
360 :
```



Now let's see what we can do with them individually.

First see how effective SQUARE can be:

```
140 :
150 background 2:pencolour 6
160 x=32
170 FOR action=0 TO 359 STEP 10
180  setheading action
190  square x
200 END FOR action
210 :
```

Now change it as follows to get a nautilus shell shape:

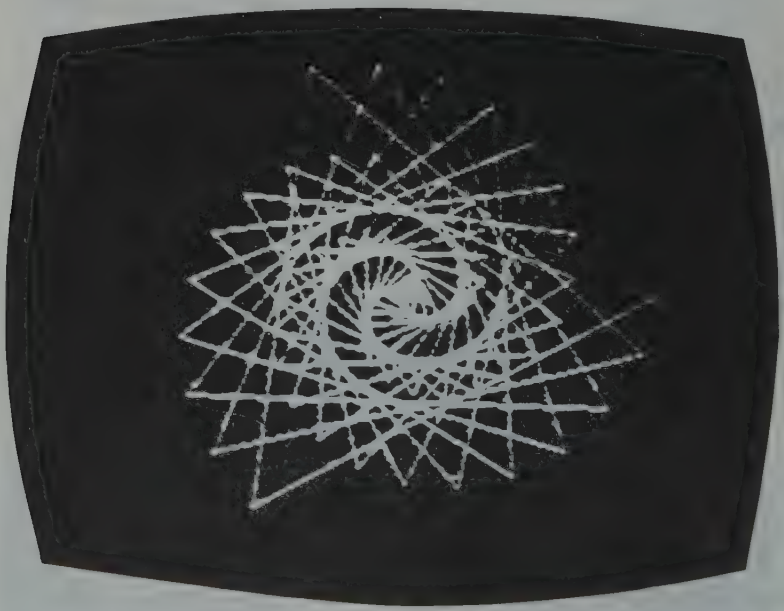
```
140 :
150 background 2:pencolour 6
160 x=1
170 FOR action=0 TO 359 STEP 10
180  setheading action
190  square x + action/7
200 END FOR action
210 :
```

From the shell we move to a windmill (of sorts):

```
140 REMark     Windmill
150 background 0:pencolour 7
160 x=34
170 FOR action=1 TO 10
180  setheading action*36
190  triangle x
200 END FOR action
```

Animation is possible, after a fashion, as our *Spinning Triangle* demonstrates:

```
140 REMark   Spinning triangle
150 background 0
160 x=24
170 FOR action=1 TO 20
180  FOR colour=1 TO 2
190   pencolour 6+colour
200   setheading action*36
210   triangle x
220  END FOR colour
230 END FOR action
240 :
```



In tribute to crystalline flowers comes *Glass Magnolia:*

```
140 REMark   Glass Magnolia
150 background 0
160 :
170 FOR action=1 TO 100
175  random 4:shift=result
180  FOR colour=1 TO 3
```

```
190  IF colour=1 THEN x=44:pencolour
(7-shift)
195  IF colour=2 THEN x=19:pencolour
5
198  IF colour=3 THEN x=30:pencolour
(3+shift)
200   setheading (action+shift)*36+5*
colour
210   triangle x
220  END FOR colour
230 END FOR action
```



A *Crossed Lattice* is easy:

```
140 REMark  Crossed Lattice
150 x=25
190 FOR t=1 TO 360 STEP 45
200  setheading t
210  square x
240 END FOR t
250 :
```

And this gives birth to two *Church window* programs:

```
140 REMark   Church window
150 x=25
190 FOR t=1 TO 720 STEP 45
200  setheading t
205  random 7:pencolour result+1
210  square x
220  random 7:pencolour result+1
230  spinner 80,50,2.6
240 END FOR t
250 :
```

```
140 REMark   Church window  Mk II
150 x=32.5
160 pencolour 7
190 FOR t=1 TO 720 STEP 45
200  setheading t
205  FOR g=1 TO 3.5 STEP .5
207   pencolour 3*g
210   square x+g
212  END FOR g
220  pencolour 6
230  spinner 80,50,3.38
240 END FOR t
250 :
```

Once you've RUN the original *QLogo* program, you can enter commands directly, and see them executed immediately on the screen, rather than include them within the program. Note that you can use FILL with *QLogo,* although the effects it produces are sometimes pretty strange.

If you want to learn more about Logo, and get ideas to develop with *QLogo,* the following books may be helpful:

Logo Programming – Anne Moller (Century, Ø 7126 Ø22Ø 8, £6.95)
Discovering Apple Logo – David D. Thornburg (Addison-Wesley, Ø 2Ø1 07769 8, £1Ø.95)



Chapter Fourteen
Windows

By using windows you can set up a number of 'mini-screens' within the overall QL screen. Each of these windows acts like a separate little screen, scrolling up the text separately. You can modify the PAPER and INK colours of individual windows, and use commands like CLS in relation to them.

The command WINDOW is used to define a new window. To use it, you need to enter four numbers (or numeric variables) as follows:

WINDOW x-width, y_depth, x_start, y_start

Note that you need to use the pixel co-ordinates to create windows.

There are a number of commands which you can use in relation to windows which accept an extra final number which determines exactly how the command will behave. The commands are BORDER, PAPER, INK, STRIP, PAN and SCROLL. The extra parameter can be from zero to four, and has the following effects:

Ø whole screen
1 above and excluding cursor line
2 bottom of screen excluding cursor line
3 whole of cursor line
4 line right of, and including, cursor

You use the CURSOR command to move the screen cursor to the position of your choice within the window.

If you wish to have more than one window on the screen at a time, you have to designate the windows with 'channel numbers'. You need to precede the window coordinates within the WINDOW

statement with the channel number:

WINDOW channel_number, x_width . . .

Then, when you're dealing with the window, you add its channel number before the chosen parameter. That is, to modify the INK colour, for example, on WINDOW 3, you would use . . .

INK 3,5

. . . where 3 is the window number, and 5 is the INK colour.

The next program, *Windows,* will form a window on the screen when you follow the instructions from line 170 and "Press a key . . .":

```
100 REMark Windows
110 PAPER 7:INK 2
120 CLS:CLS #0
130 REPeat loop
140  IF INKEY$="" THEN EXIT loop
150 END REPeat loop
160 AT 0,0
170 PRINT "Press a key to form window
"
180 REPeat loop
190  IF INKEY$<>"" THEN EXIT loop
200 END REPeat loop
210 PAPER 2:INK 6
220 WINDOW 100,100,90,80
230 CLS
240 AT 3,0
250 PRINT "Hi there"
```

You can put several windows on the screen at once, as our next demonstration program will indicate. The windows are 'opened' using an OPEN statement (see lines 140 through to 170). The OPEN command is used to initialise *devices,* a term which is broad enough to encompass windows. The number which follows the "con_" is the width of the window. This is followed by its depth (the two are linked by an "x"), and then (after an "a") by the window's measurement across and down (again with an "x" between them).

You'll see convincing evidence of the effectiveness of windows with our *Window Demo Mk II* which opens four windows on the screen, overlapping them, and scrolling text within them:

```
100 REMark Window Demo Mk II
110 PAPER 1:CLS:CLS #0
120 a$="Six is the colour of my true
love's hair as it passes through the
window of the complex equations there
 and i sing as i dance and i pass thr
ough the moon and i call for a pint t
o share with rhyming i'the gloom"
130 a$=a$ & a$
140 OPEN #5,con_130x130a90x15
150 OPEN #6,con_130x130a60x60
160 OPEN #7,con_130x130a205x105
170 OPEN #8,con_130x130a180x45
180 REPeat demo
190 PAPER #5,7:CLS #5
200 PRINT #5,a$(1 TO RND(100 TO 418))
210 PAPER #6,6:CLS #6
220 PRINT #6,a$(1 TO RND(100 TO 418))
230 PAPER #7,3:CLS #7
240 PRINT #7,a$(1 TO RND(100 TO 418))
250 PAPER #8,2:CLS #8
260 PRINT #8,a$(1 TO RND(100 TO 418))
270 IF INKEY$<>"" THEN EXIT demo
280 END REPeat demo
290 CLOSE #5:CLOSE #6
300 CLOSE #7:CLOSE #8
```

Channels

You'll be using channels frequently when working with the QL, so it's important that you have a clear idea of what they are. Very simply, they are the pipes along which information comes out of, and goes into, things hooked onto the QL ('devices').

Before you can use one of these pipes, you have to open it with the command OPEN (and where you've finished with it, you close it

with the command CLOSE). Some of the pipes open up automatically when you turn your QL on. We call these the 'default channels', and lead to 'default devices'. The screen is a default device. If you don't specify, for example, that PRINT output should go somewhere via another pipe, it will only turn up on the screen. You can't close a default channel.

You identify channels by giving them numbers. Once you've given the channel a number, that number is all the QL needs to know the device you want it to use.

Here is a summary of the I/O control words on the QL:

```
BAUD n
CLOSE #n
COPY_N name to name
DELETE name
DIR [#N,] name
FORMAT [#N,] name
NET net station number
OPEN #n, name
OPEN_IN . . .
OPEN_NEW . . .
```

And these are for I/O devices:

```
SCR [_size] [Aposition]
CON [_size] [Aposition] [_keyboard key length]
SER [n] [O/E/M/S] [H/I] [R/Z/C]
MDVn_name
NET [I/O] [_other station]
```

Chapter Fifteen
The Sound of Music

The QL's sound is controlled with the command BEEP, as is the Spectrum's sound. However, unlike the Spectrum BEEP which accepted only two parameters (controlling duration and pitch), the QL allows you a great deal of control, with seven parameters to play with.

The command is used as follows:

```
BEEP d,p1,p2,gx,gy,w,f,r
```

In this command, d is the duration (–32768 to 32767); p1 and p2 are pitches (Ø to 255;) gx which controls the time interval between steps in moving from p1 to p2; gy which defines the size of the steps taken at time interval gx; w is wrap, which controls the number of times the sound repeats; f (Ø to 15) and r, which add random noise to the sound output.

We will start our investigation of BFEP by entering a few programs which only use two numbers after the command. Try these two for size:

```
10 REMark Brrh!
20 FOR NOISE=Ø TO 130 STEP 2
30   BEEP 100,NOISE
40 END FOR NOISE

10 REMark Fluupp!
20 FOR NOISE=125 TO Ø STEP -5
30   BEEP 100,NOISE
40 END FOR NOISE
```

As you can hear, the different step sizes in the FOR/END FOR loops and their different directions produce quite different sounds.

Including a PAUSE within the loop can produce different sounds as well:

```
10 REMark Nnaahh...
20 FOR count=1 TO 3
30  FOR NOISE=85 TO 20 STEP -.5
40   BEEP 5000,225-NOISE
50   PAUSE .3
60  END FOR NOISE
70 END FOR count
```

The PAUSE can be changed during the loop:

```
10 REMark High-tower take-off
20  FOR NOISE=1 TO 241 STEP 4
30   BEEP 3000,NOISE
40   PAUSE 29/NOISE
50  END FOR NOISE
```

More than one BEEP can be held within a loop to modify the sound further:

```
10 REMark Laser attack
20 FOR count=1 TO 6
30  FOR NOISE=4 TO 24 STEP 2
40   BEEP 3000,NOISE
50   BEEP 3000,NOISE+3
60   BEEP 3000,NOISE-3
70  END FOR NOISE
80   PAUSE 5
90 END FOR count
```

Now, let's try including eight numbers after BEEP to see what effect they have:

```
10 REMark International Connection
20 FOR sound=1 TO 12
30  a=RND(1 TO 10)
40  b=a+RND(1 TO 10)
50  BEEP 10000,a,b,1,2,-9,0,0
60  PAUSE (16+a)
70 END FOR sound
```

Our next routine introduces (in line 70) the word BEEPING. BEEPING returns a zero if the QL is not producing any sound through BEEP, and a 1 if it is. The REPeat/END REPeat wait loop, then, holds the action until the sound has finished, before allowing the FOR/END FOR loop to continue:

```
10 REMark Data Transmission
20 FOR sound=1 TO 9
30  a=RND(1 TO 15)
40  b=15-a
50  BEEP 20000,a,b,1,2,-9,1,1
60   REPeat wait
70    IF BEEPING<>1 THEN EXIT wait
80   END REPeat wait
90 END FOR sound
```

Some quite extraordinary sounds can be produced, proving the truth of the statement in the manual that the command 'is best used experimentally rather than syntactically' (which translates as 'you can't really predict what it will sound like by studying the numbers'):

```
10 REMark Small arms battle
20 FOR sound=1 TO 11
30  a=RND(1 TO 5)
40  b=105-a
50  BEEP 9000,a,b,1,2,-9,RND(10 TO 15
),1
60   REPeat wait
70    IF BEEPING<>1 THEN EXIT wait
80   END REPeat wait
90   FOR silence=1 TO 24
100   END FOR silence
110 END FOR sound
```

Here are a few other effects you might like to include in your programs:

```
10 REMark Machine Gun
20 REPeat sound
30 BEEP 32767,2,44,7,1,15,11
40 IF BEEPING THEN GO TO 40
50 END REPeat sound
```

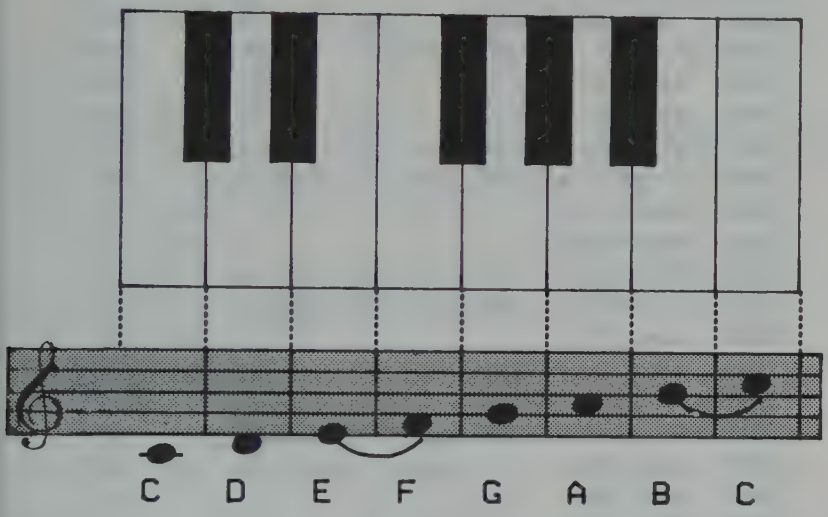
```
10 REMark Blip blip blip-blip blip
20 REPeat sound
30 BEEP 32767,188,16,7,5,13,0
40 IF BEEPING THEN GO TO 40
50 END REPeat sound
```

It's all very well me providing you with a sequence of effects, but that is of limited value and interest. Therefore, I've written a simple routine to allow the QL to generate effects at random, and tell you the arguments it has used to get those effects, so you can note down any good ones you hear for future use. The QL will produce the sound, and print the relevant numbers on the screen. Press any key for the next one:

```
10 REMark Sound potential
20 INK 7:PAPER 1
30 CSIZE 3,0
40 CLS:CLS#0
50 duration=32000
60 pitch=RND(0 TO 255)
70 pitch2=RND(0 TO 255)
80 gradx=RND(0 TO 15)
90 grady=RND(-8 TO 7)
100 wrap=RND(0 TO 14)
110 fuzzy=RND(0 TO 15)
120 randy=RND(-32767 TO 32767)
130 PRINT \\\"  Pitch = ";pitch
140 PRINT \"  Pitch2 = ";pitch2
150 PRINT \"  gradx = ";gradx
160 PRINT \"  grady = ";grady
170 PRINT \"  wrap = ";wrap
180 PRINT \"  fuzzy = ";fuzzy
190 PRINT \"  random = ";randy
200 BEEP duration,pitch,pitch2,gradx,
grady,wrap,fuzzy,randy
210 REPeat hang_on
220  IF INKEY$<>"" THEN EXIT hang_on
230 END REPeat hang_on
240 RUN
```

You can use the QL as a kind of organ, using our next program. There are twelve semitones within each octave in music, and our

organ will span one octave, so twelve notes are needed. The easiest way to understand this is to look at a piano keyboard. If you don't have a piano handy, look at our next diagram, which shows an octave on a piano, and the names of the notes underneath the keys (as well as showing what the note looks like when written down).



Here's the listing for our organ program, *Music Man:*

```
10 REMark Music Man
20 BORDER 42,0
30 CLS:CLS #0
40 CSIZE 3,1
50 PRINT \"  Music"
60 oldkey=-99
70 REPeat music
80   getkey
90 AT 3,8:INK RND(4 TO 7):PRINT m$
100 REPeat play
110 BEEP 5000,key
120 IF key<>oldkey THEN
130  EXIT play
140  ELSE oldkey=key
150 END IF
160  END REPeat play
170 END REPeat music
```

```
180 REMark ----------------
190 DEFine PROCedure getkey
200  IF INKEY$<>"" THEN GO TO 200
210  m$=""
220  key=0
230 REPeat loop
240    a$=INKEY$
250 IF a$="z" OR a$="Z" THEN
260  key=33
270  m$="C "
280 END IF
290 IF a$="s" OR a$="S" THEN
300  key=31
310  m$="C#"
320 END IF
330 IF a$="x" OR a$="X" THEN
340  key=28
350  m$="D "
360 END IF
370 IF a$="d" OR a$="D" THEN
380  key=26
390  m$="D#"
400 END IF
410 IF a$="c" OR a$="C" THEN
420  key=24
430  m$="E "
440 END IF
450 IF a$="v" OR a$="V" THEN
460  key=22
470  m$="F "
480 END IF
490 IF a$="g" OR a$="G" THEN
500  key=20
510  m$="F#"
520 END IF
530 IF a$="b" OR a$="B" THEN
540  key=18
550  m$="G "
560 END IF
570 IF a$="h" OR a$="H" THEN
580  key=17
```

```
590  m$="G#"
600 END IF
610 IF a$="n" OR a$="N" THEN
620  key=15
630  m$="A "
640 END IF
650 IF a$="j" OR a$="J" THEN
660  key=14
670  m$="A#"
680 END IF
690 IF a$="m" OR a$="M" THEN
700  key=12
710  m$="B "
720 END IF
730 IF a$="," OR a$="<" THEN
740  key=11
750  m$="C'"
760 END IF
770   IF key<>0 THEN EXIT loop
780  END REPeat loop
790 END DEFine getkey
```

The bottom row of keys on the QL (from Z to,) equals the white notes, with selected keys on the row above to represent the black notes. Here is a complete list of them, with the QL key given first, followed in brackets by its musical equivalent:

Z (C); X (D) ; C (E) ; V (F)
B (G) ; N (A) ; M (B) ; , (C')

S (C sharp/D flat); D (D sharp/E flat);
G (F sharp/G flat) ; H (G sharp/A flat);
J (A sharp/B flat)

The notes are not quite true, but are close enough for enjoying the output of the 'organ'. You may wish to try and 'tune' them more exactly. Once you've done this, you can move on with your experimental rather than syntactical exploration of the QL's sound.

Chapter Sixteen
Dealing with DATA

READ and DATA are a pair of words which appear in the same program and relate closely to each other, although they may be widely separated within the program itself.

READ enables you to tap into a bank of DATA items, to use as you wish within a program. For example, this next short program puts a number of colours into the elements of the array bright$:

```
10 DIM BRIGHT$(6,8)
11 RESTORE
12 CLS:CLS #0
20 FOR colour=1 TO 6
30  READ BRIGHT$(colour)
40 END FOR colour
50 DATA "yellow","primrose","black"
60 DATA "purple","tango","orange"
```

To check that the DATA (the items in lines 50 and 60) has been fed into the array by the READ statement in line 30, you can add the following three lines to your program, and run it again:

```
70 FOR colour=1 TO 6
80  PRINT BRIGHT$(colour)
90 END FOR colour
```

As you can see, the QL ignores lines 50 and 60 when it comes to them. DATA lines are simply read when necessary, but are otherwise ignored by the computer.

It does not matter where in the program you have the DATA statements, as you'll see when you run the following program on your trusty QL:

```
100  DATA "Ruston"
110 REMark Author data demo
120 CLS:CLS #0
130 RESTORE
140 DIM author$(7,11)
150 FOR book=1 TO 7
160   DATA "Hartnell","Big Dave"
170 READ author$(book)
180   DATA "Gifford","Shaw"
190 END FOR book
200 FOR output=7 TO 1 STEP -1
210 PRINT author$(output)
220   DATA "Vincent","Hutt"
230 END FOR output
```

The program will happily find the DATA, even though it is scattered about somewhat (and some of it is actually within the FOR/END FOR loop), printing out the following when you run it:

```
Hutt
Vincent
Shaw
Gifford
Big Dave
Hartnell
Ruston
```

RESTORE

The third word which is used in conjunction with READ and DATA is RESTORE. RESTORE moves the 'data pointer' back to the start of the DATA block, as this program shows:

```
100 REMark RESTORE demo
110 CLS:CLS #0
120 RESTORE
130 DIM golly$(9,7)
140 FOR gosh=1 TO 9
150  IF gosh=4 THEN RESTORE
160  READ golly$(gosh)
170  PRINT golly$(gosh);" ";
180 END FOR gosh
```

```
190 DATA "this","is","a","demo","to"
200 DATA "show","RESTORE","in","use"
```

On a quick examination of this program, you might imagine the output would be as follows:

this is a demo to show RESTORE in use

However, when you run it on your QL, you'll actually see this:

```
this is a this is a demo to show this
```

The RESTORE in line 30 moved the data pointer back to the start of the DATA when gosh equalled four, so the READ statement started at the beginning again.

RESTORE can also go to a specific line, as this next program shows with line 150:

```
110 RESTORE
120 CLS:CLS #0
130 :
140 FOR munchies=1 TO 9
150 IF RND(1 TO 8)=4 THEN RESTORE 240
160  READ pizza$
170  PRINT pizza$
180 END FOR munchies
190 :
200 DATA "vegetarian","cockroach"
210 DATA "four seasons","anchovies"
220 DATA "humble pie","apricot"
230 DATA "little green bag"
240 DATA "line 100","wise","weird","w
onderful"
250 DATA "wacky","wasted","wanton"
260 DATA "mean mr. mustard"
270 DATA "power pill"
```

If the random number generated in line 150 equals 4, then the data pointer goes to line 240, and starts reading from there. This means the output could be (if the 4 was generated after the fifth item was read):

```
vegetarian
cockroach
four seasons
anchovies
humble pie
line 100
wise
weird
wonderful
```

DATA is often used to feed specific values into arrays at the start of a program.

Chapter Seventeen
Extending your Vocabulary

Random Numbers

Random numbers are of great use in game and simulation programs. The QL uses a unique method of specifying the numbers you want. The first one simply uses the word RND, without any parameters, as follows:

```
10 REPeat rand_demo
20   PRINT RND;" ";
30 END REPeat rand_demo
```

If you run this on your QL, you'll get an apparently infinite list of numbers between zero and one:

```
.4514049 .2487407 .3029405 .5438805 7
.759424E-2 .5459563 .8145022 .1482666
 .6346341 .2951227 .7685726 .8432614
.3577894 1.52361E-2 .4088165 .1298482
 9.610956E-2 .1188931 .2070633 .50746
91 .1515403 .7968125 .8684341 .294093
5 .4031911 .132825 .1528747 .270505 2
.925868E-2 .3868327 .3255952 .5863073
 .1391077 .3832364 4.891591E-2 .36514
93 .6280066 .9423324 .5280188 .446667
7 .5670202 .2921778 .7231061 .7026495
 .4405589 .3984238 .4404645 .3649129
.5440834 .1496195 .1149364 .8024069 .
8544661 .3354825 9.629586E-2 .1850296
```

If you want randomly-generated integers between zero and some upper limit, you use a line as in 30 in the following brief program:

```
10 upper_limit=7
20 REPeat rand_demo
30  PRINT RND(upper_limit);" ";
40 END REPeat rand_demo
```

If you defined upper_limit as 7, you might get output as follows:

```
7 1 1 6 5 7 5 0 2 1 1 7 1 0 2 4 4 1 5
 5 5 0 3 4 7 5 1 0 2 6 6 4 0 0 5 1 7
0 2 7 3 6 5 4 4 5 4 6 0 2 0 4 6 0 4 4
 7 6 0 2 6 6 0 0 6 3 5 2 0 6 2 7 7 5
4 1 3 5 7 2 1 3 4 1 1 4 7 6 5 3 0 7 2
 5 4 4 5 3 3 6 6 5 0 0 4 1 3 4 3 2 3
6 4 0 0 2 5 2 7 4 5 5 1 3 5 7 4 3 7 1
```

It is quite likely that you will need your QL to generate random numbers in some other range, rather than from 0 to an upper limit. In this case, you make use of the word TO, which Sinclair has brought to a fine art in his BASICs, as follows:

PRINT RND (lower_limit TO upper_limit)

In program form, it might appear like this (which would give randomly-generated numbers between 95 and 105):

```
10 upper_limit=105
20 lower_limit=95
30 REPeat rand_demo
40  PRINT RND(lower_limit TO upper_li
mit);" ";
50 END REPeat rand_demo
```

The output of this program could be as follows:

```
105 105 101 105 105 96 100 101 103 95
 103 105 102 102 105 102 96 103 102 9
6 104 96 105 95 100 98 105 97 99 102
95 99 98 103 96 98 104 97 101 99 99 9
5 102 99 99 100 99 101 96 103 100 102
```

Here's a brief (and easy-to-play) game which is based largely on the production of random numbers. This sample run makes it clear what you have to do:

```
********************************

   My first number is 6
        My second is 12

         You have $20

How much do you bet my next
 one lies between 6 and 12?
           10

     My number is 5

   Sorry, you lose $10


********************************

   My first number is 13
        My second is 8

         You have $10

How much do you bet my next
 one lies between 13 and 8?
           3

     My number is 3

   Sorry, you lose $3


********************************

   My first number is 11
        My second is 4

         You have $7

How much do you bet my next
 one lies between 11 and 4?
```

```
5

My number is 8

Well done, you win $10

**********

My first number is 12
My second is 7

You have $17

How much do you bet my next
one lies between 12 and 7?
4

My number is 2

Sorry, you lose $4

**********

My first number is 13
My second is 8

You have $13

How much do you bet my next
one lies between 13 and 8?
2

My number is 5

Sorry, you lose $2

**********
```

```
My first number is 2
     My second is 5

     You have $11

How much do you bet my next
 one lies between 2 and 5?
          Ø

          Coward!!

     My number is 1Ø


******************************

     My first number is 1
     My second is 6

     You have $11

How much do you bet my next
 one lies between 1 and 6?
          11

     My number is 7

     Sorry, you lose $11

That's the end of the game

     You are broke!
```

And here's the listing, so you and your QL can enjoy a round or two of *Acey Duecy:*

```
10 REMark Acey Duecy
20 REMark Random numbers in action
30 CSIZE 1,0
40 PAPER 1:INK 6
50 BORDER 1,2,7
```

```
60 CLS:CLS #0
70 d=20
80 REMark **************
90 REPeat game
100 CLS
110 a=RND(1 TO 13)
120 b=RND(1 TO 13)
130 IF ABS(b-a)<2 THEN GO TO 120
140 IF ABS(b-a)>7 THEN GO TO 120
150 c=RND(1 TO 13)
160 IF a=c OR b=c THEN GO TO 150
170 PRINT \\\"   My first number is "
;a
180 PRINT "         My second is ";b
190 PRINT \\"         You have $";d
200 PRINT \\"How much do you bet my n
ext"
210 PRINT " one lies between ";a;
220   PRINT " and ";b;"?"
230 INPUT "            ";e$
240 IF e$="" THEN GO TO 230
250 IF e$<0 OR e$>d THEN GO TO 230
260 e=e$
270   IF e>d THEN GO TO 230
280 FLASH 1:PAPER 5:INK 2
290 IF e<1 THEN PRINT \\"          Co
ward!!"
300 PRINT \"    My number is ";c;"
     "
310 FLASH 0:INK 6:PAPER 1
320 IF e<1 THEN GO TO 390
330 IF NOT(c>a AND c<b OR c<a AND c>b
) THEN GO TO 370
340 PRINT \"  Well done, you win $";2
*e
350 d=d+2*e
360 GO TO 390
370 PRINT \"   Sorry, you lose $";e
380 d=d-e
390 FOR j=1 TO 1000:END FOR j
400 IF d>0 THEN END REPeat game
```

```
410 REMark ***************
420 PRINT \\" That's the end of the g
ame"
430 PRINT \\"       You are broke!"\\\
\
```

PAUSE

The command PAUSE is used when you want the program to stop for a designated time. The word PAUSE is followed by a number:

```
PAUSE 200
```

Each 'pause unit' is 20ms.

Making REMarks

A REMark statement is used within a program when you want to explain something, but do not want the computer to take any notice of the line. Here's an example:

```
10 REMark This program tests RND
```

A REMark statement can come after another one in a program line:

```
100 PRINT "You've won the final round
 of Blocko!":REMark end of game
```

Of course, you must not start a multi-statement line with a REMark statement:

```
450 REMark The end:PRINT "You have won"
```

The second half of this line will never be executed.

STOP

Naturally enough, this is used to stop a program:

```
100 ::::
120 IF answer$="Constantinople" THEN
PRINT "Well done!":STOP
130 PRINT "That is not the answer I w
as looking for"
140 :::::
```

INPUT

You use this to enter data (numeric or string) into a program which is currently running on the QL. The program form is:

```
10 INPUT "What is your name? ";name$
```

```
10 INPUT "How old are you? ";age
```

You can use the separators to format your questions if you wish to accept more than one input from a single INPUT statement:

```
10 INPUT "How many?"!num,"Age?"!av
```

```
10 INPUT "First name?"!f$,"Surname?"!s$
```

INKEY$

You use INKEY$ to read single keystrokes from the keyboard (or from another channel if one is designated). The program will pass over an INKEY$ line if the particular character being sought is not present:

```
100 REPeat check
110   IF INKEY$="" THEN EXIT check
120 END REPeat check
130 :
```

```
140 REPeat check2
150  b$=INKEY$
160  IF b$<>"" THEN EXIT check2
170 END REPeat check2
180 :
190 IF b$="a" THEN PRINT "OK"
```

This brief program first uses INKEY$ to read the keyboard, and holds the program there if any key is being touched (that is, INKEY$ is not returning the empty string). Once the keyboard is clear, the QL moves to line/40, which holds the action until a key is touched. Then the program moves onto line 190. If the key being touched is the lower case "a" (that is, INKEY$="a") then the QL prints up "OK".

Getting LISTed

The command LIST enables you to get your current program listed out on the screen, or to some other device.

It can be used in four different ways:

LIST 100 to 120 will list out line 100 and then all lines up to and including line120

LIST 100 TO lists all the lines in the program, starting from 100 and going through to the last line in the program

LIST TO 100 will start at the first line in the program, and list up to and including line 100

LIST 100 will just list line 100

Other Commands

PEEK and POKE allow you to look inside (PEEK) a memory location or modify (POKE) it. You use them as follows:

PEEK address
POKE address, nnn

In this form, the QL knows you want byte access (8 bit). Adding, 1 to the end of the line as follows . . .

PEEK address, 1
POKE address, nnn, 1

. . . indicates that a word access (16 bit) is wanted.

You can add ,Ø to the end of the statements, which indicate the byte access, but if you leave it off, the QL assumes you mean the ,Ø case.

AUTO is used to generate line numbers automatically when entering a program. You stop the action of AUTO with the ESC key.

CLEAR is used to strip all variables of their values. When you RUN a program, the QL CLEARs the variable store automatically.

Whereas CLEAR wipes out the variable values, NEW is used to clear out everything: variables and your program. It should be used with care.

Chapter Eighteen
Using Files

You can store material permanently in the form of files on a microdrive cartridge, then read the file back into the QL's RAM when you need it later. Basically, you open a file with a command like this:

```
OPEN_NEW #6,MDV1_name
```

In this case, the file is numbered 6. You can give a file any channel number from 3 to 15, assuming that the channel has not already been assigned to something else (like the printer). Channels Ø, 1 and 2 are already in use by the QL.

To fill the file, you simply use it like the screen, PRINTing to it, or INPUTting material to it. Therefore, both these forms of input are valid:

```
PRINT #6,stuff
```

```
INPUT #6,stuff
```

The 'stuff' you feed in can be from such sources as the output of a program, generated while it is running, the result of human interaction with a program, or DATA which is READ from statements within a program.

Once you have fed your material in, you need to shut down the file with a CLOSE command which is used as follows:

```
CLOSE #6
```

You open a file with OPEN_NEW and, once it is in place, use it with OPEN_IN, as you'll see with the sample program which follows. This program prints out the contents of the QL ROM from 14ØØØ to

14750 (38A4 to 399E) and was written by W. T. Cowhig, and first published in *Quanta:*

```
100 REMark Opening, using mdv files
110 REMark      W T Cowhig, SALE
120 :
130 CLS:CLS #0
140 k=14500
150 REMark This prints out in ASCII
160 :
170 OPEN_NEW #4,mdv1_ASCEE:REMark
   Remove NEW if file already exists
         and you are re-using it
180 :
190 FOR N=0 TO 250
200  IF (PEEK(k+N))<=128 THEN b=(PEEK
(k+N)):PRINT !CHR$(b);:PRINT #4,!CHR$
(b);
210  IF PEEK (k+N)>128 THEN c=(PEEK(k
+N)):PRINT !c;:PRINT #4,!c;
220 END FOR N
230 :
240 t$="This is ROM from 14500 to 147
50 -    38A4 to 399E"
250 PRINT t$
260 PRINT #4,t$:CLOSE #4
270 :
```

Once the file is on the microdrive cartridge, you can read it to the screen with a line like this:

```
280 COPY mdv1_ASCEE TO scr_
```

Chapter Nineteen
Financial Modelling

Although the quartet of programs provided with your QL will serve many ends, there are times when a simpler, more direct program would prove useful. I needed a very simple 'projection of a trend' program, and found it more useful to write my own than try and adapt the Psion material. It seemed to me that the program would be useful to share, as it can be adapted to a great many different situations.

This program provides a crude version of a few of the facilities of spread-sheet programs. You'll see that the program will take monthly sales figures, or number of rejects, or turnover in staff or any other count of an event which occurs fairly regularly, and which has been recorded at regular intervals. From this information, the program can extrapolate future trends, assuming the factors involved in the observed change remain much the same.

When you run the program, you'll be asked if you want a copy dumped to your printer. Next you'll be asked to enter the number of months for which figures are available. You can, of course, change months to days or years or whatever time period you are working with. You'll then be asked to enter the figure for each month. The QL will then work out the approximate change from month to month. Comparing month two with month one, month three with month two and so on. The average percentage change will then be given.

You're then able to ask the QL to make an extrapolation of the results, specifying the number of months for which you want figures projected, and whether you want this projection based on the last month for which you have a figure, or on the average result per month. After this you can choose to run the projection again; to run through the output once more without entering your figures again;

to run the program again from the beginning; or to terminate the run.

Here's the program listing:

```
10 REMark Financial Model/Projections
20 CLS:CLS #0
30  hardcopy
40 REPeat answer
50 INPUT \\\"For how many months are
figures                 available? ";m
60  IF m>1 THEN EXIT answer
70 END REPeat answer
80 tot=0
90 CLS
100 DIM a(m):DIM b(m)
110 REMark --------------
120 FOR ak=1 TO m
130 AT 17,0
140 PRINT "
          "
150 AT 17,0
160 INPUT "Enter figure for month ";(
ak);"    ";a(ak)
170 IF ak<17
180  q=ak
190  ELSE q=ak-16
200   IF q>16 THEN q=q-16:GO TO 200
210 END IF
220 AT q,2
230 PRINT "Month ";ak;" -   ";a(ak)
240 IF z=1 THEN PRINT #3,"  Month ";a
k;" -   ";a(ak)
250 tot=tot+a(ak)
260 END FOR ak
270 AT 17,0
280 PRINT "
          "
290 REMark ---------------
300 END IF
310 AT q,0
```

```
320 av=tot/m
330 FOR bk=2 TO m
340   b(bk)=(100-(a(bk-1)*100/a(bk)))
350 END FOR bk
360     PAUSE 100
370 CLS
380 PRINT \"-------------------------
----------"
390 IF z=1 THEN PRINT #3,\"----------
-------------------------"
400 PRINT \"Difference between months
:"
410 IF z=1 THEN PRINT #3,\"Difference
 between months:"
420    PRINT
430    IF z=1 THEN PRINT #3
440 FOR ak=2 TO m
450  PRINT "  Month ";ak-1;" to month
 ";ak;" - ";INT(b(ak));"%"
460  IF z=1 THEN PRINT #3,"  Month ";
ak-1;" to month ";ak;" - ";INT(b(ak))
;"%"
470 END FOR ak
480 REMark ------------------
490     PAUSE 100
500 total=0
510 FOR ak=2 TO m
520  total=total+b(ak)
530 END FOR ak
540 REMark -------------
550 average=INT(total*100/(m-1))/100
560 PRINT \"-------------------------
----------"
570 IF z=1 THEN PRINT #3,\"----------
-------------------------"
580 PRINT \"  Average change is ";ave
rage;"%"
590 IF z=1 THEN PRINT #3, \"  Average
 change is ";average;"%"
600     PAUSE 100
610 PRINT \"-------------------------
----------"
```

```
620 IF z=1 THEN PRINT #3,\"----------
------------------------"
630 PRINT \"Now a projection of chang
e."
640 IF z=1 THEN PRINT #3,\"Now a proj
ection of change."
650 INPUT \"How many month's projecti
on do you              want? ";number
660 PRINT \"-------------------------
---------"
670 PRINT \"Final recorded month was
";a(m)
680 PRINT \"Average per month was ";1
0*(INT (av/10))
690 PRINT \"Do you wish to base proje
ction on           1 - Final month; o
r                  2 - Average month?
"
700 REPeat answer
710  a$=INKEY$
720  IF a$="1" OR a$="2" THEN EXIT an
swer
730 END REPeat answer
740 d=a$
750 PRINT \"-------------------------
----------"
760 IF z=1 THEN PRINT #3,\"----------
------------------------"
770 IF d=1 THEN
780  e=a(m)
790  ELSE e=10*INT(av/10)
800 END IF
810 PRINT \"  Month 1, recorded: ";a(
m)
820 IF z=1 THEN PRINT #3,\"  Month 1,
 recorded: ";a(m)
830 REMark ------------------
840 FOR ak=2 TO number
850  e=e+average*e/100
860  PRINT "  Month ";ak;", projected
 ";INT(e)
```

```
870  IF z=1 THEN PRINT #3,"  Month ";
ak;", projected ";INT(e)
880 END FOR ak
890 REMark ---------------------
900    PAUSE 100
910 PRINT \"-------------------------
---------"
920 IF z=1 THEN PRINT #3,\"----------
-------------------------"
930 PRINT "Enter your choice:"
940 PRINT \"   1 - To run projection
again"
950 PRINT "   2 - To run through outp
ut again          but without enterin
g                 figures again"
960 PRINT "   3 - To run the program
                  again from the begi
nning"
970 PRINT "   4 - To stop"
980 REMark ------------------------
990 REPeat answer
1000  a$=INKEY$
1010  IF a$>"0" AND a$<"5" THEN EXIT
answer
1020 END REPeat answer
1030 REMark ------------------------
1040 p=a$
1050 SELect ON p
1060 ON p=1
1070   GO TO 630
1080 ON p=2
1090   GO TO 400
1100 ON p=3
1110   RUN
1120 ON p=4
1130   STOP
1140 END SELect
1150 STOP
1160 REMark ---------------------
1170 DEFine PROCedure hardcopy
1180  z=0
```

```
1190 PRINT \"Do you want a copy to yo
ur printer                (Y or N)"
1200 REPeat answer
1210  a$=INKEY$
1220  IF a$="Y" OR a$="y" OR a$="N" O
R a$="n" THEN EXIT answer
1230 END REPeat answer
1240   CLS
1250 IF a$="N" OR a$="n" THEN END DEF
ine hardcopy
1260 REMark Adjust next two lines for
             your printer system
1270    OPEN #3,ser1
1280    BAUD 9600
1290    z=1
1300 END DEFine hardcopy
```

And this is an example of it in use:

```
  Month 1 -  1234
  Month 2 -  1345
  Month 3 -  1456
  Month 4 -  1567
  Month 5 -  1678
  Month 6 -  1789
  Month 7 -  1899
  Month 8 -  1912
  Month 9 -  2123
  Month 10 -  2234
  Month 11 -  2345
  Month 12 -  2567

------------------------------------

Difference between months:

  Month 1 to month 2 - 8%
  Month 2 to month 3 - 7%
  Month 3 to month 4 - 7%
  Month 4 to month 5 - 6%
  Month 5 to month 6 - 6%
```

```
  Month 6 to month 7 - 5%
  Month 7 to month 8 - 0%
  Month 8 to month 9 - 9%
  Month 9 to month 10 - 4%
  Month 10 to month 11 - 4%
  Month 11 to month 12 - 8%

-----------------------------------

  Average change is 6.41%

-----------------------------------

Now a projection of change.

-----------------------------------

  Month 1, recorded: 2567
  Month 2, projected 2731
  Month 3, projected 2906
  Month 4, projected 3092
  Month 5, projected 3291
  Month 6, projected 3502
  Month 7, projected 3726
  Month 8, projected 3965
  Month 9, projected 4219
  Month 10, projected 4490
  Month 11, projected 4778
  Month 12, projected 5084

-----------------------------------
```

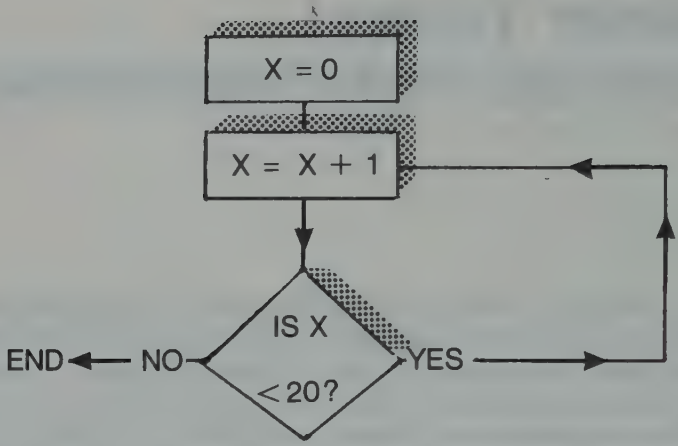
Chapter Twenty
Improving your Programming

There will come a point in your development as a programmer on the QL when you'll have mastered the use of much of SuperBASIC, and can now concentrate on writing better programs, programs which work after relatively little debugging, which are easy for others to understand and operate, and which are written logically and elegantly. As I'm sure you've realised by now, SuperBASIC is a tool which encourages the development of elegant, structured programs.

Your programs will be more likely to run with little revision after you first enter them into the QL if you plan out, in advance, exactly what you want to achieve. That is, you should think of the 'outputs' (what will appear on the screen, the printer and/or the microdrive file), the way in which that output will be generated and formatted, and the 'inputs', the interaction with the human user.

You can start with a flow chart, a series of boxes and other shapes joined by lines, which show the flow of action and decision-making within the QL while the program is running. The shapes are not too important, and I suggest you stick to just two: a rectangle for most actions your QL must carry out, with a diamond shape each time the QL has to make a decision. The corners of the diamond can be used, as you can see in the diagram, to cater for the alternatives facing the QL.

The diagram shows the flow chart of a program which sets the variable X equal to zero, then adds one to it. The value of X is then checked. If X is found to be less than 20, the program goes back to add one to X again. This continues until the value of X equals 20.



If you were using another microcomputer with a more 'normal' version of BASIC, you might then translate the flow chart into a program as follows:

```
10 REMark Ordinary BASIC
20 LET X=0
30 LET X=X+1
40 IF X<20 THEN GO TO 30
50 STOP
```

This will work perfectly adequately on just about any computer which uses BASIC, including the QL. However, it is not the most well-structured program in the world.

A QL version of that program could be:

```
10 REMark SuperBASIC
20 x=0
30 REPeat increment
40   x=x+1
50   IF x=20 THEN EXIT increment
60 END REPeat increment
70 STOP
```

Now, it may not be immediately apparent to you that this is any improvement over the first program. It even has two extra lines. However, if you look at it carefully, you'll see it has a number of

useful ideas which can be profitably used in other QL programs you write.

For a start, the action is held within a REPeat/END REPeat loop. Instead of the GO TO (in the original line 40) we have this loop which cycles until (as determined by line 50) x is equal to 50. Once the QL finds that x is equal to 20, the EXIT is triggered and the program leaves the loop. The real advantage of the QL program over the other one is that it is possible to make it extremely clear what is going on in a program. The loop name ('increment') is perfectly clear, and the boundaries of the loop are indicated by indenting the material within the loop (lines 40 and 50).

A flow chart models the flow of action and decision-making within a program. A well-structured QL program reflects the clarity of the flow chart.

The flow chart is very useful for picking up potential bugs at the earliest stages. You may, for example find that one condition the program will test to EXIT a loop will never be fulfilled, leading to the program being trapped in an infinite loop. Other parts of your program may be bypassed completely, because the condition which triggers entry into that part of the code will never be satisfied.

Once you've devised a flow chart for your program and you've run through it mentally a few times, so that the most obvious bugs are removed, you should reduce the flow chart to a series of procedure calls, possibly held within a master REPeat/END REPeat loop. It may seem pretty silly to do this for a simple program like our *set x equal to zero, add one, check if less than 20* program. However, this method comes into its own with more complex programs.

As you can see in the major programs in this volume, I'm in favour of starting each program with a series of procedure calls, with each action of the program being looked after by a separate procedure. These procedures, may of course, call up other procedures, and may include repetition and assignment constructions within them. If the steps within a program have to be performed several times in a particular sequence, the series of procedure calls can be cycled through, over and over again, until a particular condition is met which triggers an EXIT to end the run.

You'll recognize how useful this approach to programming can be when you get to the debugging stage of your program. If there *is* a bug, it is likely to be within a single procedure, so it will be relatively easy to pin down the procedure which contains the bug, rather than having to work right through the program trying to track it down.

Working with procedure 'modules' in this way also allows you to *test* sections of your program in isolation, even before the entire program is finished. I'll try to make that statement clear with an example.

Here are the opening lines of an imaginary program called *QL Checkers:*

```
100 REMark QL Checkers
110 :
120 initialise
130 :
140 REPeat game
150  print_board
160  accept_player_move
170  print_board
180  computer_moves
190  IF computer_win OR human_win THE
N EXIT game
200 END REPeat game
210 :
220 IF computer_win THEN
230   PRINT "I have won, human!"
240  ELSE
250   PRINT "You're the winner!"
260 END IF
270 :
280 STOP
```

You could have quite a bit of this program running on your QL, and tested (such as the *initialise* procedure, the one which prints the board and the procedure which accepts the player's move), before you even turned your attention to how on earth you were going to get the computer to make its move.

You would then know – for example – that you would not need to waste any extra thought on whether or not an error in the board-

printing procedure was the cause of odd output to the screen. Having tested the procedures which printed the board and the player move, you'd know the error probably lay within the procedure which allowed the computer to make its move.

All you need to do, in the early stages of development, is to make up a 'dummy procedure' which just contains the DEFine and END DEFine lines, with a central line saying something like PRINT "The QL moves here". The computer will accept that line, and when you're running the incomplete program, will demonstrate that the program is flowing in the correct sequence, even if quite a few vital procedures have not yet been written.

Notice also in the sample listing that I've used 'blank' lines (11Ø, 13Ø, 21Ø and 27Ø) containing just a colon each, in order to break the program listing into visually discrete units. There is little chance of running out of memory on the QL, so you may as well put in things like this that, on certain other computers, would be considered expensive luxuries. The 'blank' lines are, in fact, more useful visually than REMark statements, but are less use to you in highlighting what sections of the program do. If, however, you have chosen the names of your procedures and REPeats carefully, these will indicate clearly the purpose of each section of your program.

As well as the blank lines, you'll see I've indented the material within the REPeat game loop, and also within the IF/THEN. This helps make it clear what is going on.

Line 19Ø reads almost like a sentence in English. It makes use of 'Boolean variables'. These can take a value of either zero or one, with zero indicating 'false', and one indicating a 'true' condition. The variable computer_win is set equal to zero in the *initialise* procedure. The same thing, of course, happens to the human_win variable. Then, if a 'win test' condition (which is probably part of the print_board procedure) finds that either the QL or the human has won, the appropriate variable is set equal to one. This triggers the exit from the *game* loop, leading into the IF/THEN which prints the message as to the winner.

Although this description has centred around a game, the ideas it presents will stand you in good stead in any programs you write on the QL.

Work It Out First

As I pointed out at the beginning of this chapter, I suggest you try and do as much writing of your program as possible before you turn your QL on, even though there is a great temptation to dive straight into the computer and start punching in code. You'll find that the discipline of writing it out by hand in advance will serve you in good stead and should, in the long run, produce a better program than might otherwise have been the case. Overall, you'll probably end up spending less time on the program working in this way than you would if you began the process sitting at the computer keyboard.

It took me a while to learn this lesson. Although I had read suggestions along the lines of 'work out exactly what you're going to enter *before* you start at the computer' in several books, I tended to just jump right in without much prior thought.

Although I worked out rough flow charts, and had an idea what sort of display organisation I wanted, I certainly did not write much program out on paper before starting at the computer. Then, I once found myself stuck for a two-week period without a computer and ideas for several programs just itching to be written. I had to write them out in an exercise book.

The relative ease with which the programs were debugged when they were eventually entered into the computer, and the complexity of the programs I wrote in this way (including my first chess program) convinced me that this was the way I would work from then on. It is amazing how much cleaner a program can be if all the rough working out is done on paper, rather than on the computer screen.

Explicit Prompts

It is very useful, when writing a program, to keep in mind how the program will appear to a stranger when it is run by him or her for the first time. If there is an *input prompt* required, it is far more useful if the program prints up something like "HOW MANY HOURS HAS THE EMPLOYEE WORKED THIS WEEK?" instead of just "INSERT HOURS?" or an almost useless lone question mark.

The same suggestion applies to print output. It is far better that your program is written so that it prints out THE NUMBER OF HOURS WORKED ON FULL PAY THIS WEEK IS 27, rather than HOURS, FULL: 27 or an unsupported 27. Of course, providing explicit input prompts and output PRINT statements consumes memory as well as typing time when entering the program, but the contribution they make to the final program means the trouble involved is well worth it.

REM Statements

While exact PRINT statements and input prompts will help a person running a program make sense of it, REM statements can help make the program clear to those who are examining the listing for the first time. REM statements (which are, as you know, ignored by the computer when a program is run) should be used to help illuminate the flow of logic within the program, and what is happening in certain places within it. This is especially important in parts of the program where decisions are made, or calculations are carried out.

Variables

It is worth considering the use of explicit names for variables, using either the word in full (such as HOURS as a variable name in a payroll program for hours worked) or an abbreviated version of this (such as HR) which has a fairly obvious meaning. You'll discover that this makes it easy, when working on a program, to keep track of your variables. This will help you with the initial debugging and later on as well if you need to improve or extend the original program.

Explicit variable names also help make your code more 'transparent' so other programmers can work out what the various parts of the program are intended to do. You'll also find it a great help to yourself when you return to the program at a later date. It is surprising how code which seemed incredibly clear in terms of its purpose when you entered it, becomes exceedingly dense when you return to it after a long break.

Here's an example of a program which uses explicit variable names. You'll see that it is very clear what is happening in each line within

the FOR/END FOR loop (lines 110 to 160), because the variable names spell out the purpose of each arithmetic manipulation clearly:

```
10 REMark Simple and Compound
20 REMark      Interest
30 CLS:CLS #0
40 INPUT \\\"  Principal?  ";principa
l
50 INPUT \\"    Interest rate?  ";int
erest
60 INPUT \\"  For how many years?  ";
years
70 CLS
80 PRINT "-------------------------
--------"
90 PRINT "Year","Simple","Compnd.","
 Diff."
100 PRINT "-------------------------
---------"
110 FOR money=1 TO years
120 simple=principal+money*principal*
(interest/100)
130 compound=INT(100*principal*(1+int
erest/100)^money)/100
140 difference=INT(100*(compound-simp
le))/100
150 PRINT money,simple,compound,;"  "
;difference
160 END FOR money
```

Here are two examples of the program in action. Notice that the input prompts, while not very chatty, at least indicate to the user what is required:

```
Principal? 100
   Interest rate? 8.25
 For how many years? 12


-----------------------------------
Year     Simple  Compnd.    Diff.
-----------------------------------
```

```
1          108.25     108.25     0
2          116.5      117.18     .68
3          124.75     126.84     2.08
4          133        137.31     4.3
5          141.25     148.64     7.38
6          149.5      160.9      11.39
7          157.75     174.17     16.41
8          166        188.54     22.53
9          174.25     204.1      29.85
10         182.5      220.94     38.44
11         190.75     239.17     48.41
12         199        258.9      59.9
```

```
Principal? 45
   Interest rate? 16
 For how many years? 5

-----------------------------------
Year       Simple     Compnd.    Diff.
-----------------------------------
1          52.2       52.2       0
2          59.4       60.55      1.15
3          66.6       70.24      3.64
4          73.8       81.47      7.67
5          81         94.51      13.5
```

Checking Input

Any input entered by the user should be checked by the program before it is accepted to ensure that incorrect data does not cause the program to crash at some future point. Whether you want string, or numeric input, it is often wise to allow for string input, which is first checked to see whether the entered material is acceptable and then, if necessary, the string can be changed into a number.

As well as ensuring that the program will reject invalid input, you should check the program to see all the inputs which it does accept produce sensible answers when processed later on in the program. For example, make sure that your program does not accept zero as a possible number if the computer must later divide by this number.

Similarly, if numbers are to be processed by a function, and then the result of this processing used for division, you must check that an apparently valid input does not turn into zero as a result of evaluation by the function.

If the information entered by the user is rejected, and a new input is requested, the program should ideally point out why the original input was not suitable, or spell out again exactly what is required (such as "ENTER A NUMBER BETWEEN 1 AND 4"). You risk making users angry if input which appears valid is continually rejected without apparent reason.

For example, in our next program, the computer is looking always for an "e" or "E" input to terminate a string of entered numbers (see lines 11Ø, 16Ø and 33Ø) once an option (arithmetic or harmonic) has been selected. The 'menu section' of the program *(PROCedure action)* needs a "1", "2" or a "3" and rejects input (see lines 46Ø and 47Ø) which is not one of these. Here's the program listing:

```
10 REMark     Arithmetic
20 REMark and Harmonic Mean
30 CLS:CLS#0
40 action
50 REMark **************
60 DEFine PROCedure arithmetic
70 AT 3,12
80 PRINT "Arithmetic Mean"
90 PRINT \"Enter the numbers you wish
 me to use"
100 PRINT "  to determine the arithme
tic mean"
110 PRINT " >Enter 'E' to end your in
put          "
120 CSIZE 3,1
130 REPeat entry
140  INPUT "    ";q$
```

```
150     IF q$="" THEN GO TO 140
160     IF q$="e" OR q$="E" THEN EXIT
entry
170     sum=sum+q$
180    tally=tally+1
190 END REPeat entry
200 PRINT \"The arithmetic mean is ",
sum/tally
210 END DEFine arithmetic
220 REMark *****************
230 DEFine PROCedure harmonic
240 AT 3,12
250 PRINT "Harmonic Mean"
260 PRINT \\"Enter the numbers you wi
sh me to"
270 PRINT "use to find the harmonic m
ean"
280 PRINT " >Enter 'E' to end your in
put              "
290 CSIZE 3,1
300 REPeat entry
310  INPUT "    ";q$
320    IF q$="" THEN GO TO 310
330    IF q$="e" OR q$="E" THEN EXIT e
ntry
340     sum=sum+(1/q$)
350    tally=tally+1
360 END REPeat entry
370 PRINT \\"The harmonic mean is ",1
/(sum/tally)
380 END DEFine harmonic
390 REMark ******************
400 DEFine PROCedure action
410 CSIZE 0,0
420 PRINT \\"Select the program you w
ant:"
430 PRINT \"  1 - Arithmetic Mean"
440 PRINT \"  2 - Harmonic Mean"
450 PRINT \"  3 - To end program"
460 z$=INKEY$
470 IF z$<"1" OR z$>"3" THEN GO TO 46
```

```
0
480 IF z$=3 THEN
490    CSIZE 3,1
500  PRINT \" OK, thanks for your tim
e"\\
510    CSIZE 0,0
520   FOR jj=1 TO 1000:END FOR jj
530 END IF
540 CLS
550 tally=0
560 sum=0
570 IF z$=1 THEN arithmetic:GO TO 410
580 IF z$=2 THEN harmonic:GO TO 410
590 END DEFine action
```

And this is the program in action:

```
Select the program you want:

  1 - Arithmetic Mean

  2 - Harmonic Mean

  3 - To end program


Arithmetic Mean
Enter the numbers you wish me to use
  to determine the arithmetic mean
 >Enter 'E' to end your input

    564.8
    564.5
    56.78
    123.993
    E

The arithmetic mean is  327.5182


Select the program you want:
```

```
  1 - Arithmetic Mean

  2 - Harmonic Mean

  3 - To end program


Harmonic Mean

Enter the numbers you wish me to
use to find the harmonic mean
 >Enter 'E' to end your input

    564.8
    564.5
    56.78
    123.993
    e


The harmonic mean is     136.898


Select the program you want:

  1 - Arithmetic Mean

  2 - Harmonic Mean

  3 - To end program

 OK, thanks for your time
```

DOCUMENTATION

The written material which accompanies a program is often called documentation. It is useful for a program to be supported by some documentation, however sketchy.

The written information should explain, of course, what the program does, then go on to outline the flow of action within the

program. The documentation should alert the user as to the kind of actions which will be required from him or her when running the program, and give an indication of the kinds of user input and reaction which will be accepted. The format of the final output should also be discussed.

If there are ways in which the program can be developed, extended or improved, suggestions along these lines can be added to the documentation. Written references to any material which will help in understanding the algorithms used, or for giving suggested areas for program development, should also be included.

In many ways, it is reasonable to assume that the job of programming is not finished once the program is done. Without documentation, the job is only three quarters complete. Documentation finishes the task, adding a professional stamp to your work which allows the program you've written to be used most effectively.

It is worth trying to write your program documentation so that it would make sense to someone who has not seen the program running. This person should be able to get a very good idea, just from reading the documentation, of what the program does and how it interacts with the user both in terms of accepting information and in presenting the results of its computations to the users; and how the program is organised as a whole.

Documentation for a major program should start with an introduction which quickly explains what is going on and tells how to use the program. The later parts of the documentation can then discuss the program in greater detail. It is not good practice to force the user to wade through a vast amount of information in order to dig out the vital facts he or she needs to get the program running.

Chapter Twenty-One — The FORTH Engine

You can, if you like, buy a full implementation of the language FORTH for the QL (from Computer One, Science Park, Milton Road, Cambridge, Ø223 - 862616). If, however, you'd like just a little taste of some aspects of FORTH, you can use the program given in this chapter, which allows you to experiment with a 'mini-FORTH'.

FORTH was developed by Charles Henry Moore, who was reportedly frustrated by the ability of the programming languages he was currently using (such as FORTRAN and Algol) to allow him to write programs to control radio telescopes. Mr Moore first developed a primitive version of his language in the early seventies, and finally got an integrated version of the language running on the IBM 113Ø, the most powerful computer he had at his disposal. The language was called FORTH because Moore was working with 'third generation' computers, and saw his language as a 'fourth generation' one. However, as serendipity would have it, the 113Ø only allowed file names of five characters, so the word 'fourth' was shortened to FORTH. As Moore points out in the introduction to *Starting FORTH* (FORTH, Inc., Leo Brodie, Prentice-Hall, 1981), this was just as well because when compared with 'fourth', FORTH was "a nicer play on words anyway".

The best thing about FORTH is that it comes equipped with a set of standard commands, which you use to develop your own commands, which can in turn be used in the construction of further other commands. This makes it fast, powerful and flexible.

One of the most unusual things about FORTH is its arithmetic, the aspect of FORTH the first program in this chapter allows you to emulate. When you want to add 5Ø and 9 on the QL normally, you enter the following:

```
PRINT 50 + 9
```

To do this in FORTH you need to enter:

```
50 9 + .
```

You enter the first number, then a space, then a second number, then another space, then the operation you wish to perform (in this case '+' for addition) followed by a "." which causes the program to print out the answer. If you leave off the ".", the computer will still work out the answer, but will not give it out to you.

A FORTH computer is "stack-orientated". A *stack* is like a tall pile of pieces of paper. You can write on a piece of paper and put it on top of the pile, or can take a piece away, but you can only remove a sheet from the top of the pile. The last sheet you put on the pile must be the first one you take off.

When you enter the 50 followed by a space, a FORTH computer, in effect, writes it on a sheet of paper and places it on top of the stack of sheets of paper. Then, it writes the 9 on another sheet and places it on top of the 50, so the 50 is now the second piece of paper in the stack. Next the computer comes to the instruction "+" which is a word which is in its "dictionary" of things it knows how to do. "+" tells a FORTH computer to "take the top two numbers off the top of the stack, add them together, then place the result of that addition on top of the stack". This is what our "50 3 + ." program does. Finally, the computer sees the "." and uses this as a signal to "pop the value off the top of the stack" or – as in this case – print it out. If you do not include a "." the value will stay on top of the stack. You can prove this by entering a simple bit of arithmetic, but leaving off the "." to see what happens. You stop the program (by pressing the ENTER key without entering a new bit of arithmetic) and then ask the QL, in direct mode to:

```
PRINT stack
```

When you do this, it will print out the result which is on the stack.

This explanation may seem a bit bewildering, but it should become clearer to you as you run our program. Here, first of all, are some sample runs:

```
? 3 2 + .
  Stack=5

? -3 -2 + .
  Stack=-5

? 3 2 - .
  Stack=1

? -3 -2 - .
  Stack=-1

? 3 -2 -

? 3 2 * .
  Stack=6

? 7 4 / .
  Stack=1

? 4 7 / .
  Stack=0
```

```
? 3 2 + .
  Stack=5

? -3 -2 + .
  Stack=-5

? 3 2 - .
  Stack=1

? -3 -2 - .
  Stack=-1

? 3 -2 -

? 3 2 * .
  Stack=6

? 7 4 / .
  Stack=1

? 4 7 / .
  Stack=1
```

```
? 77 9 / .          ? 77 9 / .

  Stack=8             Stack=8


? 0 9 / .           ? 0 9 / .

  Stack=0             Stack=0


? 9 0 / .           ? 9 0 / .
Division by zero is Division by zero is
illegal             illegal

? 8 8 * .           ? 8 8 * .

  Stack=64            Stack=64


? k                 ? k
```

And this is the program so you can experiment with Reverse Polish Notation for yourself:

```
100 REMark     'Forth Arithmetic'
110 REMark (Reverse Polish Notation)
120 :
130 PAPER 1:INK 7
140 CLS:CLS #0
150 stack=0
160 :
170 REPeat forth
180  PRINT \\
190  INPUT " ? ";f$
200  IF LEN(f$)<2 THEN EXIT forth
210  f$=" " & f$
220  q=LEN(f$)
230  output=0
240 :
250  IF f$(q)="." THEN
260   output=1
```

```
270    f$=f$(1 TO q-2)
280    q=q-2
290   END IF
300  :
310   j=q
320   IF f$(j)="*" OR f$(j)="/" OR f$(
j)="+" OR f$(j)="-" THEN
330   flag=(f$(j)="*")+2*(f$(j)="/")+3
*(f$(j)="+")+4*(f$(j)="-")
340    x=j-1:Z=j-2
350    REPeat extraction
360     x=x-1
370     IF f$(x)=" " THEN EXIT extract
ion
380    END REPeat extraction
390    second$=f$(x+1 TO Z)
400   END IF
410  :
420   f$=f$(1 TO x-1)
430   q=LEN(f$)
440   Z=q
450  :
460   REPeat extraction2
470    q=q-1
480    IF f$(q)=" " THEN EXIT extracti
on2
490   END REPeat extraction2
500  :
510   first$=f$(q+1 TO Z)
520  :
530   IF second$="0" AND flag=2 THEN
540    PRINT " Division by zero is ill
egal"
550    END REPeat forth
560   END IF
570  :
580   IF flag=1 THEN stack=INT(INT(fir
st$)*INT(second$))
590   IF flag=2 THEN stack=INT(INT(fir
st$)/INT(second$))
600   IF flag=3 THEN stack=INT(first$+
```

```
second$)
610  IF flag=4 THEN stack=INT(first$-
second$)
620  IF output THEN PRINT \"    Stack=
";stack
630 :
640 END REPeat forth
```

FORTH works only with integers, which is why we use INT.

The Engine

Now while the 'FORTH arithmetic' program is OK as far as it goes, I soon realised that it was unnecessarily limited. Therefore, I set myself the task of writing a program which would allow you to emulate far more aspects of FORTH. Our next program, the *FORTH Engine,* is the result of that decision.

As we discussed earlier, FORTH is stack-orientated. This program has a stack which is 20 elements deep, although only the top 10 elements are actually printed on the screen. After each operation, the stack is reprinted. Beside it, to the left, is a copy of the top 10 elements of the stack as they stood before the current operation, so you can see for yourself just what effect each operation has. With the aid of the *FORTH Engine* and a book on FORTH (and you might find my book *Learn FORTH on your BASIC Micro* suitable for this), you can teach yourself a useful amount of FORTH, and can experiment with many elements of it.

FORTH operations basically come down to putting numbers on the stack ('pushing') or taking them off ('popping'). The *FORTH Engine* makes almost constant use of two procedures: push_stack and pull_stack. Another procedure, pop_stack, is used to take the top value off the stack and *print* it out on the screen.

The most exciting thing about FORTH is the ease with which new words can be defined. The FORTH word DUP duplicates the top number on the stack. The FORTH word . (dot) pops the top number off the stack and prints it out. Dot is destructive. That is, you *lose* the top number by popping it off. Therefore, if you used DUP before .

you could have the number printed out and still have it on the top of the stack to use. We could invent a word of our own – PRINTOUT – which duplicated the top number on the stack, then popped off the top number and printed it out. You create words by defining them as follows:

```
: PRINTOUT DUP . ;
```

You start with a colon (a word defined in this way is often called a *colon-definition*), follow it with a space (in FORTH spaces are vital), and then put the words (in this case DUP .) which you want as your definition, followed by a semi-colon to signal that the word has been defined. Then, from that point onwards, you could just use PRINTOUT in your FORTH programs, and the computer would automatically perform a DUP . for you. The *FORTH Engine* can handle 2Ø new words which you define (in fact, it can handle more than that with a few simple changes, but the primary purpose of this program is to allow you to experiment with FORTH more than to create a complete FORTH).

If you wanted to define a word which would square the number on top of the stack, then make a copy of the result, then print out the answer, you could define a word as follows:

```
: SQUARE DUP * DUP . ;
```

In this colon-definition, SQUARE is the name of the word you are defining. DUP duplicates the top number on the stack, then uses * to multiply it by itself. The answer is then on the top of the stack where DUP . copies the answer, then prints out the answer, leaving the copy of the answer on the stack.

Now we get to an even more exciting part of FORTH. You'll see that the last part of our definition of SQUARE (DUP .) is the same as the definition of PRINTOUT. Why don't we use PRINTOUT in our SQUARE definition? There is no reason at all why you should not do this. The real magic of FORTH is that you can keep adding more and more words to your *dictionary,* which are made up of words which you have previously defined. And because the QL saves variables when it saves a program, you can have your previously-defined words ready for access at any time. The program includes a *warm/cold* start option, with *warm* preserving the contents of your dictionary and the stack.

But, for now, back to our definition of SQUARE. You'll recall that PRINTOUT is DUP . and that SQUARE was DUP * DUP . which suggests we could redefine SQUARE as follows:

```
: SQUARE DUP * PRINTOUT ;
```

You can do this with the *FORTH Engine,* including one or two previously-defined words in a new word definition. If you want to change the definition of a word, you can use the command *FORGET name* to delete the word from the dictionary, before redefining it as you choose. (Although the *Engine* will forget the word you name, it will not forget other words which used that word as part of their definition.)

FORTH in Action

FORTH, then, works more or less like this:
— The user types something into the computer
— FORTH looks through its dictionary of words to see if any of the words match that which has been typed
— If it finds a match either within the dictionary provided with the program, or within the dictionary of words the user has defined, it executes the word, then goes to the next element in the input stream
— If it does not find a match, it assumes the word is a number

Our *FORTH Engine* works in a very similar way. It first scans the input for defined words, changing any defined words into their primitive elements. It then works along the input stream, processing it in order. If an element in the stream is neither a word nor a number, it ignores it (whereas most versions of FORTH give an error message). Any number which is found is pushed onto the stack, while any arithmetic operation or stack manipulation is carried out.

You'll see the progress of the *Engine* along the input stream by a red strip which appears over the material still to be processed. It works its way along the input stream, from left to right, as it is processed, so you can see what is going on.

Now I'm not going to attempt to teach you very much FORTH with this chapter. That task really demands a complete book, and suggestions for suitable books are included at the end of this chapter. Instead, I will go through the words which are provided with your *Engine* and explain very briefly what they do. You can easily experiment with them to use their potential to the full. Once you have the *Engine* in place on your QL, you'll find that experimenting with the various elements of the language, and creating others of your own, will soon demonstrate to you what an exciting language FORTH can be.

There are few restrictions on use of the Engine. Any name (up to 400 characters long) is valid for a *word*, with just two restrictions. Firstly, the name cannot contain a space. You can name new words with names of words already in the *supplied* dictionary, and you can combine symbols, numbers and characters in any combination (so 6gG@*h is a valid name, as is @ or even 6). If you're feeling devilish you can even redefine multiplication (*) to be division (/), or 7 as -184, although your computer would behave pretty strangely if you did. Secondly, you should not use the name of one defined word *within* another. That is, if you call one word TOTAL do not call another one SUBTOTAL, although SUBTOTA is perfectly acceptable, or even SUBtOTAL (the *FORTH Engine* distinguishes between upper and lower case for words you define yourself, although you can enter the 'supplied words' in upper or lower case (but not in a combination of the two).

The great usefulness of being able to define words can be demonstrated by the following. Suppose you want a word called CIRCLE which would take a number off the stack, and treat it as the diameter of the circle, printing out the circumference of the circle. Then you'd only need to enter something like 23 CIRCLE to get it to print out the circumference of a circle with a diameter of 23 units.

The word CIRCLE could be defined as follows (where 355/113 is being used as an integer approximation to PI):

```
: CIRCLE 355 113 */ . ." is the circu
mference" ;
```

Then, you'd only need to enter a number (or allow the computer to take one off the stack) for the diameter, and your *FORTH Engine*

would do the rest. You'll find you should have a lot of fun defining words, and using them, on the *Engine.*

There are two 'standard' versions of FORTH, fig-FORTH and FORTH-79. The *FORTH Engine* adheres to the FORTH-79 standard, but allows the use of fig-FORTH equivalents in some cases (so the fig-FORTH word MINUS can be used interchangeably with FORTH-79's NEGATE). Here is the vocabulary supplied with our program:

+	Takes the top two numbers off the stack, adds them, then pushes the result back on the stack.
—	Takes the top two numbers off the stack, subtracts the top number from the second, and puts the result back on the stack.
*	Takes the top two numbers off the stack, multiplies them and then pushes the result back on the stack.
/	Takes the top two numbers off the stack, divides the second number by the top number, pushes the quotient back on the stack.
MOD	Does a division like / but puts the remainder on the stack.
/MOD	Division again, putting the quotient *and* remainder on the stack, with the quotient on top.
*/	This does a 'multiply-then-divide' leaving the quotient. It needs three numbers from the stack.
***/MOD**	This performs like */, returning the quotient and remainder, with the quotient on top.
**	This takes two numbers from the stack, and raises the second number to the power of the first number on the stack, returning the result to the stack.
NEGATE	This is the FORTH-79 word which multiplies the

number on the top of the stack by minus one, changing its sign. The fig-FORTH equivalent is MINUS so our *Engine* caters for this as well (see line 83Ø).

ABS This performs just like ABS in BASIC, returning the absolute value of the number on top of the stack.

MAX and **MIN** This pair of words compares the top two numbers on the stack, leaving only the larger or smaller on the stack.

DUP This stands for 'duplicate' and makes a copy of the top number on the stack, pushing it onto the stack on top of the original number.

?DUP This works like DUP, but only if the stack is not empty. The fig-FORTH equivalent is -DUP so this is supplied.

OVER This takes the second number on the stack, copies it, and puts the copy on top of the stack. So if the top number was A and the second one was B, an OVER would make the stack read (from the top down) B A B.

PICK This must be preceded by a number, and selects the numbered element on the stack (so if the preceding number was 6 it would select the sixth element, counting down), copying it onto the top of the stack.

DROP This deletes the top number on the stack.

SWAP This causes the two top numbers on the stack to change places.

ROT ROT (for rotate) brings the third element on the stack up to the top, moving the former items one and two down a position each.

ROLL This, like PICK, is preceded by a number, and brings this numbered element to the top, deleting it from its original position, and moving all other elements down one position.

DEPTH This tells you how many numbers are on the stack. The *Engine* stack has a maximum depth of 2Ø.

VLIST This lists out the program's vocabulary, with user-defined words first, from the newest one to the oldest.

FORGET Must be followed by the name of a word you have defined. It then deletes this word from the dictionary, but *does not* (in contrast to real FORTH) destroy words which were defined after the 'forgotten' one.

." This is pronounced "dot-quote" and is used to preface text output. It must be followed by a space. The *Engine* regards as text all the material that follows it up to the next " which can follow the text *without* a space preceding it. This means *." HELLO"* would print out the message *HELLO.*

' This is pronounced "tick" and is used to find out if you've given a particular name to a word in the dictionary. You just enter *' name* and the program will respond with *name OK* if there is a word with that name in the dictionary.

KEY This works like INKEY$, waiting till you touch a key, then putting the ASCII code of the character on the top of the stack. Some FORTHs use INKEY as their command but I have not included this option. Please feel free to add it if you like.

EMIT This is the 'opposite' of KEY, printing out the character whose ASCII code precedes it, so 42 EMIT will print out an asterisk.

SPACES This can be used to format output, as it takes the top number off the stack, and prints out that number of spaces.

DO LOOP This is somewhat like a FOR/NEXT loop in BASIC. It repeats everything which comes between the words DO and LOOP. The number of repetitions is the difference between the top two numbers on the stack, where the second number must be higher than the first. That is, if the second number on the stack is 2Ø and the top number is 1Ø, the loop will be run through 1Ø times.

There are two additional words in the program's vocabulary which have been given meanings which are different from those they enjoy in FORTH-79.

LIST This is a FORTH word used in disk control. In the *Engine* LIST is followed by a user-defined word, and prints up the assigned meaning of the word, so you can check on a word if you have forgotten what meaning you gave to it.

SP! This is a fig-FORTH word which is used to initialise the stack pointer (which is not covered in this discussion). In the FORTH supplied with our program, SP! empties everything from the stack.

The following printouts should give you some idea of what the *Engine* does when it is up and running.

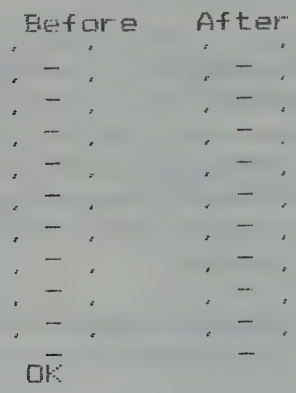
You are first given the option of starting from scratch, or retaining the current stack contents and defined words (this information is *not* lost by entering RUN on the QL):

```
Enter 1 for a warm start
      2 for a cold start

  2

Initialising  FORTH ENGINE
```

If you press "2", you are told the *Engine* is being initialised. Then the empty stack is printed up, along with the message "OK", which always indicates that the Engine is ready for some input:



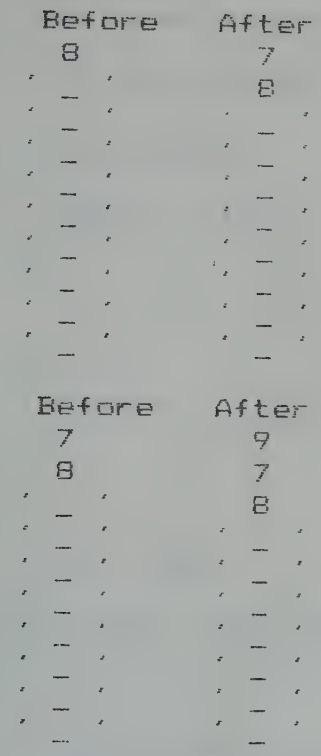
We enter a stream of commands, following the prompt "?", then the *Engine* reprints the input (after having changed any user-defined words back into their primitive elements). In this case, we enter four numbers, ask the *Engine* to perform a modular multiply then divide, then pop the top two numbers off the stack. You can see, by following through this series of printouts, how the stack behaves:

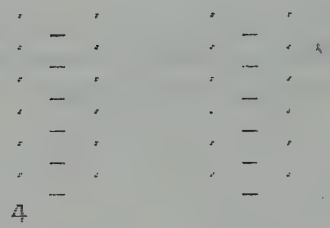
```
----------------------------------------

? 8 7 9 4 */mod . .

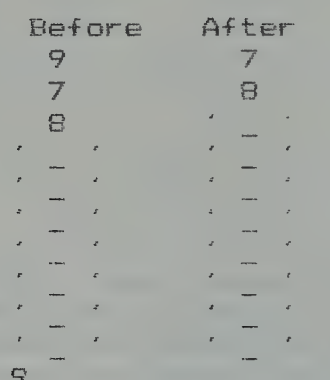
  8 7 9 4 */mod . .


 Before    After
                8
```

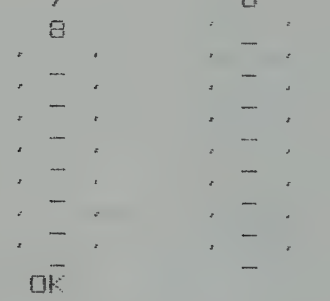

Before
After
8
7
8
Before
After
7
9
8
7
8
Before
After
9
4
7
9
8
7
8
Before
After
4
9
7
8
9
7
8



```
4
```



```
 Before    After
  9          7
  7          8
  8
9
```



```
 Before    After
  9          7
  7          8
  8
 OK
```

You can see that 7 and 8 are left on the stack. We enter a new set of commands. Firstly the computer is told to print out the words "now rotate", after which it will perform a ROT, a DUP and then pop the stack. Watch the ROT, DUP and pop in action:

```
? ." now rotate" rot dup .

  ." now rotate" rot dup .

now rotate

 Before    After
,   ,     ,   ,
, 5 ,       5
  5         7
  7         8
  8       ,   ,
,   ,     , 5 ,
, 5 ,     , 5 ,
, 5 ,     , 5 ,
, 5 ,     , 5 ,
, 5 ,     , 5 ,
, 5 ,     , 5 ,
  5         5
Stack empty
```

The ROT brings an empty element of the stack to the top. This is DUPlicated, and then popped, leading to the message "Stack empty".

Next we ask the *Engine* to print out "Rotate again" followed by a ROT, a SWAP, pushing 5 and 4 on to the stack, then multiplying them together:

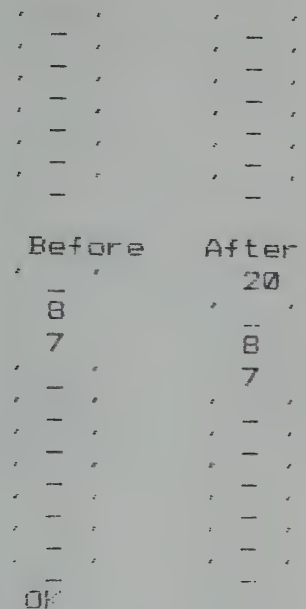
```
-----------------------------------
? ." Rotate again" rot swap 5 4 *
  ." Rotate again" rot swap 5 4 *
Rotate again
 Before    After
,   ,       5
  5       ,   ,
  8         5
  7         8
,   ,       7
, 5 ,     ,   ,
, 5 ,     , 5 ,
, 5 ,     , 5 ,
, 5 ,     , 5 ,
, 5 ,     , 5 ,
, 5 ,     , 5 ,
  5         5
```

```
 Before    After
   5         4
 .   .     .   .
 |   |     |   |
 |_  |     |   |
   8       |_  |
   7         8
 .   .     .   .
 |   |     |   |
 |_  |     |   |
 |   |     |   |
 |   |     |   |
 |   |     |   |
 |   |     |   |
 |_  |     |_  |
   _         _

 Before    After
   4         5
   5       .   .
 .   .     |   |
 |   |     |_  |
 |_  |       8
   8         7
   7       .   .
 .   .     |   |
 |   |     |   |
 |   |     |   |
 |   |     |   |
 |   |     |   |
 |   |     |   |
 |_  |     |_  |
   _         _

 Before    After
   5       .   .
 .   .     |   |
 |   |     |_  |
 |_  |       8
   8         7
   7       .   .
 .   .     |   |
 |   |     |   |
 |   |     |   |
 |   |     |   |
 |   |     |   |
 |   |     |   |
 |_  |     |_  |
   _         _

 Before    After
 .   .       20
 |_  |     .   .
   8       |   |
   7       |_  |
 .   .       8
 |_  |       7
```



We will now try defining a few words. First, we'll define *printout* which is, as discussed a little earlier, a DUP then a pop:

```
: printout dup . ;
```

We can use this word within the definition of another word. Notice how the *Engine* changes the elements of other defined words into their primitive forms before compiling the new word:

```
? : numbers 3 printout 2 printout ;

  : numbers 3 dup . 2 dup . ;
```

We test it by typing in *numbers* and watching the result:

```
? numbers

  3 dup . 2 dup .
```

```
 Before    After
   20        3
   20       20
 ,   ,      20
 |   |     ,   ,
   8        |   |
   7        8
 ,   ,      7
 |   |    ,   ,
 |   |    |   |
 |   |    |   |
 |   |    |   |
   |        |
```

```
 Before    After
   3        3
   3       20
   20      20
   20    ,   ,
 ,   ,   |   |
 |   |     8
   8       7
   7     ,   ,
 ,   ,   |   |
 |   |   |   |
 |   |   |   |
 |   |   |   |
3          |
```

```
 Before    After
   3        2
   20     . 3
   20       20
 ,   ,      20
 |   |    ,   ,
   8        |   |
   7        8
 ,   ,      7
 |   |    ,   ,
 |   |    |   |
 |   |    |   |
 |   |    |   |
   |        |
```

```
 Before    After
   2        2
   2        3
   3       20
   20      20
```

```
  20          .   .
.   .           _
  _             8
  8             7
  7           .
.   .       .   .
.   .       .   .
  _             _
2
```

```
 Before    After
  2          2
  2          3
  3          20
  20         20
  20        .   .
.   .         _
  _           8
  8           7
  7         .   .
.   .       .   .
.   .       .   .
  _           _
 OK
```

To check whether *numbers* and *printout* have really entered the program's vocabulary, we enter *vlist:*

```
? vlist

  vlist

numbers printout          +         -
   *         /          MOD       /MOD      */
*/MOD     **          ."         NEGATE   MINUS
   DEPTH     ABS        MIN        MAX       SW
AP
FORGET    VLIST     DUP       ?DUP      -DUP
   ROT       EMIT      SPACES    ROLL      OV
ER
'         DROP      PICK      LIST      KEY
   SP!       DO        LOOP
```

A Game (of sorts)

The words defined in this run are not of much use in and of themselves. I decided I'd try and be a bit more ambitious, and see if I could create a game. With such a limited language, with no facility for generating random numbers, and the only input to the *Engine* being via the word KEY, the games we can produce must be extremely limited. In this program, the *Engine* "thinks" of a number between zero and nine, and you have four guesses to work out what it is. (You are not supposed to look at the stack to see which number the *Engine* has on top.) You enter your guesses by touching one of the number keys on the QL, and the feedback from the *Engine* comes as a positive or negative number, or as a zero. The zero means you've guessed the number. This game, by the way, is almost totally useless as a game, but does show the potential of both FORTH and the *Engine.*

First we define a word *start:*

```
start   7 ." key " key 48 - ." key?"
key 48 - swap
```

This puts a seven on the stack, prints out the word *key,* accepts a key press, subtracts 48 from the ASCII code of that key (so the "1" key returns 49 which becomes 1 and so on), prints out *key?* and accepts another key press which is changed to a number. The two numbers are then transposed with SWAP.

The second word we need is *initialise:*

```
initialise      dup dup rot swap pick
```

With this the top number is duplicated twice, a rotation is performed, and then a PICK. This ensures that a 'random' number is now on top of the stack.

We have the starting conditions for the game in place. Each 'move' in the game is held in the word *play:*

```
play    dup ." now! " key 48 - - .
```

This duplicates the number on top of the stack, prints out the

message *now!* accepts a key press, subtracts 48 from this, then subtracts this number from the number on top of the stack. The number on top of the stack is a duplicate of the number the *Engine* is thinking of, so if the number popped is zero, the player has entered the same number.

All these words can be combined, with a DO LOOP, in a word called *game:*

```
: game start initialise 5 1 do play loop
```

When you compile *game,* it turns into this:

```
game    7 ." key " key 48 - ." key?" key
48 - swap dup dup rot swap pick 5 1 do
dup ." now! " key 48 - - . loop
```

This gives you four guesses at the number. You might like to 'turn off' the stack printing while you play this. Although this series of words does not, as I pointed out earlier, form very much of a game, it does demonstrate fairly clearly how words can be defined, and combined into new definitions. As well, it should give you a starting point from which you can develop your own games and similar routines.

Defining Words

Before we get to the listing of the *Engine* itself, let's look in a little more detail at the way to define your own words. There are four parts to the word. Imagine this was a defined word:

```
: STARS 42 emit 42 emit 1 1 . . ;
```

The first part of the word is the opening colon, which tells the *Engine* to go into 'compile mode'. After a space comes the *name* by which the *Engine* will subsequently recognise the word. Next is the actual definition. In this case the definition is *42 EMIT 42 EMIT 1 1. .* which would print two asterisks, push the number one on to the stack twice then pop the top two numbers off the stack. The final part of the definition is the semi-colon which tells the *Engine* the word definition is over. You cannot compile new words in the same input

stream as other actions, although you can mix user-defined words with the in-built vocabulary in the program's initial dictionary. You can also use your definitions within new definitions, so the following would be legal:

```
: GALAXY STARS STARS ;
```

You could subsequently get rid of the word stars with the command *FORGET STARS* and a check with *VLIST* would show it had vanished from the dictionary, although (unlike true FORTH) *GALAXY* would still work.

The *Engine* does not supply constants or variables, but they can be simulated simply by assigning a word to a value and using it as a number and then going through the FORGET/assign process to change its value. Therefore:

```
: testing 65 ;
```

would set the word *testing* to 65 and it could be used as if it was the number 65 from that point on. To change its value, you'd need to *FORGET testing* followed by: *testing newvalue;* to re-assign its value to *newvalue.* Note that when you *FORGET* a word, you do *not* get the memory space back again, so a maximum of 20 definitions can be made in any series of warm starts. If you try to create more than 20 words, the new word will simply override the twentieth word you defined.

Now, after all that, here's the *Engine* listing so you can start exploring FORTH on your QL:

```
100 REMark FORTH Engine
110 start_up
120 :
130 REPeat forth
140 valflag=0
150 print_stack
160 AT 18,0:INK 2:PAPER 6:PRINT " OK
":PAPER 0:INK 7
170 copy_stack
180 AT 12,0:PRINT "
```

```
                "
190 REPeat keyin
200  IF INKEY$="" THEN EXIT keyin
210 END REPeat keyin
220 a$=""
230 IF doflag=0 THEN
240  AT 12,0:INPUT "? ";a$
250 ELSE
260  FOR j=pdo TO qdo-1
270   a$=a$ & s$:BEEP 100,j
280  END FOR j
290 doflag=0
300 END IF
310 position=0
320 IF a$="" THEN EXIT forth
330 a$="    " & a$ & " !!! !!! "
340 :
350 IF nword>0 AND "forget" INSTR a$=
0 AND "FORGET" INSTR a$=0 AND "list"
INSTR a$=0 AND "LIST" INSTR a$=0 AND
"'" INSTR a$=0 THEN
360  j=0:p=0
370   REPeat definitions
380    j=j+1
390    IF j>nword THEN EXIT definitio
ns
400    p=LEN(w$(j))
410 :
420 FOR check=1 TO 7:BEEP 1,check
430    lencheck=LEN(a$)-p:FOR q=1 TO
LEN(a$)-p
440     IF a$(q TO q+p-1)= w$(j) THEN

450      a$=a$(1 TO q-1) & z$(j) & "
" & a$(q+p+1 TO)
460     END IF
470    END FOR q
480 END FOR check
490   END REPeat definitions
500 END IF
510 :
```

```
520 IF a$(1 TO 3)="!!!" THEN END REPe
at forth
530 strip_leading_space
540 :
550 AT 13,0
560 FOR j=1 TO 6
570  PRINT "
              "
580 END FOR j
590 copy$=a$(1 TO LEN(a$)-9)
600 AT 12,0:PRINT "   ";copy$;"

           "
610 :
620 lena=LEN(a$)
630 REPeat action
640 :
650 search
660 strip_leading_space
670 flag=0:flag=valflag
680 IF b$(1)=":" THEN
690  define_new_word
700  EXIT action
710 END IF
720 IF position>30 THEN AT 16,0:PRINT
 "
          ":position=0
730 kk=LEN(copy$)-LEN(a$)+9:IF kk<4 T
HEN kk=4
740 IF LEN(copy$)>2 THEN
750  AT 12,2:PRINT copy$(1 TO kk-2);
760  PAPER 2:PRINT copy$(kk-1 TO ):PA
PER 0
770 END IF
780 IF b$="do" OR b$="DO" THEN flag=1
:doflag=1:DOLOOP:END REPeat forth
790 IF b$="sp!" OR b$="SP!" THEN flag
=1:empty_stack
800 IF b$="+" OR b$="-" OR b$="*" OR
b$="/" OR b$="*/" OR b$="mod" OR b$="
MOD" OR b$="/mod" OR b$="/MOD" OR b$=
```

```
"**" THEN flag=1:operate
810 IF b$="key" OR b$="KEY" THEN flag
=1:key
820 IF b$='."' THEN flag=1:dotquote
830 IF b$="negate" OR b$="NEGATE" OR
b$="minus" OR b$="MINUS" THEN flag=1:
negate
840 IF b$="depth" OR b$="DEPTH" THEN
flag=1:depth
850 IF b$="'" THEN flag=1:tick
860 IF b$="abs" OR b$="ABS" THEN flag
=1:absolute
870 IF b$="min" OR b$="MIN" THEN flag
=1:min
880 IF b$="max" OR b$="MAX" THEN flag
=1:max
890 IF b$="swap" OR b$="SWAP" THEN fl
ag=1:swap
900 IF b$="forget" OR b$="FORGET" THE
N flag=1:forget
910 IF b$="vlist" OR b$="VLIST" THEN
flag=1:vlist
920 IF b$="dup" OR b$="DUP" OR b$="?d
up" OR b$="?DUP" OR b$="-dup" OR b$="
-DUP" THEN flag=1:dup
930 IF b$="rot" OR b$="ROT" THEN flag
=1:rot
940 IF b$="emit" OR b$="EMIT" THEN fl
ag=1:emit
950 IF b$="spaces" OR b$="SPACES" THE
N flag=1:spaces
960 IF b$="roll" OR b$="ROLL" THEN fl
ag=1:roll
970 IF b$="over" OR b$="OVER" THEN fl
ag=1:over1
980 IF b$="drop" OR b$="DROP" THEN fl
ag=1:drop
990 IF b$="pick" OR b$="PICK" THEN fl
ag=1:pick
1000 IF b$="list" OR b$="LIST" THEN f
lag=1:list1
```

```
1010 IF LEN(b$)=1 THEN IF b$="." THEN
 flag=1:pop_stack
1020 IF NOT flag  THEN IF CODE(b$)>44
 THEN IF CODE(b$)<58 THEN number=b$:p
ush_stack
1030  IF a$(1 TO 3)="!!!" THEN EXIT a
ction
1040 END REPeat action
1050 :
1060 IF a$="" THEN EXIT forth
1070 doflag=0
1080 END REPeat forth
1090 :
1100 STOP
1110 :
1120 :
1130 DEFine PROCedure push_stack
1140 copy_stack
1150 :
1160 REMark Move items down stack
1170 :
1180 FOR j=19 TO 1 STEP -1
1190  n(j+1)=n(j)
1200 END FOR j
1210 :
1220 REMark Push value on top
1230 n(1)=number
1240 print_stack
1250 END DEFine push_stack
1260 :
1270 DEFine PROCedure pull_stack
1280 copy_stack
1290 REMark Pull value from top
1300 stack_val=n(1)
1310 :
1320 REMark Now move rest
      of stack up one position
1330 :
1340 FOR j=2 TO 20
1350  n(j-1)=n(j)
1360 END FOR j
```

```
1370 n(20)=1E-31
1380 print_stack
1390 END DEFine pull_stack
1400 :
1410 DEFine PROCedure pop_stack
1420 pull_stack
1430 AT 16,position
1440 BEEP 100,4
1450 IF stack_val=1E-31 THEN
1460  PRINT "Stack empty":position=po
sition + 13
1470 ELSE
1480  PRINT stack_val:position=positi
on+LEN(stack_val)+1
1490 END IF
1500 END DEFine pop_stack
1510 :
1520 DEFine PROCedure operate
1530 LOCal p,q,r,s,t
1540 p=0:q=0:r=0:s=0:t=0
1550 pull_stack
1560 p=stack_val
1570 pull_stack
1580 q=stack_val
1590 IF b$(1 TO 2)="*/" THEN
1600  pull_stack
1610  t=stack_val
1620 END IF
1630 :
1640 IF b$="+" THEN r=q+p
1650 IF b$="-" THEN r=q-p
1660 IF b$="*" THEN r=q*p
1670 IF (b$(1)="/" OR b$="mod" OR b$=
"MOD") AND p=0 THEN AT 12,0:PRINT "\*
 division by zero illegal */":PAUSE 2
50:AT 12,0:PRINT "
                ":a$="!!!":END DEFine
operate
1680 IF b$="/" THEN r=INT(q/p)
1690 IF b$="mod" OR b$="MOD" THEN r=q
 MOD p
```

```
1700 IF b$="/mod" OR b$="/MOD" THEN s
= q MOD p:r=INT(q/p)
1710 IF b$="*/" THEN r=INT(q*t/p)
1720 IF b$="*/mod" OR b$="*/MOD" THEN
 r=INT(q*t/p):s=q*t MOD p
1730 IF b$="**" THEN r=q^p
1740 IF s<>0 THEN number=s:push_stack
1750 number=r:push_stack
1760 END DEFine operate
1770 :
1780 DEFine PROCedure search
1790  LOCal x
1800  x=0
1810 REPeat searchee
1820  x=x+1
1830  IF a$(x)=" " THEN EXIT searchee
1840 END REPeat searchee
1850 :
1860 b$=a$(1 TO x-1)
1870 IF a$(1 TO 3)="!!!" THEN END DEF
ine search
1880 a$=a$(x+1 TO)
1890 REMark
             We now have a section
             of string to process

     :
1900 END DEFine search
1910 :
1920 DEFine PROCedure empty_stack
1930 FOR j=1 TO 20
1940  n(j)=1E-31:c(j)=1E-31
1950 END FOR j
1960 END DEFine empty_stack
1970 :
1980 DEFine PROCedure print_stack
1990 AT 0,0
2000 PRINT " Before","  After"
2010 FOR j=1 TO 10
2020  IF c(j)=1E-31 THEN
2030   PRINT "  _  ",
```

```
2040   ELSE
2050   PRINT "  ";c(j),
2060  END IF
2070  IF n(j)=1E-31 THEN
2080   PRINT "   '  _  '      "
2090   ELSE
2100   PRINT "    ";n(j);"      "
2110  END IF
2120 END FOR j
2130 END DEFine print_stack
2140 :
2150 DEFine PROCedure copy_stack
2160 FOR j=1 TO 20
2170  c(j)=n(j)
2180 END FOR j
2190 END DEFine copy_stack
2200 :
2210 DEFine PROCedure swap
2220 LOCal s
2230 s=n(1)
2240 n(1)=n(2)
2250 n(2)=s
2260 END DEFine swap
2270 :
2280 DEFine PROCedure forget
2290 search
2300 FOR j=nword TO 1 STEP -1
2310  IF w$(j)=b$ THEN
2320     w$(j)=forget$
2330     z$(j)=forget$
2340     b$=""
2350  END IF
2360 END FOR j
2370 END DEFine forget
2380 :
2390 DEFine PROCedure vlist
2400 CLS
2410  FOR j=nword TO 1 STEP -1
2420   IF w$(j)<>forget$ THEN PRINT w
$(j),
2430  END FOR j
```

```
2440 PRINT "+","-","*","/","MOD","/MO
D","*/","*/MOD","**",
2450 PRINT '."',"NEGATE","MINUS","DEP
TH","ABS","MIN","MAX","SWAP","FORGET"
,
2460 PRINT "VLIST","DUP","?DUP","-DUP
","ROT","EMIT","SPACES","ROLL","OVER"
,
2470 PRINT "'","DROP","PICK","LIST",
2480 PRINT "KEY","SP!","DO","LOOP"
2490 PAUSE 250:CLS:print_stack
2500 END DEFine vlist
2510 :
2520 DEFine PROCedure dup
2530 IF (b$="?dup" OR b$="?DUP" OR b$
="-dup" OR b$="-DUP") AND n(1)=1E-31:
AT 16,position:PRINT "Stack empty":po
sition=position+12:END DEFine dup
2540 IF (b$="?dup" OR b$="?DUP" OR b$
="-dup" OR b$="-DUP") AND n(1)=0:AT 1
6,position:PRINT "Zero on stack":posi
tion=position+14:END DEFine dup
2550 FOR j=1 TO 19
2560  t(j)=n(j)
2570 END FOR j
2580 n(2)=n(1)
2590 FOR j=3 TO 20
2600  n(j)=t(j-1)
2610 END FOR j
2620 END DEFine dup
2630 :
2640 DEFine PROCedure rot
2650 LOCal s
2660 s=n(3)
2670 n(3)=n(2)
2680 n(2)=n(1)
2690 n(1)=s
2700 END DEFine rot
2710 :
2720 DEFine PROCedure emit
2730 pull_stack
```

```
2740 AT 16,position
2750 PRINT CHR$(stack_val)
2760 position=position+1
2770 END DEFine emit
2780 :
2790 DEFine PROCedure spaces
2800 pull_stack
2810 AT 16,position
2820 PRINT FILL$(" ",stack_val)
2830 position=position+stack_val
2840 END DEFine spaces
2850 :
2860 DEFine PROCedure roll
2870 LOCal s
2880 pull_stack
2890 xx=stack_val
2900 s=n(xx)
2910 FOR j=s TO 2 STEP -1
2920  n(j)=n(j-1)
2930 END FOR j
2940 n(1)=s
2950 END DEFine roll
2960 :
2970 DEFine PROCedure over1
2980 FOR j=19 TO 3 STEP -1
2990  n(j+1)=n(j)
3000 END FOR j
3010 n(3)=n(2):n(2)=n(1):n(1)=n(3)
3020 END DEFine over1
3030 :
3040 DEFine PROCedure drop
3050 FOR j=1 TO 19
3060  n(j)=n(j+1)
3070 END FOR j
3080 n(20)=0
3090 END DEFine drop
3100 :
3110 DEFine PROCedure pick
3120 LOCal x
3130 pull_stack
3140 x=stack_val
```

```
3150 FOR j=19 TO 1 STEP -1
3160  n(j+1)=n(j)
3170 END FOR j
3180 n(1)=n(x+1)
3190 END DEFine pick
3200 :
3210 DEFine PROCedure list1
3220 search
3230 IF nword=0:AT 12,0:PRINT " \* no
 words user-defined */":PAUSE 50:END
DEFine list1
3240 FOR j=1 TO nword
3250  IF w$(j)=b$ THEN PRINT \w$(j);"
 ",z$(j)
3260 END FOR j
3270 PAUSE 250
3280 print_stack
3290 END DEFine list1
3300 :
3310 DEFine PROCedure define_new_word
3320  BEEP 100,4
3330 nword=nword+1:IF nword>20 THEN n
word=20
3340 search
3350 w$(nword)=b$
3360 z$(nword)=""
3370 REPeat meaning
3380  search
3390  IF b$=";" THEN EXIT meaning
3400  z$(nword)=z$(nword) & " " & b$
3410 END REPeat meaning
3420 z$(nword)=z$(nword)(2 TO LEN(z$(
nword)))
3430 END DEFine define_new_word
3440 :
3450 DEFine PROCedure strip_leading_s
pace
3460  REPeat check
3470   IF a$(1)<>" " THEN EXIT check
3480   a$=a$(2 TO)
3490  END REPeat check
```

```
3500 END DEFine strip_leading_space
3510 :
3520 DEFine PROCedure negate
3530  pull_stack
3540  number=-stack_val
3550  push_stack
3560 END DEFine negate
3570 :
3580 DEFine PROCedure absolute
3590  pull_stack
3600  number=ABS(stack_val)
3610  push_stack
3620 END DEFine absolute
3630 :
3640 DEFine PROCedure min
3650  LOCal p,q
3660  pull_stack
3670  p=stack_val
3680  pull_stack
3690  q=stack_val
3700   IF q<p THEN
3710     number=q
3720   ELSE
3730     number=p
3740   END IF
3750  push_stack
3760 END DEFine min
3770 :
3780 DEFine PROCedure max
3790  LOCal p,q
3800  pull_stack
3810  p=stack_val
3820  pull_stack
3830  q=stack_val
3840   IF q>p THEN
3850     number=q
3860   ELSE
3870     number=p
3880   END IF
3890  push_stack
3900 END DEFine max
```

```
3910 :
3920 DEFine PROCedure depth
3930 LOCal m:m=0
3940 REPeat count
3950  m=m+1
3960  IF n(m)=1E-31 THEN number=m-1:E
XIT count
3970 END REPeat count
3980 push_stack
3990 END DEFine depth
4000 :
4010 DEFine PROCedure key
4020  key$=""
4030 REPeat clearing
4040  IF INKEY$="" THEN EXIT clearing
4050 END REPeat clearing
4060 REPeat get_key_press
4070  key$=INKEY$
4080  IF key$<>"" THEN EXIT get_key_p
ress
4090 END REPeat get_key_press
4100  number=CODE(key$)
4110  push_stack
4120 END DEFine key
4130 :
4140 DEFine PROCedure dotquote
4150 AT 16,position
4160 REPeat printing
4170  search
4180  IF b$(LEN(b$))='"' THEN EXIT pr
inting
4190  PRINT b$;" ";
4200  position=position+LEN(b$)+1
4210 END REPeat printing
4220 PRINT b$(1 TO LEN(b$)-1)
4230 position=position+LEN(b$)-1
4240 END DEFine dotquote
4250 :
4260 DEFine PROCedure tick
4270 search
4280 yesflag=0
```

```
4290 FOR j=1 TO nword
4300  IF w$(j)=b$ THEN yesflag=1
4310 END FOR j
4320 IF yesflag=1 THEN
4330  AT 16,position
4340  PRINT b$;" OK"
4350 END IF
4360 END DEFine tick
4370 :
4380 DEFine PROCedure DOLOOP
4390 s$=""
4400 pull_stack
4410 pdo=stack_val
4420 pull_stack
4430 qdo=stack_val
4440 IF pdo>=qdo THEN AT 16,position:
PRINT "Illegal DO LOOP values":doflag
=0:END DEFine DOLOOP
4450 REPeat loop_tasks
4460  IF b$="loop" OR b$="LOOP" THEN
EXIT loop_tasks
4470  s$=s$ & " " & b$
4480  search
4490 END REPeat loop_tasks
4500 s$=s$(5 TO) & " "
4510 END DEFine DOLOOP
4520 :
4530 DEFine PROCedure start_up
4540 CLS:CLS #0
4550 AT 10,0
4560 PRINT "Enter 1 for a warm start
                 2 for a cold start"
4570 REPeat yeti
4580  IF INKEY$="" THEN EXIT yeti
4590 END REPeat yeti
4600 REPeat yeti2
4610  k$=INKEY$
4620  IF k$="1" OR k$="2" THEN EXIT y
eti2
4630 END REPeat yeti2
4640 CLS
```

```
4650 IF k$="2" THEN initialise
4660 END DEFine start_up
4670 :
4680 :
4690 DEFine PROCedure initialise
4700 PAPER 2:INK 6:BORDER 0
4710 CSIZE 3,1
4720 CLS: CLS #0
4730 FLASH 1:AT 5,1:PRINT "Initialisi
ng  FORTH ENGINE":FLASH 0
4740 CSIZE 0,0
4750 REMark n - numbers
4760 REMark c - to copy stack
4770 REMark t - temporary store
4780 REMark w$ - names of new words
4790 REMark z$ - meaning of new words
4800 DIM n(20),c(20),t(20),w$(20,400)
,z$(20,400)
4810 stack=0
4820 a$="":b$=""
4830 number=0:stack_val=0:s=0:doflag=
0
4840 nword=0:REMark counts new words
4850 forget$=FILL$("*",10)
4860 FOR j=1 TO 20
4870  w$(j)=forget$
4880  z$(j)=forget$
4890 END FOR j
4900 REPeat clearing
4910  IF INKEY$="" THEN EXIT clearing
4920  BEEP 100,j:PAUSE 1
4930 END REPeat clearing
4940 PAPER 0:INK 7:CLS
4950 empty_stack
4960 END DEFine initialise
4970 :
```

If you want to investigate FORTH further, here are some books which may be helpful:

BASIC and FORTH in Parallel – S J Wainwright (Babani, 0 85934 113 5, £1.95).

Discover FORTH – Thom Hogan (McGraw-Hill, Ø 931988 79 9, £11.25).

Starting FORTH – Leo Brodie (Prentice-Hall, Ø 13 842922 7, £18.4Ø. This is possibly the best book on FORTH for first-timers I've seen).

The Complete FORTH – Alan Winfield (Sigma, Ø 9Ø51Ø4 226, £6.95).

Exploring FORTH – Owen Bishop (Granada, Ø 246 121882, £6.95).

Learn FORTH on your BASIC Micro – Tim Hartnell and Paul Kail (Interface Publications Ltd., 1984, £5.95).

FORTH is a trademark of FORTH, Inc., 23Ø9 Pacific Coast Highway, Hermosa Beach, California, 9Ø254, USA.

Chapter Twenty-Two — Simulating Reality

Simulations are ways of producing a counterfeit reality. In contrast to adventures (which we'll be looking at in the following chapter) where the 'reality' created by the computer is often magical and dream-like, the worlds accessed through computer simulations are generally more down to earth.

In a simulation, the computer manipulates variables in accordance with formulae you've specified, keeping tabs on the unfolding of the situation you've generated, and taking the place of the environment in terms of reacting to your input.

Simulations attempt to replicate life. However, because reality is notoriously hard to pin down and limit, we need to do some pretty drastic simplification before we can produce a workable simulation. Despite this simplification, as you'll soon see, a well-written simulation can imitate life to an uncanny extent. Cause and effect are linked as accurately as you can devise formulae for them. The random number generator on the QL can take the place of such things as changes in the weather, in population numbers, in the behaviour of molecules in a gas, or in the way a coloured ink is dispersed through a clear liquid.

In this chapter, we'll only look at a single simulation. However, it is a big program, and one which both poses a number of challenging problems and also indicates ways in which you can develop your own QL simulations.

The simulation program is *Big Time Operator* in which you have to run a factory which can produce a variety of objects (but only one kind of object each time the program is played) from bicycles through to ZX Spectrums and QLs. You are the boss of the factory.

You have almost total control of your staff, their output and your selling price. Your task is to stay in business for as long as possible, or until you're worth $10,000.

In this program, which explains itself as it runs, you are in charge of the factory. At the start of the run, you're told how many people you have on your staff, how much they are paid each week, how much capital you have on hand, how much stock is in your store room, and the selling price of the objects you're making. The program tells you your total wage bill, and this is the one figure you must always keep in mind. The program works in weeks, and you must manage to meet your wage and raw material commitments each week, or you'll be declared bankrupt.

Bankruptcy

The program can end in one of two ways. The most usual is bankruptcy, when you'll be told how many weeks you managed to keep your factory going. The second way the program can end is if you manage to build up capital of $10,000. As you'll discover as you endure this simulation, the cards are stacked against you. There is a saying in business life that the first job a business must do is ensure its own survival. You'll discover how trying this is when you try to run your QL factory. And despite the great simplification which has occurred to make this simulation manageable, it replicates reality in quite an uncanny way. You'll find yourself actually despairing when some sales or production figures come in, and will agonize over your hiring/firing and price-setting decisions.

The program looks pretty confusing when you glance through it. But do not worry. Although you are called upon to make a great number of decisions, your QL will prompt you at every step, leading you through the management decisions.

Time now to enter the program and experience for yourself the delights of being a *Big Time Operator:*

```
10 REMark     BIG-TIME OPERATOR
20     initialise
30 REPeat game_cycle
```

```
40     we=we+1
50     report
60     staff
70     report
80     production
90     report
100    sales
110    problems
120    ca=ca-wa*wo-rc
130 END REPeat game_cycle
140 REMark ------------
150 DEFine PROCedure problems
160 CLS
170 IF RND >.45 THEN
180  a=RND(1 TO 7)
190  PRINT\\\"   The unions are deman
ding a"
200  PRINT "    pay rise of ";a;"%"
210  wa=INT(100*(wa+wa*a/100))/100
220  delay
230  PRINT \" Pay per employee is now
 $";wa
240  delay
250  CLS
260 END IF
270 IF RND>event THEN
280  event=.81
290  PRINT\\\"  A fire in your wareho
use has"
300  PRINT "  destroyed some stock. P
lease"
310  PRINT "  stand by for a report o
n the"
320  PRINT "        damage caused.
.."
330  delay
340  a=INT(RND*st/2)+1
350  st=st-a
360  PRINT \"  There were ";a;object$
;" destroyed"
370  PRINT "  They were worth $";a*sp
```

```
;" retail"
380  delay
390  PRINT " Stock in hand is now ";s
t;object$
400  delay
410 END IF
420 IF RND<.3 THEN
430  CLS
440  PRINT \\\" Your main supplier ha
s announced"
450  PRINT "   a dramatic price rise.
.."
460  delay
470  a=INT(RND*100*CO/7)/100
480  IF a<1E-2 THEN GO TO 470
490  PRINT \" The cost of making";obj
ect$
500  PRINT "   has risen by $";a;" ea
ch"
510  delay
520  CO=CO+a
530  PRINT \"  It now costs $";CO
540  PRINT "      for each one"
550  delay
560 END IF
570 IF RND<.65 AND ma<sp THEN END DEF
ine problems
580 CLS
590 PRINT \\\" You have the chance to
 raise"
600 PRINT "   your price. Your";objec
t$
610 PRINT "      now sell for $";sp
620  delay
630 PRINT \" What percentage increase
 would"
640 PRINT "     you like to impose?
                    ";
650 INPUT a
660 re=re+a
670 sp=INT(100*(sp+a*sp/100))/100
```

```
680  delay
690 PRINT \\"The";object$;" now sell
for $";sp
700  delay
710 END DEFine problems
720 REMark ------------------
730 DEFine PROCedure sales
740  PRINT \"  Your total stock of";o
bject$
750  PRINT "        is now ";st
760    delay
770  FLASH 1:PAPER 0:INK 7
780  PRINT \\" Please stand by for a
sales report"
790    delay
800    FOR slide=1 TO 24
810     PRINT "
                "
820    END FOR slide
830  FLASH 0
840    FOR slide=0 TO 18
850     PAPER RND(1 TO 7):AT slide,0
860     PRINT "
                  "
870    PAUSE 4
880    END FOR slide
890  PAPER 2:INK 6
900  IF a>st THEN GO TO 830
910  CLS
920  PRINT \\\" The total number of";
object$
930  PRINT "      sold was ";a
940  st=st-a
950  za=a*sp
960  PRINT \"  Income from that sale
was $";za
970  ca=INT(a*sp*100)/100+ca
980   delay:PAUSE 100
990 END DEFine sales
1000 REMark --------------------
1010 DEFine PROCedure report
```

```
1020  CLS
1030  IF ca+st<1 THEN bankrupt:STOP
1040  IF ca+st>9999 THEN success:STOP
1050   PAPER 7:INK 2
1060  PRINT \"Shop floor report, sir,
 for week ";we;" "
1070   PAPER 2:INK 7
1080  PRINT \"  Capital in hand is $"
;INT(ca*100)/100
1090  PRINT "  Running costs are $";r
c;" a week"
1100  PRINT \"  Your stores hold ";st
;object$
1110  PRINT "             worth $";INT
(st*sp*100)/100
1120  PRINT \" They sell for $";sp;"
each, and"
1130  PRINT "       cost $";CO;" each
to make"
1140  delay
1150  PRINT \"  Your workforce is ";w
o;" strong,"
1160  PRINT "    and you pay them $";
wa;" each"
1170  PRINT "  a week, so the wages b
ill this"
1180  PRINT "         week is $";wa*wo
1190   delay
1200  PRINT \"     Each person can ma
ke ";pr
1210  PRINT "   ";object$;", a total
of ";pr*wo
1220 END DEFine report
1230 REMark --------------
1240 DEFine PROCedure production
1250  PRINT \"  How many do you wish
to produce?                  ";
1260  INPUT ma
1270  IF ma=0 THEN END DEFine product
ion
1280  PRINT
```

```
1290    IF ma*CO>ca THEN
1300     PRINT "  You do not have enou
gh money"
1310     delay:PRINT "                ";
1320     GO TO 1260
1330    END IF
1340    IF ma>pr*wo THEN
1350     PRINT "   You do not have eno
ugh people           in your workforce
"
1360     delay
1370     PRINT "                      ";
1380     GO TO 1260
1390    END IF
1400   PRINT "  Yes sir...the target f
or week ";we
1410   PRINT "              is ";ma;obje
ct$
1420   ma=ma-INT(RND*ma/5+1)
1430   delay
1440   PRINT \"       The number of";ob
ject$
1450   PRINT "    actually produced in
 week ";we
1460   PRINT "               was ";ma
1470   st=st+ma
1480   ca=ca-CO*ma
1490   delay
1500 END DEFine production
1510 REMark ------------
1520 DEFine PROCedure staff
1530  a=0
1540  PRINT \"   How many people do y
ou"
1550  PRINT "        want to hire?
                     ";
1560  INPUT a
1570  IF a=0 THEN GO TO 1630
1580  wo=wo+a
1590  PRINT \"  The total workforce i
s now ";wo
```

```
1600  delay
1610 END DEFine staff
1620   report
1630  PRINT \"How many people do you
wish to fire?                    ";
1640  INPUT a
1650  IF a=0 THEN
1660   delay
1670   END DEFine staff
1680  END IF
1690  IF a>wo THEN GO TO 1640
1700  a=RND(1 TO a)
1710  delay
1720  PRINT \"The unions will allow y
ou to get              rid of ";a
1730  wo=wo-a
1740  delay
1750 END DEFine staff
1760 REMark -------------
1770 DEFine PROCedure bankrupt
1780  PRINT \"        You are bankrupt
!!"
1790  delay
1800  PRINT \"        Oh the shame of
it!"
1810  delay
1820  PRINT \"    Still you kept the
business             going for ";we;"
 weeks"
1830 END DEFine bankrupt
1840 REMark ------------------
1850 DEFine PROCedure initialise
1860  RANDOMISE
1870  RESTORE
1880  INK 7:PAPER 2:BORDER 1,6
1890  CLS:CLS #0
1900   FOR z=1 TO RND(1 TO 10)
1910     READ object$
1920   END FOR z
1930  event=.61
1940  ca=RND(500 TO 999)
```

```
1950  st=RND(100 TO 599)
1960  sp=RND(10 TO 14)
1970  CO=RND(7 TO 11)
1980  IF CO>sp THEN GO TO 1970
1990  wo=RND(7 TO 16)
2000  wa=12+INT(RND*sp)
2010  pr=RND(5 TO 10)
2020  rc=RND(100 TO 119)
2030  we=0
2040  re=1:
REMark re = 'sales resistance factor'
2050 END DEFine initialise
2060 REMark --------------
2070 DEFine PROCedure delay
2080  BEEP 32000,RND(40 TO 60),RND(1
TO 5),RND(1 TO 5),RND(1 TO 5),RND(1 T
O 5),RND(1 TO 5),-22286
2090  FOR wait_around=1 TO 700
2100  END FOR wait_around
2110 END DEFine delay
2120 REMark -------------
2130 DEFine PROCedure success
2140  PRINT \\\" You've made $10,000
and can"
2150  PRINT "          now retire"
2160 END DEFine success
2170 REMark --------------------
2180 DATA " eponyms"," Sinclair QLs",
" bicycles"," ZX Spectrums"
2190 DATA " harmonicas"," kazoos"," l
ecterns"," pith helmets"
2200 DATA " ZX80's"," barbells"
```

Chapter Twenty-Three — Creating and Playing Adventures

Let's face it. Life can be pretty tame, sometimes. There don't seem to be many dragons waiting to be slain in my city, and chests heaped high with abandoned gold are in scarce supply. I can't remember the last time I met an Evil Magician down at the local supermarket, and it's been ages since I discussed battle tactics with sentient androids at the local tavern.

The hunger for excitement lies in all of us. The desire to take on the personalities of other, more vibrant, people – even for just an hour or so – is a common one. Although you can't conjure up devils and werebears, envoke the power of a Shield of Protection or employ trolls to carry sacks of emeralds from the ruins of an abandoned castle, role-playing games allow you to do just that.

Adventure gaming has hit the big time. You've probably seen the claims that it is the 'fastest growing game in the world'. Whether that's true or not, it indicates that adventure gaming is a leisure pursuit which satisfies the inner needs of many people.

You may well have taken part in adventure role-playing games yourself.

But these real-life campaigns have one enormous disadvantage. You need people to play with and against. You need a referee (often called the Game Master, or Dungeon Master) to control the world and its artifacts and encounters. It is not always particularly easy to get all these other human beings together just when you decide you'd like to indulge in a little bit of adventuring. That's where the QL can come in.

Unpredictable

Although QL adventure games lack a little of the spontaneity of such games when played with live company, they can be remarkably unpredictable and exciting to play. The fact that the dreaded *Uncle Clive Monster* exists only within the QL's RAM seems in no way to diminish the relief you feel when it dies. The gems you find lying all over the place are no less 'real' than those discovered in live-action adventures.

The word adventure, then, is used to describe the class of computer games in which the player moves through an alternative reality. In this "otherworld" there are monsters to be fought, treasures to be discovered, maps to be made, and puzzles to be solved.

There are two adventure-type programs in this chapter. Both are long, major games, and should have you working for hours trying to solve them.

One feature of true adventure games is that the reality they model is *consistent*. That is, the world created within the adventure program is solid, and – apart from any events specific to that game, such as an earthquake, or a magic spell – the parts of the world do not shift in a random fashion. In a properly constructed adventure the rivers stay in place, dungeon walls do not mysteriously shift and move every time you turn your back, and objects you put down in one cave within an underground labyrinth do not suddenly appear of their own volition a hundred leagues away.

Map-making is one of the true adventure-player's skills and delights. Working your way through an imaginary, but self-consistent environment, tackling monsters and collecting treasures, solving puzzles as you go up and down staircases and chutes, exploring side tunnels, getting lost in self-circling mazes and so on, is only fascinating if the world you are exploring is mappable. Both the "worlds" created in this chapter of the book can be mapped, and the first one even gives you glimpses from time to time of the map when seen from above.

Our first program is *The Marquis of Kwue-El*. This is a "grid adventure". In other words, the "map of the world" is based on a

grid, and you move up (north), down (south), right (east) and left (west) on the grid. This world is populated with a range of fearsome possibilities, including pits containing quicksand or dragons, as well as magic caves which can transport you at random within the land of Kwue-El. From time to time, as I mentioned before, you'll be given a quick peek at a map of the land. On it, question marks are random caves, dollar signs are treasure caves, solid blocks and impassable barriers ... and so on. (I'll leave it to you to work out the other symbols.)

If you're a veteran of computer games, you'll recognise that this program is a development of the old *Hunt the Wumpus* games. In this program, unlike the original *Wumpus* ones, you can fight the dragon if you like. You have five arrows at the start of the game, and you can fire into any cave adjoining yours in one of the four directions.

You need to survive for 25 minutes to win this game, and your final score depends upon how many dragons you slay, and how much treasure you find. You have a magic amulet with you which gives you somewhat unreliable information on the contents of the caves near you. Your success in this game depends on your quick wits, and your ability at visualising the cave system.

When you're ready to follow the prompts and risk your life, enter and run this program:

```
10 REMark The Marquis of Kwue-El
20    initialise
30    CLS:PRINT \\\
40    map
50 REPeat game_cycle
60    q=RND(0 TO 6)
```

```
70   IF q=0 AND e<>55 THEN map
80  INK 6:PAPER 2
90  CLS:PRINT\\\\
100   PRINT "  Marquis ";name$;", you
 are in cave ";e
110   IF g>0 THEN PRINT "  You're car
rying ";:INK 0:PAPER 7:FLASH 1:PRINT
"$";g;:INK 6:PAPER 2:FLASH 0:PRINT "
of gold"
120   amulet
130 PRINT \" You have ";:PAPER 6:INK
2:PRINT 25-h;:PAPER 2:INK 6:PRINT " u
nits of charisma left"
140     FLASH 1
150 PRINT \"    What do you want to d
o?"
160     FLASH 0
170 IF INKEY$<>"" THEN GO TO 170
180 PRINT \"N - to move north  S - to
 move south"
190 PRINT  "E - to move east   W - to
 move west"
200 PRINT  "F - fight dragon   Q - qu
it"
210 j$=INKEY$:BEEP 100,RND(3 TO 13)
220 IF j$<>"n" AND j$<>"N" AND j$<>"s
" AND j$<>"S" AND j$<>"e" AND j$<>"E"
 AND j$<>"w" AND j$<>"W" AND j$<>"f"
AND j$<>"F" AND j$<>"q" AND j$<>"Q" T
HEN GO TO 210
230 u=0
240 IF (j$="n" OR j$="N") AND a(e-10)
=88 OR (j$="s" OR j$="S") AND a(e+10)
=88 OR (j$="e" OR j$="E") AND a(e+1)=
88 OR (j$="w" OR j$="W") AND a(e-1)=8
8 THEN
250  PRINT "You cannot move that way!
"
260  GO TO 210
270 END IF
280 IF j$="q" OR j$="Q" THEN
```

```
290  q=9
300  end_of_game
310  STOP
320 END IF
330 a(e)=46
340 IF j$="n" OR j$="N" THEN e=e-10
350 IF j$="s" OR j$="S" THEN e=e+10
360 IF j$="e" OR j$="E" THEN e=e+1
370 IF j$="w" OR j$="W" THEN e=e-1
380 IF j$="f" OR j$="F" THEN fight_dr
agon
390 IF a(e)=63 THEN magic
400 IF a(e)=68 THEN dragon
410 IF a(e)=81 THEN quicksand
420 IF a(e)=36 THEN gold
430 h=h+1
440 IF h=25 THEN
450   q=9
460   end_of_game
470   STOP
480 END IF
490 delay
500 END REPeat game_cycle
510 REMark ---------------
520 DEFine PROCedure magic
530  INK 7:FLASH 1
540 PRINT \" Marquis ";name$;", you've
e stumbled"
550 PRINT " into a magic cave, and no
w you'll"
560 PRINT " be whisked to another cav
e..."
570  INK 6:FLASH 0
580  FOR cambridge=1 TO 6
590   FOR research=1 TO 6
600    BEEP 10900,research,199,1,1
610    PAUSE 10/research:BEEP 1000,ca
mbridge
620    BEEP 10090,10-cambridge,224,15
,1,11,7
630    PAUSE 15/cambridge
```

```
640   END FOR research
650  END FOR cambridge
660 a(e)=46
670 REPeat whisk
680  e=RND(12 TO 87)
690  IF a(e)<>88 THEN EXIT whisk
700 END REPeat whisk
710 END DEFine magic
720 REMark ---------------
730 DEFine PROCedure dragon
740 PRINT \\"You are in a dragon's la
ir..."
750 PRINT "      Start saying your pr
ayers"
760 delay
770  INK 5:PRINT \"-----------------
---------------":INK 6
780 m=RND(1 TO 5)
790 IF m=1 THEN
800  PRINT "It has flown away"
810  delay
820  END DEFine dragon
830 END IF
840 PRINT "It awakens...and it has se
en you!"
850 delay
860 IF m=5 THEN
870  PRINT "It has recently eaten"
880  PRINT "and so goes back to sleep
"
890  delay
900  END DEFine dragon
910 END IF
920 PRINT \"And now it attacks!!"
930 delay
940 m=RND(1 TO 10)
950 IF m>8 THEN
960  PRINT "but you fight back valian
tly..."
970  delay
980  PRINT "          ...and win!"
```

```
990  END DEFine dragon
1000 END IF
1010 PRINT "Goodbye Marquis, ";name$
1020   delay
1030   q=9
1040   end_of_game
1050   STOP
1060 END DEFine dragon
1070 REMark ---------------
1080 DEFine PROCedure quicksand
1090  FOR flap=1 TO 10
1100   MODE 4:PAUSE 2:MODE 8:PAUSE 2:
BEEP 1000,flap*flap
1110  END FOR flap
1120 FOR suck=1 TO 12
1130  FOR slime=1 TO suck
1140   PRINT " ";
1150   BEEP 9600,suck*slime
1160  END FOR slime
1170   inkcol=RND(0 TO 7):papcol=7-in
kcol
1180   STRIP papcol:PRINT "Horrors...
.quicksand!!"
1190  BEEP 19700,27,95,14,suck,10,8,2
0770:PAUSE 5
1200 END FOR suck
1210 delay:delay
1220   PAPER 2:INK 6
1230 PRINT "

          "
1240   q=9:h=0
1250   end_of_game
1260   STOP
1270 END DEFine quicksand
1280 REMark ---------------
1290 DEFine PROCedure gold
1300 k=RND(100 TO 199)
1310 PRINT
1320 FOR gems=1 TO 12
1330 PRINT " ";
```

```
1340    FOR glitters=1 TO gems
1350     PRINT "$";
1360     BEEP 9950,glitters*gems:PAUS
E 4:BEEP 9950,(glitters+1)/gems
1370    END FOR glitters
1380   INK RND(4 TO 7)
1390   PRINT "  Treasure..."
1400   PRINT
1410   BEEP RND(3200 TO 23200),gems/g
litters,1,11,-6,11,7,871
1420   PAUSE gems
1430 END FOR gems
1440 delay
1450 INK 7
1460 PRINT "You've found a hoard of d
ragon"
1470 PRINT "     gold, worth $";k
1480 g=g+k
1490 delay
1500 END DEFine gold
1510 REMark ---------------
1520 DEFine PROCedure amulet
1530 y=1
1540 REPeat scan
1550  l=a(e+p(y))
1560  IF l<>46 OR y=8 THEN EXIT scan
1570  y=y+1
1580 END REPeat scan
1590 IF l=46 THEN END DEFine amulet
1600 PRINT \"  Your amulet signals th
at"
1610 PRINT "  there is ";
1620 SELect ON l
1630  ON l=88
1640   PRINT "a solid wall ";
1650  ON l=63
1660   PRINT "a magic cave ";
1670  ON l=68
1680   PRINT "a dragon ";
1690  ON l=81
1700   PRINT "quicksand ";
```

```
1710  ON l=36
1720   PRINT "gold ";
1730 END SELect
1740 PRINT "nearby"
1750 delay
1760 END DEFine amulet
1770 REMark ----------------
1780 DEFine PROCedure fight_dragon
1790 INK 7:PAPER 1
1800 ar=ar-1
1810   IF ar=0 THEN
1820    PRINT "You have used up all y
our arrows"
1830    delay
1840    END DEFine fight_dragon
1850   END IF
1860 PRINT \\"You have ";ar;" arrows
in your quiver"
1870 ss=0
1880 PRINT "Which direction do you wi
sh to shoot in (N, S, E or W), ";name
$;"?"
1890 REPeat get_shot
1900  s$=INKEY$
1910  IF (s$="n" OR s$="N") AND a(e-1
0)=68 THEN ss=1:yt=e-10
1920  IF (s$="s" OR s$="S") AND a(e+1
0)=68 THEN ss=1:yt=e+10
1930  IF (s$="e" OR s$="E") AND a(e+1
)=68 THEN ss=1:yt=e+1
1940  IF (s$="w" OR s$="W") AND a(e-1
)=68 THEN ss=1:yt=e-1
1950  IF s$="n" OR s$="N" OR s$="s" O
R s$="S" OR s$="e" OR s$="E" OR s$="w
" OR s$="W" THEN EXIT get_shot
1960 END REPeat get_shot
1970 PRINT
1980 IF ss=0 THEN
1990  PRINT "There was no dragon ther
e..."
2000  delay
```

```
2010  PRINT "You have wasted an arrow
"
2020  delay
2030  END DEFine fight_dragon
2040 END IF
2050 PRINT "Well done, Marquess ";nam
e$
2060 PRINT \"You have hit a ferocious
 dragon"
2070 delay
2080 IF RND(1 TO 3)=2 THEN
2090  PRINT "You killed it!"
2100  a(yt)=46:k=RND(100 TO 200)
2110  PRINT \"You are rewarded with $
";k
2120  g=g+k
2130 ELSE
2140 PRINT "But it is only wounded...
"
2150 END IF
2160 delay
2170 END DEFine fight_dragon
2180 REMark ------------------
2190 DEFine PROCedure end_of_game
2200 PAPER 2:INK 6
2210 IF h<1 THEN
2220  PRINT "All your charisma is exh
austed..."
2230  ELSE
2240   PRINT "You have ";25-h;" units
 of charisma left"
2250 END IF
2260 delay
2270 IF g>0 THEN
2280  PRINT "You amassed $";g;" worth
 of gold"
2290 END IF
2300   map
2310 END DEFine end_of_game
2320 CSIZE 3,0
2330 DEFine PROCedure map
```

```
2340 PRINT \\
2350 CSIZE 1,0
2360 a(e)=72
2370 PRINT "  ";
2380 FOR j=1 TO 100
2390  PAPER 2:INK 6
2400  m=a(j)
2410  IF a(j)=88 THEN INK 1:PAPER 7
2420  IF a(j)=63 THEN INK 7:PAPER 3:F
LASH 1
2430  IF a(j)=68 THEN INK 6:PAPER 2
2440  IF a(j)=81 THEN INK 7:PAPER 1
2450  IF a(j)=36 THEN INK 2:PAPER 7
2460  IF a(j)=72 THEN INK 0:PAPER 5
2470  PRINT CHR$(m);
2480  PAPER 2:INK 6:FLASH 0
2490  PRINT " ";
2500  IF 10*INT(j/10)=j THEN PRINT:PR
INT "  ";
2510 END FOR j
2520   delay
2530 CSIZE 0,0
2540 END DEFine map
2550 REMark ---------------
2560 DEFine PROCedure initialise
2570 BORDER 3,1
2580 PAPER 2:INK 6
2590 CLS:CLS #0
2600 PRINT \\"  Welcome to the world
of Kwue-El"
2610   delay:delay
2620 PRINT \"  Your mission is to exp
lore the"
2630 PRINT " caves of this world, loo
king for"
2640 PRINT " treasure, and attempting
 to slay"
2650 PRINT " the fearsome dragons whi
ch live in"
2660 PRINT "  the caves.  What is you
r name?"
```

```
2670 PRINT \\\"                ";
2680 INPUT name$
2690  name$=name$(1 TO 7)
2700 PAPER 6:INK 0
2710 CLS
2720 PRINT \\\\"    All hail, Marquis
 ";name$
2730  delay
2740 PRINT \\"  You start this explor
ation with"
2750 PRINT "   25 units of charisma,
and you"
2760 PRINT " must complete your task
before your"
2770 PRINT " charisma is worn out. Yo
u consume"
2780 PRINT " one unit for each move t
hat you"
2790 PRINT "                  make.."
2800 DIM a(100):DIM p(8)
2810 h=0:q=0:l=0:g=0:ar=6
2820 FOR b=1 TO 100
2830  a(b)=46
2840  BEEP 900,b
2850  IF b<12 OR b>90 OR 10*INT(b/10)
=b OR 10*INT(b/10)=b-1 THEN a(b)=88
2860 END FOR b
2870 FOR b=1 TO 5
2880    RESTORE
2890   FOR d=1 TO 5
2900    z=RND(12 TO 87)
2910    IF a(z)=88 THEN GO TO 2900
2920    READ a(z)
2930    BEEP 1000,d*b
2940   END FOR d
2950  END FOR b
2960 FOR b=1 TO 8
2970  READ p(b)
2980 END FOR b
2990 e=55
3000  PAPER 2:INK 6:CLS
```

```
3010 END DEFine initialise
3020 REMark ------------------
3030 DEFine PROCedure delay
3040  FOR patience=1 TO 10
3050   BEEP 1000,RND(1 TO 100),RND(10
0 TO 200),RND(1 TO 8),RND(1 TO 8),RND
(1 TO 32000)
3060   PAUSE 4
3070  END FOR patience
3080 END DEFine delay
3090 REMark ------------------
3100 DATA 88,63,68,81,36
3110 DATA -11,-10,-9,-1,1,9,10,11
```

Werewolves

Our second adventure is called *Werewolves and Wanderer* (where you are the wanderer, and the werewolves and other unsavoury creatures inhabit the imaginary environment you are about to visit), takes place within an ancient, abandoned, stone castle.

You read about the castle in a faded letter you found in a trunk left to you by your grandfather. Unfortunately, you could only find the second page of the letter, so you are not sure of the full story. However, from the page remaining, you understand the castle was abandoned centuries ago after a curse was placed on the inhabitants by an old witch who had been evicted during a particularly harsh thunderstorm. The king's wife was ailing, and he (wrongly, as it turned out) blamed the witch for his wife's illness. He thought by throwing the witch out, her malignant influence on his wife would cease. This did not happen, and his wife became more and more ill, and finally died.

Her last days were not peaceful. The old woman's curse brought a reign of terror to the castle, as many odd creatures and ghosts took up residence within the castle. Finally, the king and his court could stand it no more, and they fled from their former home, never to be heard of again.

The creatures invoked by the witch stayed on, and are living there still. You are about to enter their realm.

The program begins by asking you your name, so it can use it throughout the adventure. The QL will then tell you – as it will after each action you initiate – your current status. You'll be told how much money you have, your strength rating, and such things as whether or not you are wearing armour.

Next you'll be asked what you want to do, and you enter a single-letter command to give the QL your instructions. You'll need to start with "I" (for inventory) in order to buy a torch. Without it, you'll be unable to explore the castle at all.

You'll also be advised to buy some food when in "inventory mode". Food is a very valuable possession. You start the game with a limited amount of strength, which is gradually consumed as the game goes on. If it drops to zero, you die, and the game ends. Fights against wild monsters also diminish your strength. Eating food replenishes it. You have to keep an eye on your current 'strength rating' throughout the game, and always make sure you have food on hand to replenish your rating when needed.

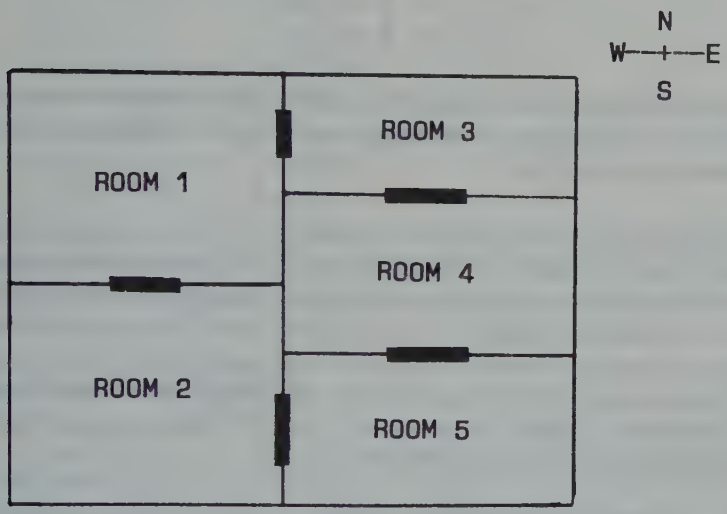
Mapping the Environment

As was pointed out earlier, an adventure environment must be coherent. That is, the explorer making his way through the environment must be able to draw up a complete map as he works his way through it. If he draws a door connecting the study with the library on an environment floor plan, because he has discovered that going through the study door leads into the library, he is entitled to expect that turning around and going back will bring him back into the study. The game-player should be able to build up an entire plan in this way, checking his plan from time to time by 'walking around' the house, castle, forest, underground labyrinth or whatever where the adventure is taking place.

The first step, then, in building an adventure program is to construct an environment which can be both mapped, and represented in some way which the computer can store.

You'll be pleased to know it is relatively easy to satisfy both these conditions.

Look at the following five-room environment, a very simple one, which we shall treat as though it was a computer adventure environment.



The key to holding an environment like this in a way the computer can understand and manipulate is to set up an array, each element of which represents a room. The solid markers between rooms are doors.

If you were in room one, you could move east into room three, or south into room two. In room four you can move north into room three and south into room five, and so on. Imagine we have set up an array, which we have dimensioned as DIM A (5,4). The first dimension is the room, and the second one is the four possible directions from that room (that is, north, south, east and west).

Armed with the map of the five-room environment, we can now build up a *travel table,* which can then be fed into the array, to allow us to move from point to point within the environment. Here's the travel table for the simple, five-roomed environment we've mapped:

ROOM	N	S	E	W
1	Ø	2	3	Ø
2	1	Ø	5	Ø
3	Ø	4	Ø	1
4	3	5	Ø	Ø
5	4	Ø	Ø	2

Take some time to study this table, and the way it relates to the map, because it is the single most important key to building adventure programs you can learn.

Look at the table for room one. Under the 'N' (for north) column, we see a zero, meaning you cannot move north from room one (a fact which is easily verified by looking at our map). However, under the "S" we see the number two, meaning that if we travelled south from room one we would end up in room two (again you can verify this from the map). Move east (the 'E' column) from room one, and you'll end up in room three. The Ø in the 'W' column means there is no travel possible west from room one.

You can work right through the table, if you like, checking that the numbers on it correspond to the 'reality' of the map.

Now, to allow the player to move around the environment, we only need to (a) fill each element of the array with the relevant information from the *travel table;* (b) tell the player where he or she is; and (c) allow the decisions entered by the player regarding the direction he or she wants to move to be checked against the array, and then – if possible – updated to reflect the player's new location. It is easier to do this than you might think.

Moving About

Firstly, we need to write a small program to feed the relevant information into the array. Two simple READ/DATA loops like the following will do it:

```
100 DIM a(5,4)
110 RESTORE
120 :
```

```
130 FOR b=1 TO 5
140  FOR c=1 TO 4
150   READ a(b,c)
160  END FOR c
170 END FOR b
180 :
190 DATA 0,2,3,0
200 DATA 1,0,5,0
210 DATA 0,4,0,1
220 DATA 3,5,0,0
230 DATA 4,0,0,2
```

As you can see, the DATA statements correspond exactly with the items in our Travel Table.

The Player's Location

If we decided that the room the player is currently occupying could be designated by the variable RO we could tell the player where he or she was as follows, as well as indicating which exits existed:

```
100 PRINT "You are now in room number
 ";ro
110 IF a(ro,1)<>0 THEN PRINT "A door
leads north"
120 IF a(ro,2)<>0 THEN PRINT "There i
s an exit to the south"
130 IF a(ro,3)<>0 THEN PRINT "You can
 leave via the east"
140 IF a(ro,4)<>0 THEN PRINT "A doorw
ay opens to the west"
```

The player's input could be a single letter ("N" for north, and so on) and the program could look at the input, and check to see if an exit to that direction existed:

```
790  IF (mo$="n" OR mo$="N") AND a(ro
,1)=0 THEN PRINT " No exit that way":
GO TO 650
800  IF (mo$="s" OR mo$="S") AND a(ro
```

```
,2)=0 THEN PRINT " There is no exit s
outh":GO TO 650
810  IF (mo$="e" OR mo$="E") AND a(ro
,3)=0 THEN PRINT " You cannot go in t
hat direction":GO TO 650
820  IF (mo$="w" OR mo$="W") AND a(ro
,4)=0 THEN PRINT " You cannot move th
rough stone":GO TO 650
830  IF (mo$="u" OR mo$="U") AND a(ro
,5)=0 THEN PRINT " There is no way up
 from here":GO TO 650
840  IF (mo$="d" OR mo$="D") AND a(ro
,6)=0 THEN PRINT " You cannot descend
 from here":GO TO 650
```

Consistency and Reality

Although the rooms only exist on paper and in elements in an array, the fact that they behave like 'real rooms' soon allows them to be perceived as though they were solid and real in a way which is uncanny. Add descriptions of each room – YOU ARE IN A SMALL WORKMAN'S HUT ATTACHED TO THE BACK OF THE MANOR HOUSE, WITH A PILE OF STRAW OVER IN THE FAR CORNER, AND A SHOVEL AND AN AXE LYING UNDERNEATH THE WINDOW. A LARGE LOAF OF BREAD IS ON THE TABLE, AND BESIDE IT IS A NOTE. DOORS LEAVE TO THE NORTH AND TO THE WEST – and you'll find the environment takes on quite solid dimensions in your mind.

Once the map becomes more complex, as in WEREWOLVES AND WANDERER, and the descriptions help clarify the mental images of the rooms, you'll find you have a counterfeit reality with immense power in your hands.

You might like to try and write a simple program, before proceeding further, which allows you to move around the five-room environment we've been looking at.

Commands

Here are the single-letter commands you use to 'drive' this adventure:

F – fight
R – run from a monster (not always allowed)
Q – quit the adventure (shame!)
N – move north
S – move south
E – move east
W – move west
U – move upwards (the castle is mapped on *three* levels)
D – move downwards
I – inventory (to buy the torch and other supplies)
C – consume food
P – pick up treasure
M – invoke the magic amulet to move at random (it can only be used once, so save it until a time of real need)

Your score at the end of the program is related to a number of factors, including the length of time you've survived, how much strength you have, the wealth you've found, and whether or not you're still carrying the magic amulet.

If you feel brave enough, enter and run our *Werewolves* program now.

```
10 REMark Werewolves and Wanderer
20  RESTORE :CLEAR
30  begin
40 REPeat loop
50   major_routine
60   IF ro=11 THEN EXIT loop
70 END REPeat loop
80 REMark -- End of game --
90 PRINT \"  You've done it!!"
100  delay
110 PRINT \"  That was the exit from
the castle"
120  delay
130 PRINT \\" You have succeeded, ";n
```

```
ame$;"!"
140 PRINT \"You managed to get out of
 the castle"
150  delay
160 PRINT \"        Well done!"
170  delay
180 PRINT \"  Your score is ";
190 PRINT 39*tally + 51*strength + 21
*wealth + food + 309*mk + 45*suit + 1
00*amulet + 17*sword + 7*axe
200 STOP
210 REMark ----------------
220 DEFine PROCedure major_routine
230  strength=strength-5
240  IF strength<25 AND strength>=1 T
HEN
250   PRINT "   Warning, ";name$;", y
our"
260   PRINT "    strength is running
out!"
270  END IF
280  IF strength<1 THEN
290    PRINT " Your strength is all g
one..."\"     and you are now dead...
"
300   GO TO 170
310  END IF
320  tally=tally+1
330  PRINT \\" ";name$;", your streng
th is ";strength
340  IF wealth>0 THEN PRINT "  You ha
ve $";wealth
350  IF food>0 THEN PRINT "  Your pro
visions sack holds"\"          ";food;
" units of food"
360  IF suit=1 THEN PRINT " You are w
earing armour"
370  IF axe<>0 OR sword<>0 OR amulet<
>0 THEN
380   PRINT " You are carrying ";
390   IF axe=1 THEN PRINT "an axe ";
```

```
400   IF sword=1 THEN PRINT "a sword
";
410   IF sword+axe>0 AND amulet=1 THE
N PRINT "and ";
420   IF amulet=1 THEN PRINT !"the"!"
magic"!"amulet"
430  END IF
440 PRINT
450  IF light=0 THEN
460   PRINT "  It is too dark to see
anything"
470  ELSE
480   room_description
490  END IF
500  k=a(ro,7)
510   IF k>9 THEN INK 2:PAPER 7:PRINT
 \" There is treasure here worth $";k
:INK 6:PAPER 0:END IF
520  IF k<0 THEN
530   BEEP 1000,1:PAUSE 10:BEEP 1000,
2:PAUSE 10:BEEP 1000,3
540   INK 0:PAPER 5
550   PRINT \\\"Danger!!! There is a
monster here...":delay
560   IF k=-1 THEN m$="Ferocious Were
wolf":ff=5
570   IF k=-2 THEN m$="Fanatical Fles
hgorger":ff=10
580   IF k=-3 THEN m$="Maloventy Mald
emer":ff=15
590   IF k=-4 THEN m$="Devastating Ic
e-Dragon":ff=20
600   PRINT \"  It is a ";m$
610   PRINT \"  The danger level is "
;ff;"!!"
620   PRINT "
           "
630   INK 6:PAPER 0
640  END IF
650  delay
660  FLASH 1
```

```
670  PRINT \\" What do you want to do
? "
680  FLASH 0
690  REPeat get_move
700   mo$=INKEY$
710   IF k<0 AND (mo$<>"f" AND mo$<>"
F" AND mo$<>"r" AND mo$<>"R") THEN GO
 TO 700
720   IF mo$<>"" THEN EXIT get_move
730  END REPeat get_move
740  INK 1
750  PRINT \"-----------------------
-------------"
760  INK 6
770  PRINT
780  IF (mo$="q" OR mo$="Q") THEN GO
TO 180
790  IF (mo$="n" OR mo$="N") AND a(ro
,1)=0 THEN PRINT " No exit that way":
GO TO 650
800  IF (mo$="s" OR mo$="S") AND a(ro
,2)=0 THEN PRINT " There is no exit s
outh":GO TO 650
810  IF (mo$="e" OR mo$="E") AND a(ro
,3)=0 THEN PRINT " You cannot go in t
hat direction":GO TO 650
820  IF (mo$="w" OR mo$="W") AND a(ro
,4)=0 THEN PRINT " You cannot move th
rough stone":GO TO 650
830  IF (mo$="u" OR mo$="U") AND a(ro
,5)=0 THEN PRINT " There is no way up
 from here":GO TO 650
840  IF (mo$="d" OR mo$="D") AND a(ro
,6)=0 THEN PRINT " You cannot descend
 from here":GO TO 650
850  IF (mo$="r" OR mo$="R") AND RND>
.7 THEN
860   PRINT " No, you must stand and
fight"
870   mo$="f"
880   delay
```

```
890  END IF
900  IF mo$="r" OR mo$="R" THEN
910   PRINT " Which way do you want t
o run?"
920   delay
930   k=0
940   mo$=""
950   GO TO 680
960  END IF
970  IF (mo$="f" OR mo$="F") AND a(ro
,7)>-1 THEN
980   PRINT " There is nothing to fig
ht here!"
990   GO TO 650
1000  END IF
1010  IF mo$="i" OR mo$="I" THEN inve
ntory
1020  IF (mo$="c" OR mo$="C") AND foo
d=0 THEN
1030   PRINT " You have no food!"
1040   GO TO 650
1050  END IF
1060  IF mo$="p" OR mo$="P" THEN trea
sure
1070  IF mo$="f" OR mo$="F" THEN figh
t
1080  IF mo$="m" OR mo$="M" THEN
1090   REPeat choose
1100    ro=RND(1 TO 19)
1110    IF ro<>6 AND ro<>11 THEN EXIT
 choose
1120   END REPeat choose
1130   amulet=0:REMark Amulet is only
                    good for one spe
ll
1140  END IF
1150  IF mo$="c" OR mo$="C" THEN eat_
food
1160  IF mo$="n" OR mo$="N" THEN ro=a
(ro,1)
1170  IF mo$="s" OR mo$="S" THEN ro=a
```

```
(ro,2)
1180  IF mo$="e" OR mo$="E" THEN ro=a
(ro,3)
1190  IF mo$="w" OR mo$="W" THEN ro=a
(ro,4)
1200  IF mo$="u" OR mo$="U" THEN ro=a
(ro,5)
1210  IF mo$="d" OR mo$="D" THEN ro=a
(ro,6)
1220 END DEFine major_routine
1230 REMark --------------
1240 DEFine PROCedure fight
1250  FOR j=1 TO 30
1260   BEEP 100,j:BEEP 100,200-j
1270  END FOR j
1280  IF INKEY$<>"" THEN GO TO 1280
1290  FLASH 1:PRINT \"  Press any key
 to fight...":FLASH 0
1300  REPeat getkey
1310   IF INKEY$<>"" THEN EXIT getkey
1320  BEEP 100,5:END REPeat getkey
1330  FOR j=1 TO 6
1340   INK RND(1 TO 7)
1350   PRINT \" * - * - * - * - * - *
 - * - *"
1360  END FOR j
1370  INK 6
1380  IF suit=1 THEN
1390   PRINT " Your armour increases
your chance   of success"
1400   ff=3*(INT(ff/4))
1410   delay
1420  END IF
1430  IF axe=0 AND sword=0 THEN
1440   PRINT " You have no weapons an
d must           fight with your bare
hands"
1450   ff=INT(ff+ff/5)
1460   delay
1470  END IF
1480  IF axe<>0 OR sword<>0 THEN
```

```
1490   IF axe=1 AND sword=0 THEN
1500    PRINT " You have only an axe
to                      fight with"
1510    ff=4*INT(ff/5)
1520   END IF
1530   IF axe=0 AND sword=1 THEN
1540    PRINT " You must fight with
                        your sword"
1550    ff=3*INT(ff/4)
1560   END IF
1570   IF axe=1 AND sword=1 THEN
1580    PRINT " Which weapon? 1 - Axe
  2 - Sword"
1590    REPeat choose
1600     weap$=INKEY$
1610     IF weap$="1" OR weap$="2" TH
EN EXIT choose
1620    END REPeat choose
1630   z=weap$
1640   IF z=1 THEN ff=4*INT(ff/5)
1650   IF z=2 THEN ff=3*INT(ff/4)
1660  END IF
1670  END IF
1680  REMark -- The Battle --
1690  PRINT \
1700 REPeat melee
1710  PRINT:FLASH RND(0 TO 1)
1720   INK RND(3 TO 7)
1730   BEEP 1000,RND(10 TO 212)
1740   BEEP 1000,RND(10 TO 212)
1750  IF RND>.5 THEN
1760   PRINT " The ";m$;" attacks"
1770    ELSE
1780   PRINT "You attack the ";m$
1790  END IF
1800   delay
1810  IF RND>.5 THEN
1820   PRINT " You manage to wound it
!"
1830   ff=INT(5*ff/6)
1840  END IF
```

```
1850   delay
1860  IF RND>.5 THEN
1870   PRINT " The monster wounds you
!"
1880   strength=strength-5
1890  END IF
1900  IF RND<.35 THEN EXIT melee
1910 END REPeat melee
1920   INK 6
1930  IF RND>ff/10 THEN
1940   PRINT \" And you managed to ki
ll the          ";m$;"!!"
1950   mk=mk+1
1960   strength=strength+INT(strength
/2)
1970    ELSE
1980   PRINT \" The ";m$!"defeated"!"
you..."
1990   strength=INT(strength/2)
2000  END IF
2010  a(ro,7)=0:delay:delay
2020  FLASH 0
2030 END DEFine fight
2040 REMark ---------------
2050 DEFine PROCedure room_descriptio
n
2060       INK 2
2070 PRINT \" ^v^v^v^v^v^v^v^v^v^v^v^
v^v^v^v^v^
              "
2080      INK 6
2090 SELect ON ro
2100 ON ro=1
2110 PRINT " You are in a hallway."
2120 PRINT " There is a door to the s
outh, and"
2130 PRINT " through windows to the n
orth you"
2140 PRINT " can see a secret herb ga
rden."
2150 ON ro=2
```

```
2160 PRINT " This is the Audience Cha
mber."
2170 PRINT " There is a window to the
 west. By"
2180 PRINT " looking to the right you
 can see"
2190 PRINT " the entrance to the cast
le."
2200 PRINT " Doors leave this room to
 the north,          east and south.
"
2210 ON ro=3
2220 PRINT " You are in the Great Hal
l, an"
2230 PRINT " L-shaped room. There are
 doors to"
2240 PRINT " the east and to the nort
h. There"
2250 PRINT " is a door to the west in
                    the alcove."
2260 ON ro=4
2270 PRINT " This is the Monarch's pr
ivate"
2280 PRINT " meeting room. There is a
 single              exit to the sou
th"
2290 ON ro=5
2300 PRINT " This inner hallway conta
ins a door"
2310 PRINT " to the north, and one to
 the"
2320 PRINT " west, and a circular sta
irwell"
2330 PRINT " passes through the room.
 You can"
2340 PRINT " see an ornamental lake t
hrough the"
2350 PRINT "          windows to the so
uth"
2360 ON ro=6
2370 PRINT " You are at the entrance
```

```
to a"
2380 PRINT " forbidding-looking stone
 castle.           You are facing east
."
2390 ON ro=7
2400 PRINT " This is the castle's kit
chen."
2410 PRINT " Through windows in the n
orth wall"
2420 PRINT " you can see a secret her
b garden."
2430 PRINT " A door leaves the kitche
n to                    the south"
2440 ON ro=8
2450 PRINT " You are in the store roo
m, amidst"
2460 PRINT " spices, vegetables, and
vast sacks"
2470 PRINT " of flour and other provi
sions."
2480 PRINT "There is a door to the no
rth, and             one to the south.
"
2490 ON ro=9
2500 PRINT " You have entered the lif
t..."
2510  delay
2520 PRINT " It slowly descends..."
2530  delay:ro=10
2540 GO TO 3080
2550 ON ro=10
2560 PRINT " You are in the rear vest
ibule."
2570 PRINT " There are windows to the
 south"
2580 PRINT " from which you can see t
he"
2590 PRINT " ornamental lake. There i
s an exit"
2600 PRINT " to the east and one to t
he north."
```

```
2610 ON ro=12
2620 PRINT " You are in the dank, dar
k dungeon."
2630 PRINT " There is a single exit,
a small"
2640 PRINT " hole in the wall towards
 the west."
2650 ON ro=13
2660 PRINT " You find yourself in the
 prison"
2670 PRINT " guardroom, in the baseme
nt of the"
2680 PRINT " castle. The stairwell en
ds in this"
2690 PRINT " room.  There is one othe
r exit, a"
2700 PRINT "   small hole in the east
 wall."
2710 ON ro=14
2720 PRINT " You are in the master be
droom on"
2730 PRINT " the upper level of the c
astle..."
2740 PRINT " Looking down from the wi
ndow to"
2750 PRINT " the west, you can see th
e entrance"
2760 PRINT " to the castle, while the
 secret"
2770 PRINT " herb garden is visible b
elow the"
2780 PRINT " north window. There are
doors to"
2790 PRINT "      east and to the sou
th."
2800 ON ro=15
2810 PRINT " This is the L-shaped upp
er hallway."
2820 PRINT " There is a door to the n
orth, and"
2830 PRINT " you can also see a stair
```

```
well."
2840 PRINT " Through the south window
 you"
2850 PRINT " catch a glimpse of the l
ake."
2860 ON ro=16
2870 PRINT " This room was used as th
e castle"
2880 PRINT " Treasury in by-gone year
s...."
2890 PRINT " There are no windows, ju
st exits"
2900 PRINT " to the north and to the
east"
2910 ON ro=17
2920 PRINT " Ooh!! You are in the cha
mbermaids'"
2930 PRINT " bedroom. There is an exi
t to the"
2940 PRINT " west and a door to the s
outh."
2950 ON ro=18
2960 PRINT " This tiny room on the up
per level"
2970 PRINT " is the Dressing Chamber.
 There's"
2980 PRINT " a window to the north, w
ith a view"
2990 PRINT " of the herb garden down
below."
3000 PRINT "    A door leaves to the
south."
3010 ON ro=19
3020 PRINT " This is the small room o
utside the"
3030 PRINT " castle lift, which can b
e entered"
3040 PRINT " by a door to the north.
Another"
3050 PRINT " door leads to the west.
You can"
```

```
3060 PRINT "      see the lake through
 the"
3070 PRINT "          southern windows
."
3080 END SELect
3090 END DEFine room_description
3100 REMark ------------------
3110 DEFine PROCedure treasure
3120  FOR j=1 TO 60
3130   BEEP 101,j
3140  END FOR j
3150  IF a(ro,7)<10 THEN
3160   PRINT " There is no treasure t
o pick up"
3170   delay
3180  END IF
3190  IF light=0 THEN
3200   PRINT " You cannot see where i
t is"
3210   delay
3220 END IF
3230  IF light=1 AND a(ro,7)>9 THEN
3240   wealth=wealth+a(ro,7)
3250   a(ro,7)=0
3260   PRINT \" You pick up the treas
ure!"
3270   delay
3280  END IF
3290 END DEFine treasure
3300 REMark ------------------
3310 DEFine PROCedure eat_food
3320  FOR j=1 TO 20
3330   BEEP 500,200+j:BEEP 500,200-j
3340  END FOR j
3350  CLS
3360  IF food>=1 THEN
3370   PRINT " You have ";food;" unit
s of food"
3380   PRINT " How many do you want t
o eat?"
3390   REPeat food_input
```

```
3400     PRINT "          ";
3410     INPUT z
3420     IF z<=food THEN EXIT food_inp
ut
3430    END REPeat food_input
3440    food=INT(food-z)
3450    strength=INT(strength+5*z)
3460    delay
3470   END IF
3480 END DEFine eat_food
3490 REMark ------------------
3500 DEFine PROCedure begin
3510  BORDER 0,0:PAPER 0:INK 6
3520  CLS:CLS #0
3530  strength=100
3540  wealth=75
3550  food=0
3560  tally=0
3570  mk=0:REMark Number of monsters
                          killed
3580 REMark ------------------
3590 REMark Set up castle
3600 DIM a(19,7)
3610 FOR b=1 TO 19
3620  FOR c=1 TO 7
3630   READ a(b,c)
3640  END FOR c
3650  BEEP 1000,5*b
3660 END FOR b
3670 INPUT \\" What is your name, exp
lorer?

                        ";name$
3680 CLS
3690 ro=6:REMark Starting position
                     (ro=room number)
3700 sword=0
3710 amulet=0
3720 suit=0
3730 light=0
3740 axe=0
```

```
3750 REMark ------------------
3760 REMark Allot treasure
3770 FOR j=1 TO 4
3780  REPeat allot
3790   m=RND(1 TO 19)
3800   IF m<>6 AND m<>11 AND a(m,7)=0
 THEN EXIT allot
3810  END REPeat allot
3820  a(m,7)=RND(10 TO 110)
3830 END FOR j
3840 REMark ------------------
3850 REMark Allot monsters
3860 FOR j=1 TO 4
3870  REPeat allot
3880   m=RND(1 TO 18)
3890   IF m<>6 AND m<>11 AND a(m,7)=0
 THEN EXIT allot
3900  END REPeat allot
3910  a(m,7)=-j
3920 END FOR j
3930  a(4,7)=RND(100 TO 199)
3940  a(16,7)=RND(100 TO 199)
3950 END DEFine begin
3960 REMark ------------------
3970 DEFine PROCedure inventory
3980 cheat=0
3990 PRINT " Provisions and Inventory
"
4000 REPeat provisions
4010 delay
4020  IF wealth=0 THEN EXIT provision
s
4030 IF wealth>=.1 THEN
4040  PRINT \\"     You have $";wealt
h
4050  PRINT " You can buy 1 - Flaming
 torch ($15)"
4060  PRINT "               2 - Axe ($1
0)"
4070  PRINT "               3 - Sword (
$20)"
```

```
4080  PRINT "                4 - Food ($
2 per unit)"
4090  PRINT "                5 - Magic a
mulet ($30)"
4100  PRINT "                6 - Suit of
 armour ($50)"
4110  PRINT "                0 - To cont
inue game"
4120  FLASH 1
4130 PRINT \"    Enter number require
d"
4140  FLASH 0
4150 IF INKEY$<>"" THEN GO TO 4150
4160 REPeat getnum
4170  z$=INKEY$
4180  IF z$>="0" AND z$<="6" THEN EXI
T getnum
4190 END REPeat getnum
4200 z=z$
4210 IF z=0 THEN EXIT provisions
4220 IF wealth<0 AND z<>0 THEN cheat=
1:EXIT provisions
4230 SELect ON z
4240 ON z=1
4250  light=1
4260  wealth=wealth-15
4270 ON z=2
4280  axe=1
4290  wealth=wealth-10
4300 ON z=3
4310  sword=1
4320  wealth=wealth-20
4330 ON z=5
4340  amulet=1
4350  wealth=wealth-30
4360 ON z=6
4370  suit=1
4380  wealth=wealth-50
4390 ON z=4
4400  PRINT " How many units of food?
"
```

```
4410   REPeat how_many
4420    PRINT "      ";
4430    INPUT many
4440    IF 2*many<=wealth THEN EXIT h
ow_many
4450   END REPeat how_many
4460   many=INT(many)
4470   food=food+many
4480   wealth=wealth-2*many
4490 END SELect
4500 END IF
4510 END REPeat provisions
4520 IF cheat=1 THEN
4530  PRINT " You have tried to cheat
 me!!"
4540  wealth=0:suit=0:axe=0:sword=0
4550  amulet=0:food=INT(food/4)
4560  delay
4570 END IF
4580 END DEFine inventory
4590 REMark ----------------
4600 DEFine PROCedure delay
4610  IF INKEY$<>"" THEN GO TO 4610
4620  BEEP 14000,RND(1 TO 50)
4630  PAUSE 23
4640  BEEP 32000,RND(60 TO 112),255,4
,5,7,0,RND(0 TO 15)
4650  PAUSE 23
4660  BEEP 32000,RND(220 TO 250),255,
4,5,7,0,RND(0 TO 15)
4670  FOR patience=1 TO 500
4680  END FOR patience
4690 END DEFine delay
4700 REMark ------------------
4710 DATA 0,2,0,0,0,0,0:REMark room 1
4720 DATA 1,3,3,0,0,0,0:REMark 2
4730 DATA 2,0,5,2,0,0,0:REMark 3
4740 DATA 0,5,0,0,0,0,0:REMark 4
4750 DATA 4,0,0,3,15,13,0:REMark 5
4760 DATA 0,0,1,0,0,0,0:REMark 6
4770 DATA 0,8,0,0,0,0,0:REMark 7
```

```
4780 DATA 7,10,0,0,0,0,0:REMark 8
4790 DATA 0,19,0,8,0,8,0:REMark 9
4800 DATA 8,0,11,0,0,0,0:REMark 10
4810 DATA 0,0,10,0,0,0,0:REMark 11
4820 DATA 0,0,0,13,0,0,0:REMark 12
4830 DATA 0,0,12,0,5,0,0:REMark 13
4840 DATA 0,15,17,0,0,0,0:REMark 14
4850 DATA 14,0,0,0,0,5,0:REMark 15
4860 DATA 17,0,19,0,0,0,0:REMark 16
4870 DATA 18,16,0,14,0,0,0:REMark 17
4880 DATA 0,17,0,0,0,0,0:REMark 18
4890 DATA 9,0,16,0,0,0,0:REMark 19
```

Once you're fairly confident you know how this adventure was constructed, you can use the program you now have on microdrive to create other adventures of your own, using the program as a master 'command handler' into which you can pour your own map, room descriptions and monster and treasure names.

Support and Ideas

There is an impressive body of support literature for Adventure gaming, as a visit to your local Games shop will demonstrate. The selection which follows is very much a product of my own interest in the field, and it should not be seen as even an attempt at selecting the 'best' publications. However, the list is made up from those works which I've found of interest and value. There are probably over a hundred of equal worth, but at least this list should give you a starting point:

THROUGH DUNGEONS DEEP: A Fantasy Gamers' Handbook – Robert Plamondon (Reston Publishing Company, Inc., Reston, Virginia, 1982).

What is Dungeons and Dragons® –John Butterfield, Philip Parker and David Honigmann (Penguin Books Ltd., Harmondsworth, Middlesex, England, 1982). *Dungeons and Dragons is the adventure game which was originated by TSR Hobbies Inc.™ The term is a registered trademark.

Dicing with Dragons, an Introduction to Role-Playing Games – Ian Livingstone (Routledge & Kegan Paul, London, Melbourne and Henley, 1982).

Fantasy Role Playing Games – J Eric Holmes (Hippocrane Books, Inc., New York, 1981).

As well as books, there are a number of game-playing aids which will help you in building the framework environment within which your adventure can unfold. These include:

DUNGEONS & DRAGONS® boxed game sets are a good way to learn some of the possibilities of Adventure gaming. You can start (and stay, if you like) with the *BASIC SET WITH INTRODUCTORY MODULE.* This set typically contains two books, and a set of six dice, with various numbers of sides. The books are BASIC RULES which sets out the concepts of role-playing, and explains how characters are 'rolled up', how their personalities are derived, how fights are sorted out, and many aspects of the Dungeon or Game Master's role.

As well as dice and *BASIC RULES,* the set contains a campaign background, *THE KEEP ON THE BORDERLANDS.* This comes with a great deal of information, including a series of maps, room and player information, and further details on fight resolution. I feel that the *BASIC RULES* boxed set is probably the best source of ideas you can get your hands on. It is also a very good way to achieve greater understanding of how role-playing games are developed and controlled.

These sources of ideas are suggested, of course, only for your own use, to produce games for your own entertainment. You cannot incorporate copyright material into programs for public sale

The following people make and distribute adventure artifacts and games which may give you further ideas:

Avalon Hill Games, 65Ø High Street, North Finchley, London N12 ØNL.
Citadel Miniatures, 1Ø Victoria Street, Newark, Nottinghamshire.
Games of Liverpool, 5Ø-54 Manchester Street, Liverpool, L1 6ER.

Flying Buffalo, PO Box 1ØØ, Bath Street, Walsall, West Midlands.
Games Workshop Ltd., 1 Dalling Road, London, W6; 27-29 Sunbeam Road, London NW1Ø.
Simpubs Ltd., Oakfield House, 6Ø Oakfield Road, Altrincham, Cheshire WA15 8EW.
TSR Hobbies (UK) Ltd., The Mill, Rathmore Road, Cambridge, CB1 4AD.

Appendices

Appendix One — Multi-tasking Demonstration

This program, by Derek Wilson, which first appeared in *Quanta,* generates an on-screen read-out of the QL's real-time clock, running as an independent task. First RUN the following program with a formatted cartridge in microdrive 2. This generates a machine code file called "clock".

```
90 REMark   Multi-tasking - Clock
91 REMark From QUANTA - Vol 1 Issue 8
92 :
100 c=RESPR(100)
105 RESTORE
110 FOR i=0 TO 68 STEP 2
120  READ x:POKE_W i+c,x
130 END FOR i
140 SEXEC mdv2_clock,c,100,256
1000 DATA 29439,29697,28683,20033,174
02
1010 DATA 48,13944,200,20115,12040
1020 DATA 28691,20033,17402,74,-27698
1030 DATA 13944,236,20115,8279,-11314
1040 DATA 13944,208,20115,16961,16962
1050 DATA 30463,28688,20035,24794
1060 DATA 0,7,240,10,272,200
```

Once you have it in place, type:

```
EXEC mdv2_clock
```

This will produce a continuous time display near the bottom right hand corner of the display screen.

Note that line 250 governs the position and colour of the clock window, Change the second last 200 to 272 if you are running the clock in Mode 4. The DATA items are, in order:

BORDER colour — byte

BORDER width — byte

PAPER colour — byte

INK colour — byte

(These are pairs of bytes, entered by POKE_W as words. You generate the PAPER and INK word, for example as 256*PAPER + INK. Therefore, white PAPER and red INK is 256*7 + 2 = 1794)

WINDOW width — word

WINDOW height — word

x origin — word

y origin — word

Appendix Two — Mathematical Functions

A number of standard trigonometrical and mathematical functions are provided on the QL. This next list outlines them briefly. Note that angles for trig. functions need to be expressed in radians.

COS – this is used to work out the cosine of a number in the range -60000 to +60000. You use it by following the word COS with the argument in brackets: COS(564).

SIN – this will return the sine of the argument and, like COS, operates in the range -60000 to +60000. The format for using it is SIN(argument).

TAN – TAN(argument) will return the tangent of the argument, which must be between -30000 and +30000. You use it as TAN(90).

ATAN and ACOT are used to determine the arctangent and arcotangent of an argument. Unlike the previous functions we've looked at, there is no limit to the argument's size (apart from the QL's own limit). You use these as ATAN(argument) and ACOT (argument).

COT produces the cotangent of the argument. Like TAN, it only works with arguments in the range -30000 to +30000. The format is the same as the others: COT(argument).

EXP – you use this to get e raised to the power of the argument. To ensure that you don't overflow the QL's arithmetic limits, the argument must be between -500 and 500. You use it as EXP (argument).

LN and LOG – these are used to obtain logarithms of arguments. LN gives the natural log, and LOG returns the log to base 10. The argument must be positive, but has no upper limit (apart from that of the QL).

INT returns the integer of the argument giving, for example, 29 for INT(29.8756) and -28 for INT(-28.789). The argument must be

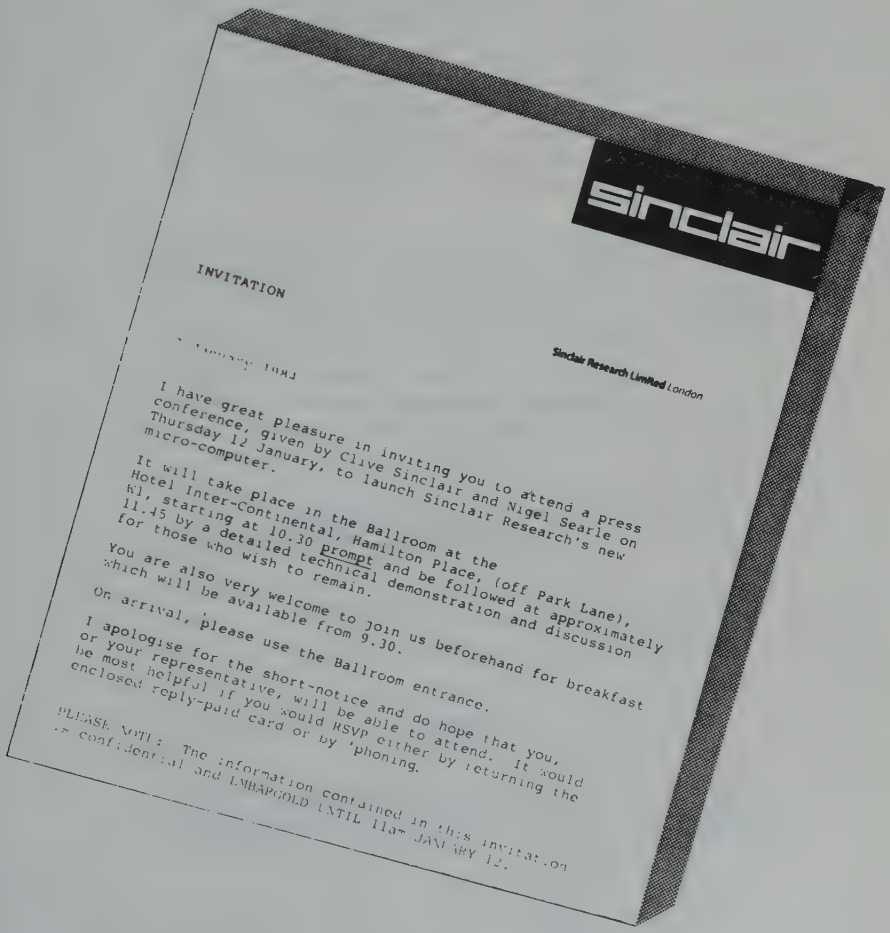
between –32767 and 32767. If you wish to round a number up to the next whole number, add Ø.5 as in INT(number + Ø.5).

ABS gives you the absolute value of an argument, giving you the number back if the argument is positive, so ABS(674) is 674, and minus the argument if it's negative, so ABS(–3452) is 3452.

SQRT is used to get the square root of an argument, which must be greater than zero. Simply follow the SQRT by the argument in brackets, as SQRT(876).

Appendix Three — How It All Began

How it all began. The letter which told us the QL was going to be revealed:



sinclair

Sinclair Research Limited London

INVITATION

[illegible] January 1984

I have great pleasure in inviting you to attend a press conference, given by Clive Sinclair and Nigel Searle on Thursday 12 January, to launch Sinclair Research's new micro-computer.

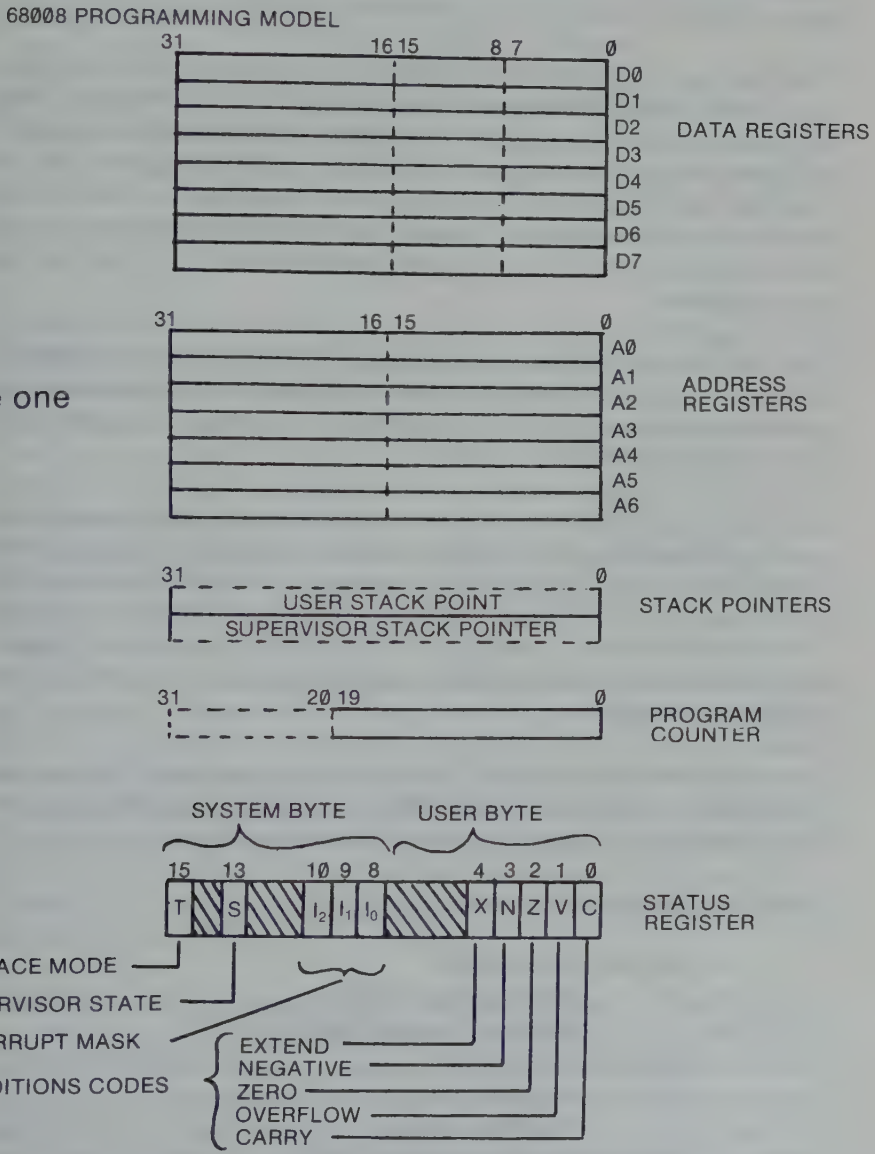
It will take place in the Ballroom at the Hotel Inter-Continental, Hamilton Place, (off Park Lane), W1, starting at 10.30 prompt and be followed at approximately 11.45 by a detailed technical demonstration and discussion for those who wish to remain.

You are also very welcome to join us beforehand for breakfast which will be available from 9.30.

On arrival, please use the Ballroom entrance.

I apologise for the short-notice and do hope that you, or your representative, will be able to attend. It would be most helpful if you would RSVP either by returning the enclosed reply-paid card or by 'phoning.

PLEASE NOTE: The information contained in this invitation is confidential and EMBARGOED UNTIL 11am JANUARY 12.



ure one

gure two

The 68ØØ8 programming model is described in figure one, and consists of 17 32-bit registers, a 32-bit program counter, and a 16-bit status register.

The first eight registers (DØ to D7) are used as data registers for bit, byte (8-bit), word (16-bit) and long word (32-bit) operations. The second set of seven registers (AØ to A6) and the system stack pointer (A7) may be used as stack pointers and base address registers. All 16 registers may be used as index registers. The system stack pointer may be either the supervisor stack pointer (SSP) or the user stack pointer (USP), depending upon the status of bit 13 (S bit) in the status register.

The status register (figure two) may be thought of as consisting of two bytes, a user byte and a system byte. The user byte contains five bits which define the carry (C), overflow (V), zero (Z), negative (N) and extend (X) condition codes.

The system byte contains five bits which define the current interrupt priority (four levels of interrupt), whether the processor is in Trace mode, and/or the supervisor or user state In Trace mode, the processor will go into an exception processing state (like a software interrupt) after every instruction. This makes the writing of single-step debugging utilities very easy. Hardware single-stepping is also quite easy to achieve.

The program counter is a 32-bit register, but only 2Ø bits are available in the case of the 68ØØ8, giving 1Mb of address space. Most instructions operate on bytes, words and long words.

There are 14 addressing modes available:

REGISTER DIRECT
- **Data Register Direct**
- **Address Register Direct**

ABSOLUTE DATA
- **Absolute Short**
- **Absolute Long**

PROGRAM COUNTER
- **Relative with Offset**
- **Relative with Index and Offset**

REGISTER INDIRECT
- **Register Indirect**
- **Post-increment Register Indirect**
- **Predecrement Register Indirect**
- **Register Indirect with Offset**
- **Indexed Register Indirect with Offset**

IMMEDIATE DATA
- **Immediate**
- **Quick Immediate**

IMPLIED
- **Implied Register**

The instruction set was designed around a relatively small number of mnemonics to make things easier for the programmer. However, the large number of addressing modes, which apply to most of the instructions (termed an orthogonal instruction set) mean there is a total of over 1000 useful instructions. These include signed and unsigned multiply/divide, quick arithmetic operations (with an 8-bit quantity as part of the op. code) and BCD arithmetic. If that is not enough, certain op. codes initiate exception processing, enabling you to define your own instructions and simulate their operation in software.

If you'd like to read more, 68000/68008 hardware is covered in two publications available from any Motorola distributor, such as Celdis, who are on (0734) 585171. These are the 16-bit Microprocessors Data Manual (which also includes the 8-bit peripheral chips) and both volumes of the MCU/MPU Applications Manual. These are around £10 each, I believe. You can obtain data sheets and application notes included in these publications for nothing, but it is probably worth paying for the books, in terms of convenience and easy reference.

68000Assembly Language Programming by Kane, Hawkins and Leventhal, published by Osborne/McGraw-Hill is recommended for 68000/68008 assembly language programmers. It is an excellent book, if rather expensive at about £17, and seems to have become difficult to obtain, possibly because of the QL. It might also be advisable to get the M68000 16/32-bit Microprocessor Programmer's Reference Manual, fourth edition, written by Motorola, and published by Prentice-Hall at about £15. This should be taken as 'the

Bible' when you are not sure about anything concerned with the 68ØØØ/68ØØ8 instruction set. Using this, it might be possible to hand-assemble small programs, because of the way the op. codes are documented.

Motorola sales offices will give you a Programmer's Reference Card for nothing, and I am told that they are also giving away cards with a detailed microphotograph of the 68ØØØ chip, and one of the chips itself (just the little piece of silicon) mounted on the card. These are reject devices, presumably.

This appendix was written by Leon Heller, and is based on material which appeared in *NATGUG News,* the newsletter of the National TRS-8Ø and Genie Users' Group and in *Quanta.* He is author of the book *Understanding the 68ØØ8.*

Appendix Five — Error Messages

Errors are reported by SuperBASIC in a standard form:

At line *line_number error_ text*

Where the line number is the number of the line where the error was detected and the error text is listed below.

(1) Not complete
An operation has been prematurely terminated.
(2) Invalid job
An error return from QDOS relating to system calls controlling multitasking or I/O.
(3) Out of memory
QDOS and/or SuperBASIC has insufficient free memory.
(4) Out of range
Usually results from an attempt to write outside a window or an array index is incorrect.
(5) Buffer full
An I/O operation to fetch a buffer full of characters filled the buffer before a record terminator was found.
(6) Channel not open
Attempt to read, write or close a channel which has not been opened.
(7) Not found
File system, device, medium or file cannot be found.
SuperBASIC cannot find an identifier. This can result from incorrectly nested structures.
(8) Already exists
The file system has found an already existing file with the same name as a new file to be opened.
(9) In use
The file system has found that a file or device is already exclusively used.
(10) End of file
End of file detected during input.
(11) Drive full
A Microdrive device has been filled.
(12) Bad name
The file system has recognised the name but there is a syntax

or parameter value error.
In SuperBASIC it means a name has been used out of context. For example, a variable has been used as a procedure.

(13) Xmit error
RS-232-C parity error

(14) Format failed
Attempted format operation has failed, the medium is possibly faulty.

(15) Bad parameter
There is an error in the parameter list of a system or SuperBASIC procedure or function call.
An attempt was made to read data from a write only device.

(16) Bad medium
The medium is possibly faulty.

(17) Error in expression
An error was detected while evaluating an expression.

(18) Overflow.
Arithmetic overflow, division by zero, square root of a negative number, etc.

(19) Not Implemented

(20) Read only
There has been an attempt to write data to a shared file.

(21) Bad line
SuperBASIC syntax error has occurred.
This error can also occur after an error in a function of procedure call. It will prevent any lines being typed in which would modify the program. Typing NEW or CLEAR will modify this condition.

Error Recovery

After an error has occurred the program can be restarted on the next line by typing

CONTINUE

If the error condition can be corrected, without changing the program, the program can be restarted on the line which triggered the error. Type

RETRY

Appendix Six — Glossary of Computer Terms

Accumulator – part of the computer's logic unit which stores the intermediate results of computations.

Address – a number which refers to a location, generally in the computer's memory, where information is stored.

Algorithm – the sequence of steps used to solve a problem.

Alphanumeric – generally used to describe a keyboard, and signifying that the keyboard has alphabetical and numerical keys. A numeric keypad, by contrast, only has keys for the digits one to nine, with some additional keys for arithmetic operations, much like a calculator.

APL – this stands for Automatic Programming Language, a language developed by Iverson in the early 196Øs, which supports a large set of operators and data structures. It uses a non-standard set of characters.

Application software – these are programs which are tailored for a specific task, such as word processing, or to handle mailing lists.

Artificial Intelligence – the section of computer science which concentrates on eliciting machine behaviour which, if it came from a human being, would be called intelligent. One of the aims of AI (as Artificial Intelligence is often written) is to make computers more useful. AI research may also turn out to help us understand our own thinking processes.

ASCII – this stands for American Standard Code for Information Exchange. This is an almost universal code for letters, numbers and symbols, which has a number between Ø and 255 assigned to each of these, such as 65 for the letter A.

Assembler – this is a program which converts another program written in an assembly language (which is a computer program in which a single instruction, such as ADD, converts into a single instruction for the computer) into the language the computer uses directly.

BASIC – stands for Beginner's All-purpose Symbolic Instruction Code, the most common language used on microcomputers. It is easy to learn, with many of its statements being very close to English. SuperBASIC is a major development of BASIC.

Batch – a group of transactions which are to be processed by a computer in one lot, without interruption by an operator.

Baud – a measure of the speed of transfer of data. It generally stands for the number of bits (discrete units of information) per second.

Benchmark – a test which is used to measure some aspect of the performance of a computer, which can be compared to the result of running a similar test on a different computer.

Binary – a system of counting in which there are only two symbols, '0' and '1' (as opposed to the ordinary decimal system, in which there are ten symbols, '0', '1', '2', '3', '4', '5', '6', '7', '8' and '9'). Your computer 'thinks' in binary.

Boolean Algebra – the algebra of decision-making and logic, developed by English mathematician George Boole, and at the heart of your computer's ability to make decisions. Much of a computer's strength in demonstrating 'intelligent' behaviour lies in its use of Boolean Algebra.

Bootstrap – a program, run into the computer when it is first turned on, which puts the computer into the state where it can accept and understand other programs.

Buffer – a storage mechanism which holds input from a device such as keyboard, then releases it at a rate which the computer dictates.

Bug – an error in a program.

Bus – a group of electrical connections used to link a computer with an ancillary device, or another computer.

Byte – the smallest group of bits which makes up a computer word. Generally a computer is described as being 'eight bit' or '16 bit', meaning the word consists of a combination of eight or sixteen zeros or ones.

Central Processing Unit (CPU) – the heart of the computer, where arithmetic, logic and control functions are carried out.

Character code – the number of ASCII (see ASCII) which refers to a particular symbol, such as 32 for a space and 65 for the letter 'A'.

COBOL – stands for Common Business Orientated Language, a standard programming language, close to English, which is used primarily for business.

Compiler – a program which translates a program written in a high level (human-like) language into a machine language which the computer is able to understand directly.

Concatenate – to add (adding two strings together is known as 'concatenation').

CP/M – these initials stand for Control Program/Microcomputer, an almost universal disk operating system developed and marketed by Digital Research, Pacific Grove, California.

Data – a general term for information processed by a computer.

Database – a collection of data, organised to permit rapid access by computer. A relational data base is one in which the interconnections between various elements within the database are stored explicitly to aid manipulation of, and access to, the elements within the data base.

Debug – to remove bugs (errors) from a program.

Disk – a magnetic storage medium (further described as a 'hard disk', 'floppy disk' or even 'floppy') used to store computer information and programs. The disks resemble, to a limited extent, 45 rpm sound records, and are generally eight, five and a quarter, or three and a half inches in diameter. Smaller 'microdisks' are also available for some systems.

Documentation – the written instructions and explanations which accompany a program.

DOS – stands for Disk Operating System (and generally pronounced 'doss'), the versatile program which allows a computer to control a disk system.

Dot-matrix printer – a printer which forms the letters and symbols by a collection of dots, usually on an eight by eight, or seven by five, grid.

Double-density – adjective used to describe disks when recorded using a special technique which, as the name suggests, doubles the amount of storage the disk can provide.

Dynamic memory – computer memory which requires constant recharging to retain its contents.

EPROM – stands for Erasable Programmable Read Only Memory, a device which contains computer information in a semi-permanent form, demanding sustained exposure to ultra-violet light to erase its contents.

Error messages – information from the computer to the user, sometimes consisting only of numbers or a few letters, but generally of a phrase (such as 'Out of memory') which points out a programming or operational error which has caused the computer to halt program execution.

Expert system – a computer program which, drawing on the encoded expertise of human experts, performs a specialised task as well as (or, in some cases, better than) the human expert in that field would. They often call on extensive databases of knowledge, and are sometimes called knowledge-based systems.

Field – a collection of characters which form a distinct group, such as an identifying code, a name or a date; a field is generally part of a record.

File – a group of related records which are processed together, such as an inventory file or a student file.

Firmware – the solid components of a computer system are often called the 'hardware', the programs, in machine-readable form on

disc or cassette, are called the 'software', and programs which are hard-wired into a circuit, are called 'firmware'. Firmware can be altered, to a limited extent, by software in some circumstances.

Flag – this is an indicator within a program, with the 'state of the flag' (ie the value it holds) giving information regarding a particular condition.

Floppy disk – see disk.

Flowchart – this is a written layout of program structure and flow, using various shapes, such as a rectangle with sloping sides for a computer action, and a diamond for a computer decision. A flowchart is generally written before any lines of program are entered into the computer.

FORTRAN – a high level computer language, generally used for scientific work (from FORmula TRANslation).

Gate – a computer 'component' which makes decisions, allowing the circuit to flow in one direction or another, depending on the conditions to be satisfied.

GIGO – acronym for 'Garbage In Garbage Out', suggesting that if rubbish or wrong data is fed into a computer, the result of its processing of such data (the output) must also be rubbish.

Global – a set of conditions which affects the entire program is called 'global', as opposed to 'local'.

Graphics – a term for any output of computer which is not alphanumeric, or symbolic.

Hardware – the solid parts of the computer (see 'software' and 'firmware').

Heuristic – a path towards a goal which has been worked out by experience, rather than by calculation; a path of this type is not guaranteed to produce a certain result (in contrast to an algorithm which is a technique or procedure which, when applied, produces a desired result); chess programs play, in large measure, heuristically.

Hexadecimal – a counting system often used by machine code programmers because it is closely related to the number storage methods used by computers, based on the number 16 (as opposed to our 'ordinary' number system which is based on 1Ø).

Hex pad – a keyboard, somewhat like a calculator, which is used for direct entry of hexadecimal numbers.

High-level languages – programming languages which are close to English. Low-level languages are closer to those which the computer understands. Because high-level languages have to be compiled into a form which the computer can understand before they are processed, high-level languages run more slowly than do their low-level counterparts.

Human interface – the 'outward and visible sign' of an expert system, with which the human user of the system interacts; these often work with natural language.

Inference system – the mechanism by which a computer program reaches conclusions; some systems make hard and fast YES/NO decisions, others operate in the world of 'fuzzy logic', where shades of grey (degrees of uncertainty) are permitted.

Input – any information which is fed into a program during execution.

I/O – stands for Input/Output port, a device the computer uses to communicate with the outside world.

Instruction – an element of programming code, which tells the computer to carry out a specific task. An instruction in assembler language, for example, could be ADD which (as you've guessed) tells the computer to carry out an addition.

Interpreter – converts the high-level ('human-understandable') program into a form which the computer can understand.

Joystick – an analogue device which feeds signal into a computer which is related to the position which the joystick is occupying; generally used in games programs.

Kilobyte – the unit of memory measurement; one kilobyte (generally abbreviated as K) equals 1,Ø24 bits (a bit is the smallest discrete unit).

Knowledge base – the accumulated knowledge upon which an expert system makes its judgements.

Knowledge engineering – the process by which human expertise is transferred to an expert system.

Knowledge Information Processing Systems – these are the Fifth Generation computer systems which are under development in Japan.

Low-level language – a language which is close to that used within the computer (see High-level language).

Machine language – the step below a low-level language; the language which the computer understands directly.

Memory – the device or devices used by a computer to hold information and programs being currently processed, and for the instruction set fixed within a computer which tells it how to carry out the demands of the program. There are basically two types of memory (RAM and ROM).

Microprocessor – the 'chip' which lies at the heart of your computer. This does the 'thinking'.

Modem – this stands for MOdulator/DEModulator, and is a device which allows one computer to communicate with another via the telephone.

Monitor – (a) a dedicated television-screen for use as a computer display unit, contains no tuning apparatus; (b) the information within a computer which enables it to understand and execute program instructions.

Motherboard – a unit, generally external, which has slots to allow additional 'boards' (circuits) to be plugged into the computer to provide facilities (such as high-resolution graphics, or 'robot

control') which are not provided with the standard machine.

Mouse – a control unit, slightly smaller than a box of cigarettes, which is rolled over the desk, moving an on-screen cursor in parallel to select options and make decisions within a program. 'Mouses' work either by sensing the action of their wheels, or by reading a grid pattern on the surface upon which they are moved.

Network – a group of computers working in tandem.

Numeric pad – a device primarily for entering numeric information into a computer, similar to a calculator.

Octal – a numbering system based on eight (using the digits Ø, 1, 2, 3, 4, 5, 6 and 7).

On-line – device which is under the direct control of the computer.

Operating system – this is the 'big boss' program or series of programs within the computer which controls the computer's operation, doing such things as calling up routines when they are needed and assigning priorities.

Output – any data produced by the computer while it is processing, whether this data is displayed on the screen or dumped to the printer, or is used internally.

Pascal – a high level language, developed in the late 1960s by Niklaus Wirth, which encourages disciplined, structured programming.

Port – an output or input 'hole' in the computer, through which data is transferred.

Program – the series of instructions which the computer follows to carry out a predetermined task.

PILOT – a high level language, generally used to develop computer programs for education.

RAM – stands for Random Access Memory, and is the memory on board the computer which holds the current program. The contents

of RAM can be changed, while the contents of ROM (Read Only Memory) cannot be changed under software control.

Real-time – when a computer event is progressing in line with time in the 'real world', the event is said to be occurring in real time. An example would be a program which showed the development of a colony of bacteria which developed at the same rate that such a real colony would develop. Many games, which require reactions in real time, have been developed. Most 'arcade action' programs occur in real time.

Refresh – the contents of dynamic memories (see memory) must receive periodic bursts of power in order for them to maintain their contents. The signal which 'reminds' the memory of its contents is called the refresh signal.

Register – a location in computer memory which holds data.

Representation – the organisation of information within a computer so that it can be managed by the knowledge base control system.

Reset – a signal which returns the computer to the point it was in when first turned on.

ROM – see RAM.

RS-232 – a standard serial interface (defined by the Electronic Industries Association) which connects a modem and associated terminal equipment to a computer; the QL has two RS-232-C output ports.

S-100 bus – this is also a standard interface (see RS-232) made up of 100 parallel common communication lines which are used to connect circuit boards within micro-computers.

SNOBOL – a high level language, developed by Bell Laboratories, which uses pattern recognition and string manipulation.

Software – the program which the computer follows (see firmware).

Stack – the end point of a series of events which are accessed on a last in, first out basis.

Subroutine – a block of code, or program, which is called up a number of times within another program.

Symbolic inference – see inference system.

Syntax – as in human languages, the syntax is the structure rules which govern the use of a computer language.

Systems software – sections of code which carry out administrative tasks, or assist with the writing of other programs, but which are not actually used to carry out the computer's final task.

Thermal printer – a device which prints the output from the computer on heat-sensitive paper.

Time-sharing – this term is used to refer to a large number of users, on independent terminals, making use of a single computer, which divides its time between the users in such a way that each of them appears to have the 'full attention' of the computer.

Turnkey system – a computer system (generally for business use) which is ready to run when delivered, needing only the 'turn of a key' to get it working.

VLSI – Very Large Scale Integration of components on a 'chip', with present development in Japan and the US aiming for the equivalent of 1Ø million transistors per chip (current chips contain the equivalent of around half a million transistors at most!)

Volatile memory – a memory device which loses its contents when the power supply is cut off.

Word processor – a dedicated computer (or a computer operating a word-processing program) which gives access to an 'intelligent typewriter' with a large range of correction and adjustment features.

Appendix Seven –
The Independent QL User's Group (IQLUG)

This is a very active group, which publishes the monthly newsletter *Quanta*. The group kindly allowed me to include some articles from issues of the newsletter in this book. The source of these articles is given with the articles.

The chairman and newsletter editor is Leon Heller, who has had a lot of experience with the 68000 and the 68006, and has a couple of home-brew systems which use these processors. He has a 68000 cross-assembler running on one of his other systems, which enables him to write 68000 code and run it on the 68000 and 68008.

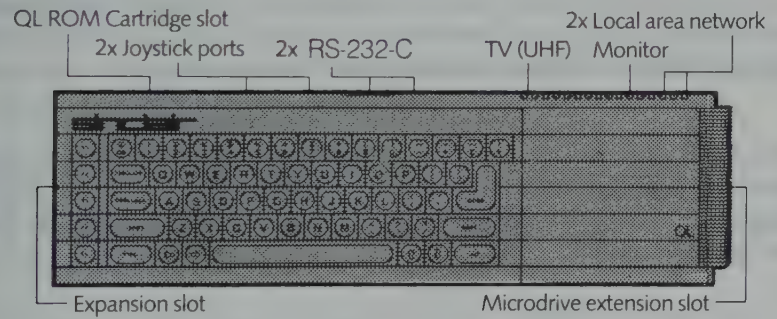
The co-founder of IQLUG is Brian Pain, the secretary, who can be reached at 24 Oxford Street, Stony Stratford, Milton Keynes. He can provide information on the club and on *Quanta*. Brian lectures in accountancy at a college of higher education, and formed what is now the National TRS-80 and Genie Users' Group in the late seventies.

Membership of IQLUG is by subscription to the newsletter, and is open to anyone with an interest in the QL. The club can assist members with problems related to the QL. Workshops are held from time to time at various locations around the country. The club maintains a software library which is available to members.

Appendix Eight – Making Connections

When you unpack your QL, you'll see the computer itself with its user guide, a power supply, aerial lead, network lead, two Microdrive wallets (one with four blank Microdrives in it, and the other containing the Microdrives with the supplied software) and some little plastic feet to put under your QL if you wish to change the angle of the keyboard. You'll probably find – as I did – that the QL is much easier to use if you stick the feet on.

Hidden at the back of your QL is a number of slots and connectors, where you plug in the ROM cartridges, printers, joystick and the like.



If you turn the computer around to look at these connectors, you'll see they are for the following uses (starting from the left hand end as you look at the back of the machine):

- **networking**
- **networking**
- **power**
- **RGB monitor output**
- **UHF television output**
- **serial port one**
- **serial port two**
- **control one (joystick one)**
- **control two**
- **ROM cartridge**

There is an expansion slot on the right hand side (looking from the back) to take a half megabyte memory board, or other peripherals.

The network slots allow you to link together up to 64 QL's in what is called a 'local area network' (QLAN, in Sinclair-speak). The various computers in the network can 'talk' to each other at 1ØØK baud (baud is the rate of transmission of information and to show how fast this is, consider that the Spectrum reads programs off cassette at a baud rate of approximately 1.5K). Data can be sent over the local area network to any computer which is currently 'listening' for it.

At the front, on the right hand side, are the slots for your Microdrives. Don't put a Microdrive in its slot before you turn the QL on. If you do this, you run the risk of corrupting the data stored on the Microdrive. When a Microdrive is spinning, and the QL is reading from it, the light beside that slot comes on. Don't attempt to take a Microdrive out of its slot when the light is on or you may well damage both the computer and your Microdrive.

The RGB port is for connecting an RGB monitor. If you've only ever run your QL on a domestic television, you'll be in for a very pleasant surprise when you hook it up to an RGB monitor. The picture is extremely clear.

Appendix Nine – Further Reading

There are a number of books in print on the QL. The ones I have found most useful are:

An Introduction to Programming the Sinclair QL – R A & J W Penfold (Babani, Ø 85934 125 9, £1.95); and

Good Programming with QL SuperBASIC – Roy Atherton (Longman, Ø 582 29662 5, £5.95).

Also of interest are:

Quill, Easel, Archive and Abacus on the QL – A McCallum-Varey (Sunshine, Ø 946Ø8 55 6, £6.95).

Advanced Programming with the Sinclair QL – Martin Gandoff (Hutchinson, Ø Ø9 158961 4, £6.95).

General books of interest include:

Tim Hartnell's QL Games Compendium – Tim Hartnell (Interface, Ø 947695 Ø4 4, £5.95).

The Art of Structured Programming – Peter Juliff (Interface, Ø 9Ø7563 79 1, £5.95).

NOTES

NOTES

NOTES

NOTES

NOTES

From Tim Hartnell, the author *Personal Computer World* magazine called 'Mr Sinclair', comes this essential work for all Sinclair QL programmers.

Step by step, element by element, Tim Hartnell leads you through SuperBASIC, showing what an exciting, important language it is. You'll see how to adapt your current programming skills and style to take advantage of the elaborate structured possibilities of SuperBASIC. And because each command is demonstrated with programs, you'll be building a library of worthwhile game and utility programs as you go.

Among the treasures in this book is a machine-code multi-tasking clock, which runs in the bottom right hand corner of your screen *all the time*, without interfering with anything else you're doing. With *QLogo*, you'll give your QL a version of Logo which provides a significant number of 'standard' Logo comman and which can even be controlled in direct entry mode. Th *Engine*, a mini-Forth written in SuperBASIC, will allow yo teach yourself Forth on the QL, without having to buy an extra language!

There is much more in this book, including a host of game major section on creating adventures (complete with a m 20K adventure for you to solve), a program for modellin financial projections, file-handling and a routine to defin own characters, along with an introduction to the 68008 processor. It's all here in this essential work.

KS-361-788

another great book from
Interface

£7.95

ISBN 0-947695-19-2

9 780947 695194